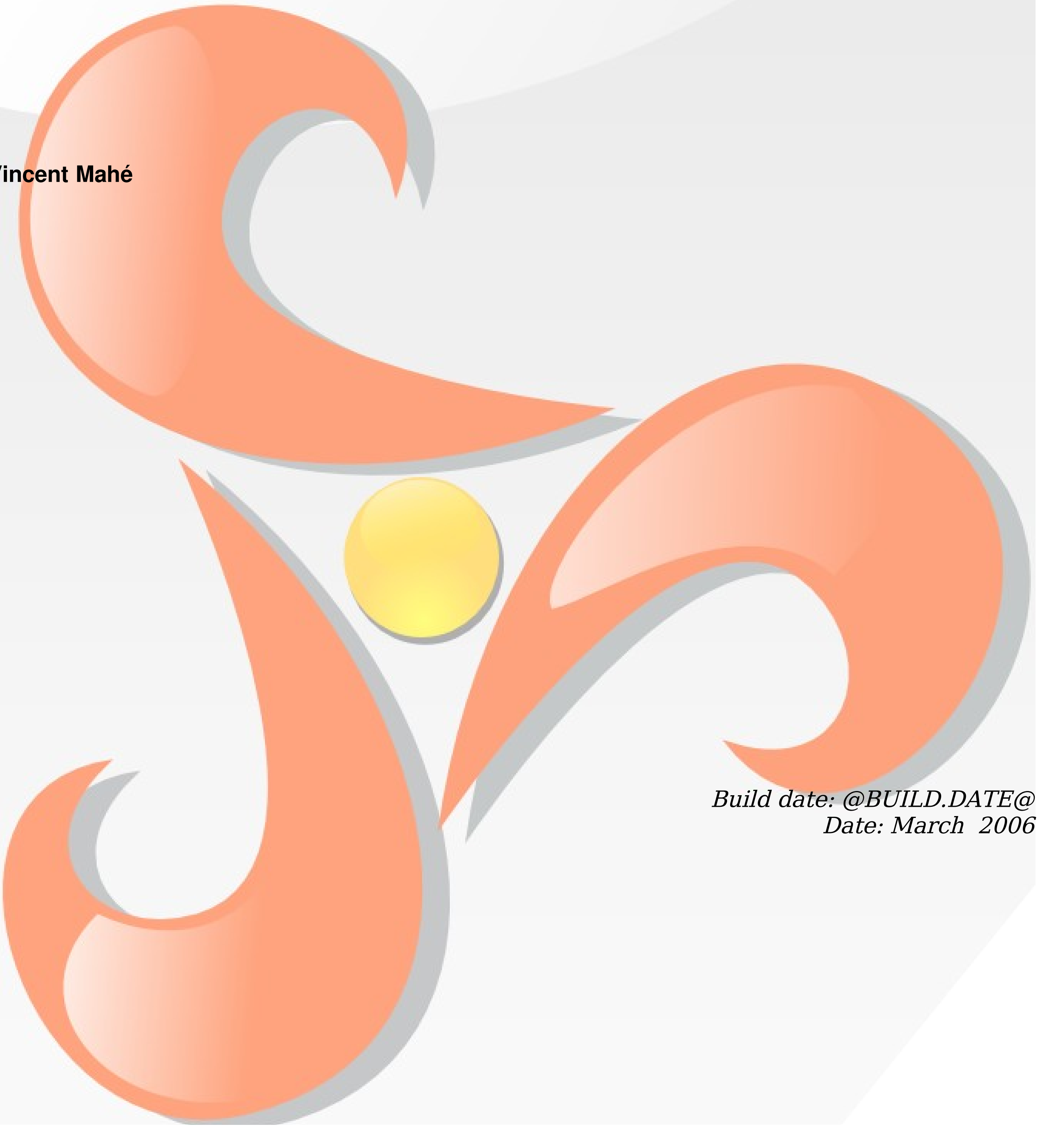
Build date: @BUILD.DATE@

Vincent Mahé

INRIA

Umlaut : Demonstrations



Vincent Mahé

Build date: @BUILD.DATE@
Date: March 2006

Metadata

to be filled by the author

The metadata gather information allowing a better spreading of the informations through various media: databases, cd, internet and intranet repositories...

The following information will be considered as metadata.

Document information	
Title	Umlaut : Demonstrations
Subtitle	
Volume number	
Series number	
Keywords	Kermeta, metamodeling, UML, transformation
More information	Licence EPL
Document date	17/03/2006
Status	
Access conditions	
Access conditions revision Date	17/06/2006
Contract number	
ISRN	

Conference	
Title	title
References	Reference
Sponsor	sponsor
Dates	
Address	

Authors

Author N°1

Firstname	Vincent
Name	Mahé
Role	
Quality	
Corporation	Université de Rennes 1
Corporation acronym	
Corporation division	Triskell
Address	vmahe@irisa.fr

Author N°2

Firstname	
Name	
Role	
Quality	
Corporation	
Corporation acronym	
Corporation division	
Address	

Author N°3

Firstname	
Name	
Role	
Quality	
Corporation	
Corporation acronym	
Corporation division	
Address	

Corporate Authors

Corporate Author N°1

Corporation name	INRIA
Corporation acronym	
Corporation division	Triskell
Address	
Post office box	
ZIP code	
City	
State	France
Telephon	
Fax	
E-mail	
Web site	http://www.inria.fr

Contract Sponsors

Contract sponsor N°1

Corporation name	
Corporation acronym	
Corporation division	
Address	
Post office box	
ZIP code	
City	
State	
Telephon	
Fax	
E-mail	
Web site	

Organisme Commanditaire N°2

Corporation name	
Corporation acronym	
Corporation division	
Address	
Post office box	
ZIP code	
City	
State	

Telephon

Fax

E-mail

Web site

Abstract

This document presents some basic demonstrations about various aspects of the KerMeta Umlaut tools. It intends to be a introduction for anybody who want to use Umlaut.

Preface

Kermeta is a language dedicated to build executable meta-models as MOF is defined to build meta-data models. Some Kermeta tools are dedicated to specific model manipulations, like the Umlaut tool. So the aim of this short document is to present the use of this Umlaut. Umlaut has been design to process executable UML2 models (implying state charts, static diagrams & instances diagrams) for validation purposes. It authorizes interactive simulation of the machine and will link to CADP tools suite (automatic generation of test code,...).

Kermeta and its tools are evolving software and despite that we put a lot of attention to this document, it may contain errors (more likely in the code samples). If you find any error or have some information that improves this document, please send it to us using the bugtracker in the forge: http://gforge.inria.fr/tracker/?group_id=32 Last check: v0.1.0

The most uptodate version of this document is available online from http://www.kermeta.org .

Table of contents

1 Introduction to UMLAUT tool

This document presents some obvious cases of use of the Umlaut tools, for demonstrate its goals and introduce its common use.

1.1 Presentation

UMLAUT (**UML** **A**ll p**U**rpose **T**ransformer) was first a test-purposed tool designed independently by the Triskell research team, with a UML 1.3 embedded version. As its use and maintenance were difficult and the UML change to the 2.0 version, Triskell team decided to rewrite its main features upon the Kermeta environment. Kermeta is a metamodeling language which allows describing both the structure and the behavior of models. It has been designed to be fully compliant with the OMG EMOF metamodel (part of the MOF 2.0 specification) and provides an action language for specifying the behavior of models. As UML2 metamodel exist and can be processed by Kermeta (under an ecore form), UMLAUT can be focused on UML processing tasks. Written in Kermeta (which is itself available as a Eclipse plug-in), this tool is available on all Java platforms, solving maintenance and usability problems of the first version.

Kermeta is intended to be used as the core language of a metamodel oriented platform (a Metamodel Development Kit). It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language.

1.2 Principles underlying Umlaut

Software engineering faced problems since its birth, which increase with the number of code lines and the multiplication of hardware platforms. Large research work is done to reduce these problems, giving new concepts which need to be embedded in engineering tools for a easier use by effective software designers.

Three advanced concepts are addressed by Umlaut:

- **Formal Description Techniques** (FDT) with UML: they are mathematical approach of specification, design and verification of software. The specifications are written such a way the design can be validate, then the implementation conformance to the design is verified.

  With regards to that goal, Umlaut offers a UML2 models simulator which permits to validate the analysis description of a planned computer system in the early stages of a software life cycle.

- OMG **Model Driven Architecture** (MDA): this concept divides the design of software in two parts which are the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The PIM is designed by the engineer without implying a piece of hardware, and the PSM may be derived from the PIM by application of hardware specificities.

  Umlaut proposes many useful UML2 tools like extraction of interfaces or cloning to realize such transformations. Kermeta, the framework underlying Umlaut, has been designed as a meta-modeling tool for doing models transformations.

- **Prototyping by executable UML models**: as customers needs and demands growth in size and complexity, engineers must improve their capture of end-users requests. So they present prototype of the planned system to the persons who will use it (or pay for it) in earlier stages of the engineering process, in order to avoid mistakes in the more expensive stages.

  Umlaut embeds a "uml2km" tool which transforms a UML2 static model in a Kermeta equivalent. As Kermeta is a executable language, the KM model corresponding to the

UML one can be ran and the user have a look on its behavior for comments.

The Valooder and its simulator embed all those principles and their corresponding Umlaut features so the demonstration we present for the complete Valooder tool is useful to all ways you may use the UML2 MDK the UMLAUT is.

1.3 Technical background

The all UMLAUT relies upon the Kermeta integrated environment and the EMF and XMI plug-ins for Eclipse IDE.

As some bugs are solved in the last versions of Kermeta, you need to run UMLAUT on the later Kermeta runtime, which needs the 3.2 version of Eclipse and corresponding versions of EMF and XMI (2.1).

Since the Eclipse 3.2 version and the corresponding EMF and XMI plug-ins, you must reference the `UML2.ecore` meta-model into your local repository, by:

-> right click on the meta-model file

-> Kermeta

-> Register EPackages into repository

This avoids you a "`kermeta::persistence::ResourceLoadException`" with:

```
EMF Loading error : could not find a class (uml2::Model) in
loaded libraries. Please check your require statements
```

But the UMLAUT has been designed on the UML 2.0 version, when the UML2 plug-in in Eclipse 3.2 is about the UML 2.1 version, with some changes. As a format of serialization, the XMI changed deeply between Eclipse 3.1.2 and 3.2.

So you must use the Eclipse 3.1.2 to create and save UML2 models

And Eclipse 3.2 to run Kermeta for their transformations

We will switch to UML 2.1 as soon as we can because the new version of the TopCased graphical editor embeds all the UML diagrams needed for the Valooder (static diagram, state machine, use case, instances diagram) and will be very useful for UMLAUT users.

2 UML simulator

Valooder is one of those possible uses, dedicated to models testing. Its goals are to offer general ways to simulate and test the behavior of the future system, and generate the tests suite which will be ran on the implementation of this system.

In a nutshell, the simulator [**will**] offers (within the Valooder transformation) :

- a interactive state machines simulator (Kermeta executable model)
- a I/O LTS compatibility, which give a way for plug the machine into the CADP tests suite through of code generation (Eiffel to C to CADP)
- *a code generator (Kermeta to Eiffel/Java)* [**not yet implemented**]
- *a model checker (with CADP)* [**not yet implemented**]

For simulation or validation of the designed system, UMLAUT needs some informations about it, presented as UML2 models :

- a Class diagram to describe involved elements and their relations
- a States Chart diagram for each element of the system we want to study the behavior (one states chart by static class, including the external actor)
- a Use Case to present and identify the external actors who manipulate the system and induct a behavioral response from the system

All these diagrams must be embedded in an UML2 model, which can also contains some graphical representations of those diagrams.

- an Instances diagram for each initial state of the system to be tested or simulated, describing what instances are involved and their initial values.

2.1 An explicit example

The simulator is dedicated to process models expressing behavior through state machines so we begin with a simple example about a electrical 3-way switch.

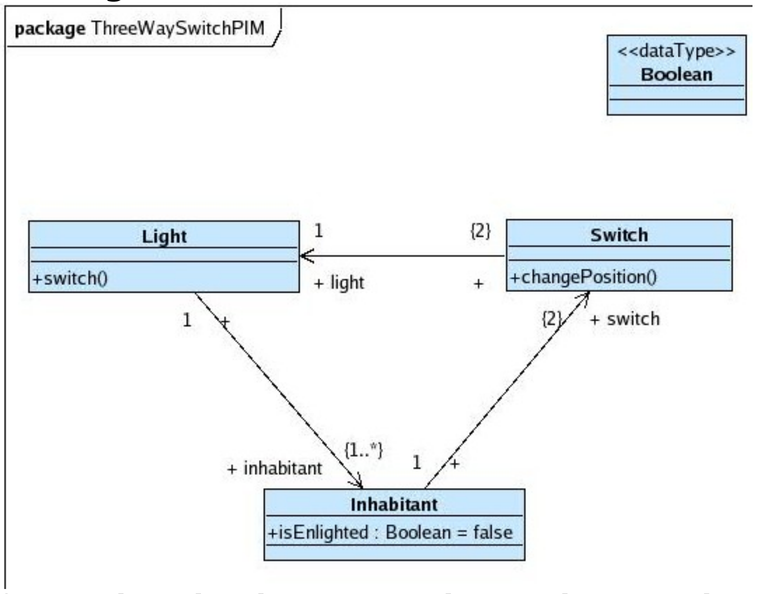
Everybody knows the home case of a light operated by two switches : one change of any of these switches' position changes the state of the light. Its an easy mean to get enlighten a stairwell or a long hall from each of its extremities. The inhabitants can switch On the light when they enter the room or the hall from one of its parts and to switch Off when they exit by any other part of this place.

For our demo purpose, the system can be described as 2 switches (each with 2 states : “up” & “down”), 1 light (with 2 states : “On” & “Off”), and 1 inhabitant who will operate the switches.

2.2 PIM model

As blackbox validation of implementations is one of the UML simulator goals (through bi-simulation and test generator), we must write the class diagram and the instances diagram as a Plateform Independent Model, because a 3-Way Switch could be implemented by different hardwares (hard wires, but also wireless, radio commands, solar cells accumulator...). PIM version of diagrams would be like the next diagram.

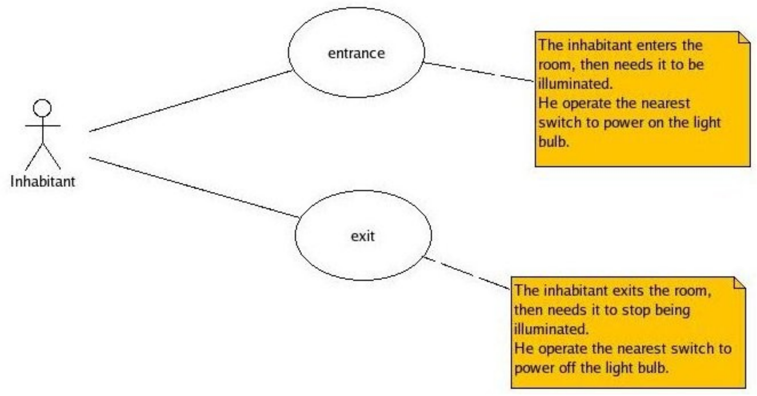
2.2.1 Classes diagram



Actors must be referenced on this diagram, as the simulator need to know where they act the system. Here, the external actor is an Inhabitant which manipulate the switches and can switch up and down instances of Switch Class.

You may notice that the main classes have no attributes describing their inside situation, as this will be assumed by the different states of their own statemachine (look at the statecharts diagrams further). Having both an attribute and a state would imply a copying of information (or mistakes if forgotten). As the status of the instance is insured by the state machine itself, the firing of a transition which depends on a condition upon this variable will be driven by the states, by construction. The initial state of a class is given (enforced) by its state machine diagram.

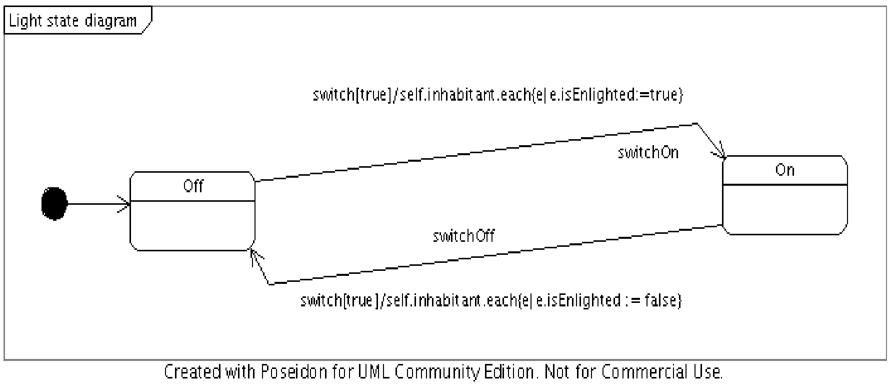
2.2.2 Use Case diagram


entrance
The inhabitant enters the room, then needs it to be illuminated.
He operate the nearest switch to power on the light bulb.
Inhabitant
exit
The inhabitant exits the room, then needs it to stop being illuminated.
He operate the nearest switch to power off the light bulb.

This diagram is needed because of the Actor(s) it defines. The Actor(s) will activate classes of the Class diagram. Valooder simulator presents the system to the user as he would be the Actor and demand him to choice between possible transitions at each step of the simulation.

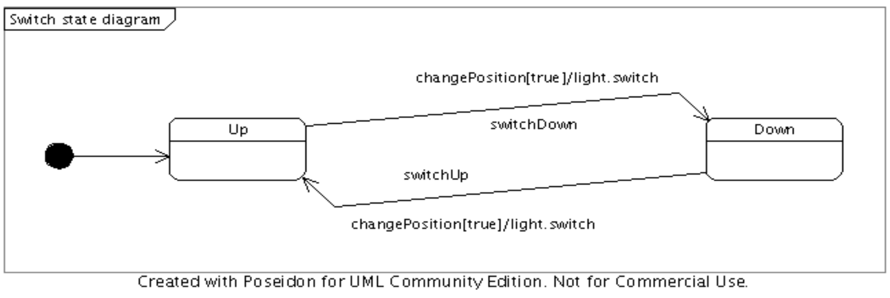
2.2.3 State charts diagrams

Light class states machine :


Light state diagram
switch[true]/self.inhabitant.each{e| e.isEnlighted:=true}
switchOn
Off
On
switchOff
switch[true]/self.inhabitant.each{e| e.isEnlighted := false}
Created with Poseidon for UML Community Edition. Not for Commercial Use.

The Light could have two values (excluding each other) : illuminated or not. It is drawn by the two states “On” and “Off”.

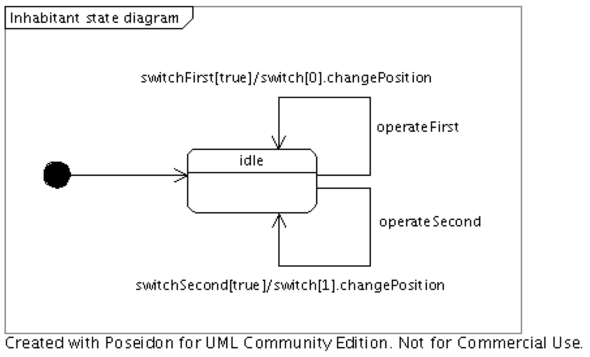
Switch class states machine :


Switch state diagram
changePosition[true]/light.switch
switchDown
Up
Down
switchUp
changePosition[true]/light.switch
Created with Poseidon for UML Community Edition. Not for Commercial Use.

A Switch only have two positions, drawn by the two states “Up” and “Down”.

Inhabitant class states machine :

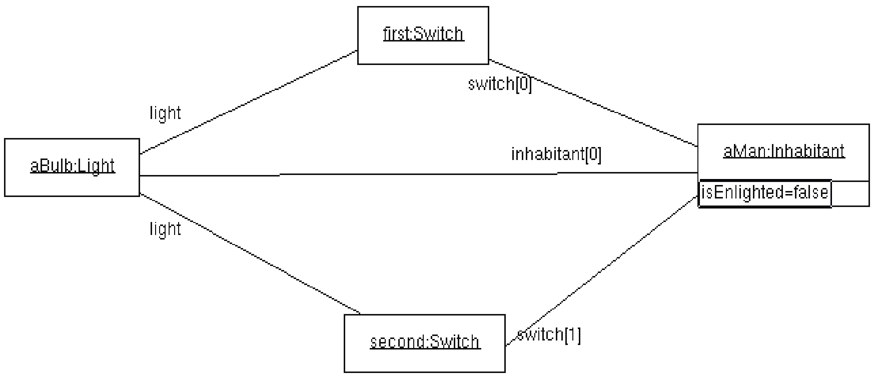
May be you notice that some couples state/action are not drawed on those states


Inhabitant state diagram
switchFirst[true]/switch[0].changePosition
operateFirst
idle
operateSecond
switchSecond[true]/switch[1].changePosition
Created with Poseidon for UML Community Edition. Not for Commercial Use.

charts, like the action *switchOff()* from the *Off* state of the Light. Those indeterminate cases are "semantic variation points", where choices need to be done by the designer of the system in order to run it inside UMLAUT tools (simulator).

2.2.4 Instances diagram of initial system

We will test a given initial situation and try to validate the model by running it through the simulator. We hope our model behaves as expected, and the simulation will ensure us (or dismiss our model).



You can refer to the corresponding state machines to know the initial values of the instances for the values depending on state machines. Other attributes must be explicitly valued, like the "isEnlighted" variable of Inhabitant instances.

2.2.5 Transitions constraints

UMLAUT needs to know the conditions which authorize each transition presented in the states charts. Despite UMLAUT is an UML2 environment, you must specify those conditions in Kermeta language, as the executable reified model will be generated in this language.

The inexplicit cases would be presented to the designer as semantic variation points needing a choice by the UMLAUT user.

2.2.6 Blackbox view

For a user of the switches and light, the expected behavior of the system can be drawn with the next two predicate :

- if the light is *Off*, changing one switch position put the light *On* so inhabitants became lighted.
- if the light is *On*, changing one switch position put the light *Off* so inhabitants became unlighted.

2.3 A model of this example

The Kermeta environment is based on OMG standards. It can read and write any model in the OMG XMI format for UML2.

For UML2, the file format is :

```
uml:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:uml="http://www.eclipse.org/uml2/1.0.0/UML"
```

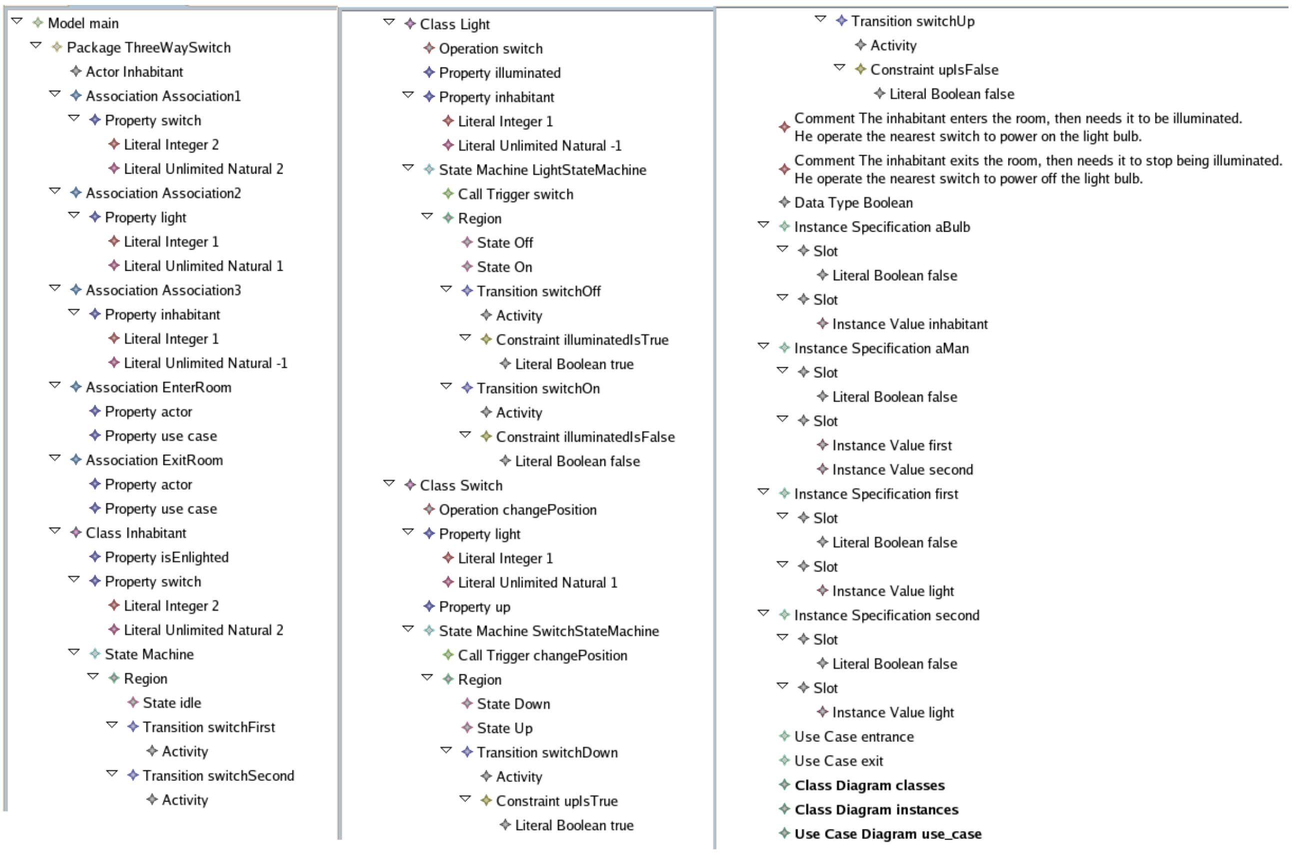
We used TopCased (plug-in for the Eclipse IDE) UML2 graphical editor to draw the static diagram, because it is currently the only free tool to save the models it produces in the right file format. As TopCased includes only static, instances and use cases diagrams for the moment (spring 2006), we wrote the model beginning with those graphs and manually complete it with state charts and links between them in the Eclipse Reflexive editor ("outline" window of TopCased).

We used Poseidon UML2 editor (Community edition) for drawing other diagrams for this document (but this tool can't save its UML models in the correct format).

2.3.1 XMI version of the model

You should note that UML2 modeling is quite sophisticate so recording some features is not close to evidence. Links between instances are intermediated by Slot elements, for example, and transitions are referenced in states through Activity objects. Those syntax depths are masked by graphic tools, but TopCased doesn't offer such a functionality for states charts from know.

In the reflexive editor, the complete XMI model would seem like that :

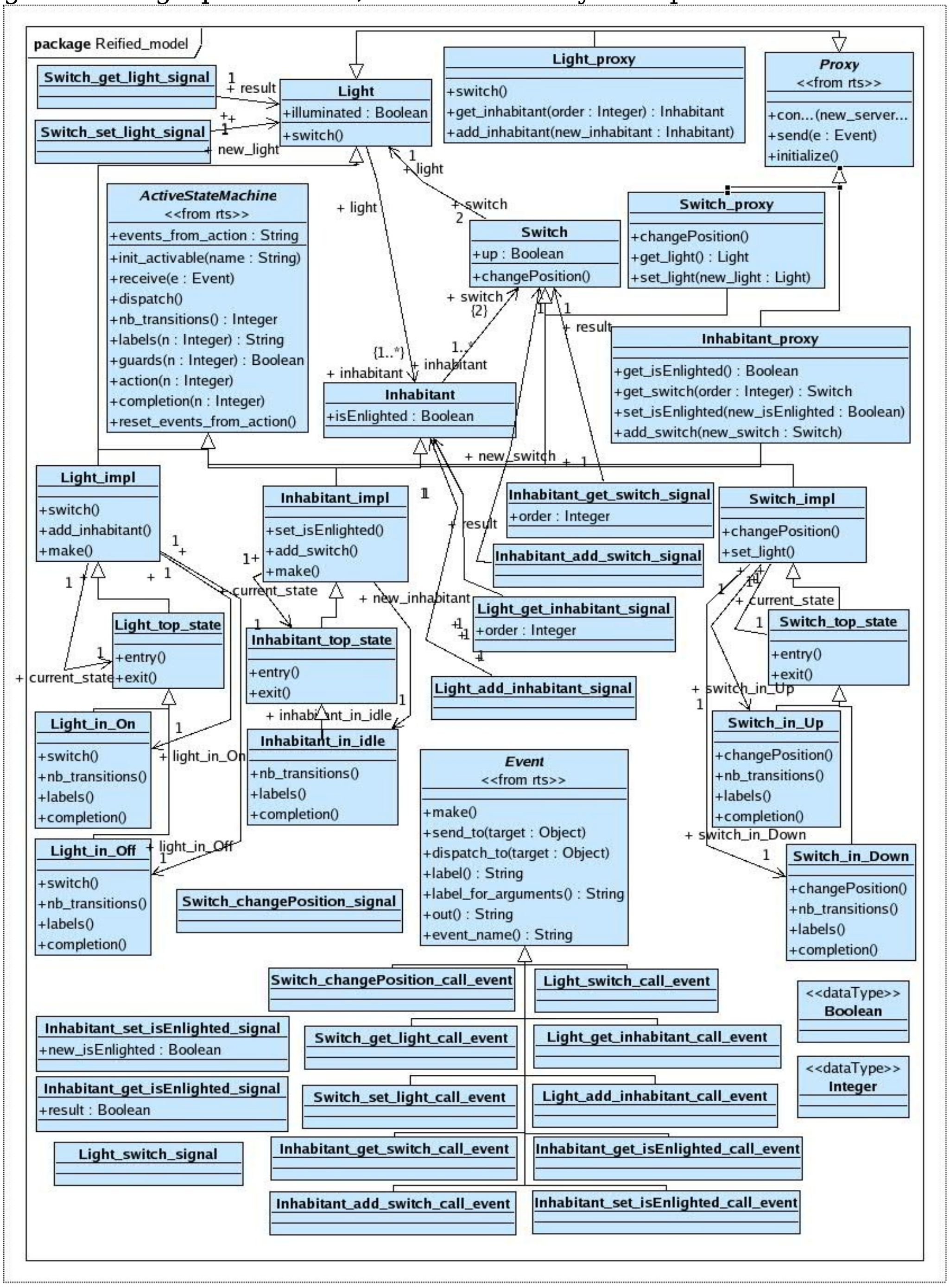


```
Model main
  Package ThreeWaySwitch
    Actor Inhabitant
    Association Association1
      Property switch
        Literal Integer 2
        Literal Unlimited Natural 2
    Association Association2
      Property light
        Literal Integer 1
        Literal Unlimited Natural 1
    Association Association3
      Property inhabitant
        Literal Integer 1
        Literal Unlimited Natural -1
    Association EnterRoom
      Property actor
      Property use case
    Association ExitRoom
      Property actor
      Property use case
    Class Inhabitant
      Property isEnlighted
      Property switch
        Literal Integer 2
        Literal Unlimited Natural 2
      State Machine
        Region
          State idle
          Transition switchFirst
            Activity
          Transition switchSecond
            Activity
    Class Light
      Operation switch
      Property illuminated
      Property inhabitant
        Literal Integer 1
        Literal Unlimited Natural -1
      State Machine LightStateMachine
        Call Trigger switch
        Region
          State Off
          State On
          Transition switchOff
            Activity
            Constraint illuminatedIsTrue
              Literal Boolean true
          Transition switchOn
            Activity
            Constraint illuminatedIsFalse
              Literal Boolean false
    Class Switch
      Operation changePosition
      Property light
        Literal Integer 1
        Literal Unlimited Natural 1
      Property up
      State Machine SwitchStateMachine
        Call Trigger changePosition
        Region
          State Down
          State Up
          Transition switchDown
            Activity
            Constraint upIsTrue
              Literal Boolean true
          Transition switchUp
            Activity
            Constraint upIsFalse
              Literal Boolean false
    Comment The inhabitant enters the room, then needs it to be illuminated.
    He operate the nearest switch to power on the light bulb.
    Comment The inhabitant exits the room, then needs it to stop being illuminated.
    He operate the nearest switch to power off the light bulb.
    Data Type Boolean
    Instance Specification aBulb
      Slot
        Literal Boolean false
      Slot
        Instance Value inhabitant
    Instance Specification aMan
      Slot
        Literal Boolean false
      Slot
        Instance Value first
        Instance Value second
    Instance Specification first
      Slot
        Literal Boolean false
      Slot
        Instance Value light
    Instance Specification second
      Slot
        Literal Boolean false
      Slot
        Instance Value light
    Use Case entrance
    Use Case exit
    Class Diagram classes
    Class Diagram instances
    Use Case Diagram use_case
```

Each transition has its "source" and "target" properties fixed to the correct state.

2.4 Manually reified model

As Valooder transformation builds only UML2 XMI reified models, we present the resulting model in graphical form, drawn manually in TopCased UML2 modeler.



2.5 Running it in the simulator

2.5.1 First validation test : one man

After having passed the model in the Valooder tool, you get an executable version of the reified model in the Kermeta language (look at the file “*three_way_switch.kmt*” to see it, as written by hands, in the “*uml2.samples*” package).

For being able to running it you need to associate it with a simulator interface, as it's done in the “*three_way_switch_simulator.kmt*” program.

Run it as a Kermeta application. You're expected to see the following output :

```
 .... Running the simulator ....


===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb'
 * object 'second'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb'
 * object 'aBulb'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : ♦
```

The simulator user interface has given you the starting status of all the instances you have put in the initial model, then presented the fireable transitions.

The “aMan” is not enlighten, the 2 switches are up and the “aBulb” is off.

The simulator asks you for the one you want to fire and is waiting for the answer.

The man is out a room, and enters it. He needs some light and will use the nearest switch. Then you choose the first transition, giving “0” to it. The simulator fires it and emits the corresponding event, which could itself fire an internal transition (emitting a new event...) or affect a value to an attribute of one instance. After playing all internal transitions, it will give you the new status of all the instances, and the new possible actions, like this :

```
   You choose the '0' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = true
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Down) :
     - light = proxy to 'aBulb'
 * object 'second'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb'
 * object 'aBulb'  (in state : light_in_On) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : ♦
```

You can see that many changes occur during the run step. The “aMan” is now enlighten, the switch “first” is down, and the light “aBulb” is on.

Now the man decides to go out the room, using the other switch. So you choose the “1” transition :

```
   You choose the '1' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Down) :
     - light = proxy to 'aBulb'
 * object 'second'  (in state : switch_in_Down) :
     - light = proxy to 'aBulb'
 * object 'aBulb'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : ♦
```

You can see the switch “first” hadn't change from the previous step, and the switch “second” is now down. The light is off and the man in darkness.

So the model seems to have the behavior we can expect from a 3-way switch.

You can enter “q” for ending the simulation :

```
  --> give the choosen transition number ( 'q' for quit) : q

Exiting the simulator
```

2.5.2 An other validation test : two bulbs

The first initial system had only one instance of “Light” using the model.

But we may (and must) interrogate ourselves : what will occur if two lights are actioned by our 3-way switch system?

You can add another “Light” to the initial instance model or give another instance model with two “Light” instances as initial configuration.

We did it for you and have got the *“three_way_switch_2bulbs.kmt”* file and its corresponding simulator can be ran through the *“three_way_switch_2bulbs_simulator.kmt”* program :

```
 .... Running the simulator ....


===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb2'
 * object 'second'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb2'
 * object 'aBulb1'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) :  ♦
```

Let “aMan” enter in the room (choice '0'). You obtain the following result :

```
   You choose the '0' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = true
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Down) :
     - light = proxy to 'aBulb2'
 * object 'second'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb2'
 * object 'aBulb1'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_On) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : ♦
```

What a mistake! The simulator tells you that the “aBulb2” has switched On, not the “aBulb1”. That's not what was expected.

Continue running the model by leaving the room by the same door :

```
   You choose the '0' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb2'
 * object 'second'  (in state : switch_in_Up) :
     - light = proxy to 'aBulb2'
 * object 'aBulb1'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : q

Exiting the simulator
```

Now, the two lights are Off. If you continue to run the model, you will see only the “aBulb2” activated. May be you noticed that only “aBulb2” is known by each of the two switches.

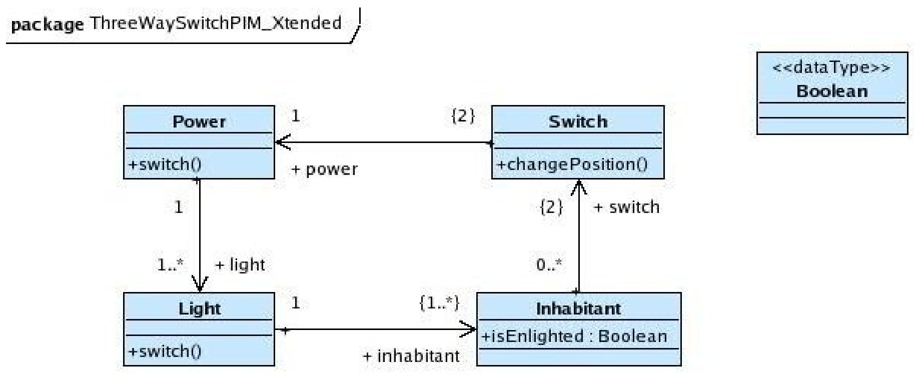
So has the simulator failed running the model?

You can see an ambiguous formulation of the problem : the model didn't offer a way for embed of multiple lights.

So the correct answer about a simulator failure is NO, definitely NO. Its purpose IS to detect such model lacks, and it did it just the way it is supposed to do. The simulator offered us a clear view of a model lack/failure we must correct in order to get a better design and avoid implementation mistakes.

2.5.3 An enhanced model

We write a more sophisticated model in order to allow multiple lights. It comes with a “Power” virtual entity which represents the links between switches and light bulbs. It can be a classical wire, or solar cells actioned by infrared detectors (the so-called switches).



We can process the new model into UMLAUT simulator, with two bulbs like the previous try.

```
 .... Running the simulator ....


===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Up) :
     - power = proxy to 'power'
 * object 'second'  (in state : switch_in_Up) :
     - power = proxy to 'power'
 * object 'aBulb1'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'power'  (in state : power_in_Off) :
     - light(0) = proxy to 'aBulb2'
     - light(1) = proxy to 'aBulb2'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : 0
```

Let “aMan” enter in the room (choice '0'). You obtain the following result :

```
   You choose the '0' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = true
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Down) :
     - power = proxy to 'power'
 * object 'second'  (in state : switch_in_Up) :
     - power = proxy to 'power'
 * object 'aBulb1'  (in state : light_in_On) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_On) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'power'  (in state : power_in_On) :
     - light(0) = proxy to 'aBulb2'
     - light(1) = proxy to 'aBulb2'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : 1
```

Good! The simulator tells you that the two bulbs have switched On, not only the “aBulb2”. That is what we expected.

Continue running the model by leaving the room by the same door :

```
   You choose the '1' transition

===================================
    Current state of the system
===================================

 * object 'aMan'  (in state : inhabitant_in_idle) :
     - isEnlighted = false
     - switch(0) = proxy to 'first'
     - switch(1) = proxy to 'second'
 * object 'first'  (in state : switch_in_Down) :
     - power = proxy to 'power'
 * object 'second'  (in state : switch_in_Down) :
     - power = proxy to 'power'
 * object 'aBulb1'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'aBulb2'  (in state : light_in_Off) :
     - inhabitant(0) = proxy to 'aMan'
 * object 'power'  (in state : power_in_Off) :
     - light(0) = proxy to 'aBulb2'
     - light(1) = proxy to 'aBulb2'

Fireable transitions :
   -- 0 -- aMan -> !switch.elementAt(0).changePosition
   -- 1 -- aMan -> !switch.elementAt(1).changePosition
  --> give the choosen transition number ( 'q' for quit) : q

Exiting the simulator
```

Now, the two lights are Off. If you continue to run the model, you will see the two bulbs activated. So our new model embeds finely the need for multiple lights.

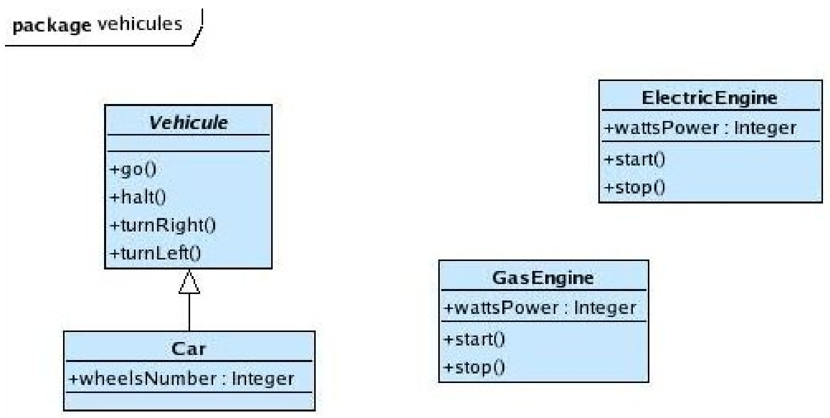
3 UML transformations

The O.M.G. “Model Driven Architecture” is based upon a major distinction between the analysis and design levels in software conception. The analysis work is supposed to produce platform independent models (P.I.M.) and the design level generates (in an automated way if possible) more detailed models which include the specificities of the platform (hardware and operating system) they target. There is only one PIM, which embeds all the domain needed particularities. It can be derived in many PSM, one for each targeted system.

The given transformations are useful for designers who need to automatize the path from a PIM analysis model to a PSM design model.

3.1 A simple example

Take a simple model describing cars structure. This model is too simple so it needs to be enhanced.



We can write a Kermeta program which will transform this model.

3.2 A simple transformer

This transformer needs access to the UML tools then it must load the original model.

As this model must stay inviolate, the first task is the cloning of this model into a equivalent one which will be the final result.

The “vehicules” model needs an engine interface the two engine classes will inherit from, an accessor on the “wattsPower” attribute, a composition link between the “Car” and the “GasEngine” classes.

When new added elements must be named, the tool has naming methods which

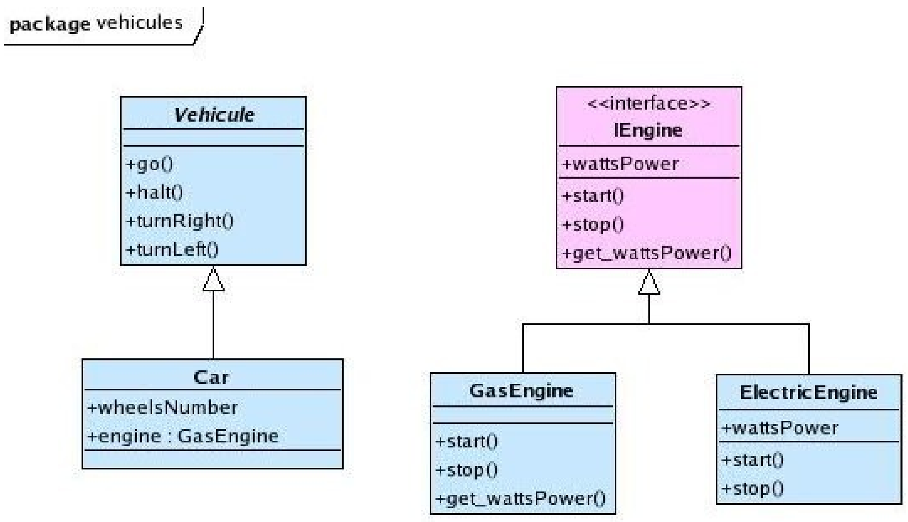
can be overwritten in order to specialize this naming. To do such a modification, create your own tool by inheritance of the generic one and rewrite the good operations (look at `SpecializedEngineInterface_Tool` Class in the example).

```
package vehicules;

require kermeta
require "../../../fr.irisa.triskell.uml2/src/kermeta/transformations/StaticTools.kmt"
require "../../../fr.irisa.triskell.uml2/src/kermeta/transformations/UmlCloneFactory.kmt"

class Main
{
    reference origModelName : kermeta::standard::String
    reference origModel : uml2::Model
    reference cloneModelName : kermeta::standard::String
    reference cloneModel : uml2::Model
    reference root_package : uml2::Package

    /**
     * This sample program will preserve the given sample model
     * in order to be reused as many times as you want.
     * So the program loads the model, clones it,
     * and then does the transformations on the clone.
     * You can then open the resulting model in the Ecore editor
     * or use a graphical editor (like TopCased) to draw it.
     */
    operation main() : Void is do

        //creating needed instances
        origModelName := "../models/vehicules.uml2"

        // load the given UML2 resource
        var inputRepository : kermeta::persistence::EMFRepository init kermeta::persistence::EMFRepository.new
        var inputResource : kermeta::persistence::EMFResource
        inputResource ?= inputRepository.createResource(origModelName, "../ecore/UML2.ecore")
        inputResource.load()

        // "instances" only gives the main diagramm package for the targeted code tree
        origModel ?= inputResource.instances.one

        var factory : uml2::transformations::UmlCloneFactory init uml2::transformations::UmlCloneFactory.new
        factory.initialize
        cloneModel := factory.cloneStaticDiagram(origModel)

        // we need the package containing the classes
        root_package ?= cloneModel.ownedMember.select{ e | e.name == "vehicules" }.one

        addGetterOnWattsPower
        addEngineInterface
        addCarEngineComposition
        addEngineGeneralisation

        cloneModelName := "../models/improved_vehicules.uml2"
        var outputRepository : kermeta::persistence::EMFRepository init kermeta::persistence::EMFRepository.new
        var outputResource : kermeta::persistence::EMFResource
        outputResource ?= outputRepository.createResource(cloneModelName, "../ecore/UML2.ecore")
        outputResource.instances.add(cloneModel)
        outputResource.save()
    end

    /////////////  code for transformations /////////////////
    operation addGetterOnWattsPower() is do
        // we get the "GasEngine" class (before this class will loose its attributes, passed to the interface)
        var cl : uml2::Class
        cl ?= root_package.ownedMember.select{ e | e.name == "GasEngine" }.one

        // we need the accessors tool for transformation
        var accTool : uml2::transformations::Accessor_Tool init uml2::transformations::Accessor_Tool.new

        // we make the getter corresponding to each property of the class
        cl.ownedAttribute.each{ prop |
            // we add the getter operation corresponding to the property
            accTool.addGetter(cl, prop)
        }
    end
    operation addEngineInterface() is do
        // we get the "GasEngine" class to generalize it in a "IEngine" interface
        var cl : uml2::Class
        cl ?= root_package.ownedMember.select{ e | e.name == "GasEngine" }.one
        // we need the interface tool for extraction
        var ifTool : SpecializedEngineInterface_Tool init SpecializedEngineInterface_Tool.new

        // we get then the interface
        var ifc : uml2::Interface
        ifc := ifTool.extractInterface(cl)

        // we add the new interface to the resulting model
        root_package.ownedMember.add(ifc)
        // Rem: the generalization between the interface and the (original)
        // concrete class doesn't need to be added to the model
        // (it is owned by the concrete class)
    end
    operation addCarEngineComposition() is do
        // we get the "GasEngine" class as the component
        var clEngine : uml2::Class
        clEngine ?= root_package.ownedMember.select{ e | e.name == "GasEngine" }.one

        // we get the "Car" class as the container
        var clCar : uml2::Class
        clCar ?= root_package.ownedMember.select{ e | e.name == "Car" }.one

        // we need the association tool for transformation
        var assocTool : uml2::transformations::Association_Tool init uml2::transformations::Association_Tool.new

        // we then get the new composition
        var assoc : uml2::Association init uml2::Association.new
        assoc := assocTool.addNavigableComposition(clCar, clEngine, "poweredBy", "engine")

        // we add the new association to the resulting model
        root_package.ownedMember.add(assoc)
    end
    operation addEngineGeneralisation() is do
        // we get the "ElectricEngine" class
        var cl : uml2::Class
        cl ?= root_package.ownedMember.select{ e | e.name == "ElectricEngine" }.one

        // we get then the "IEngine" interface
        var ifc : uml2::Interface
        ifc ?= root_package.ownedMember.select{ e | e.name == "IEngine" }.one

        // we need the generalization tool for transformation
        var geneTool : uml2::transformations::Generalization_Tool init
uml2::transformations::Generalization_Tool.new

        // we make the inheritance
        geneTool.addGeneralization(ifc, cl)
        // Rem: the generalization between the interface and a
        // concrete class doesn't need to be added to the model
        // (it is owned by the concrete class)
    end
}

///////////// class for overwrite interface method for name
class SpecializedEngineInterface_Tool inherits uml2::transformations::Interface_Tool
{
    /** overwrite the original method */
    method nameForInterface(name : kermeta::standard::String) : kermeta::standard::String is do
        result := "IEngine"
    end
}
```

The cloned and transformed model has been saved under *improved_vehicules.uml2 and can be used* as it is or being draft by a graphical tool like TopCased.

3.3 The resulting model

The resulting model of our transformer program looks like above.



It comply with our goals:

- The result model includes all the UML elements which were in the original model, with all their characteristics.
- The GasEngine class has now an interface and a accessor to its attribute.
- The interface operations are abstract when the GasEngine operations are concrete implementations (may be with an empty body for some of them).
- The ElectricEngine inherits from this interface.
- The Car now is now composed of a GasEngine (the corresponding association exists in the model even it has not be represented by TopCased).

Any UML2 model can be transformed like this.

You can visit it and apply conditional transformations on the fly to each kind of UML2 element. Look at the Valooder chapter in the Umlaut-Architecture-Guide to have a idea of how to do.

4 Prototyping with UML

The software industry faces many challenges.

One of them is the difficult understanding of its customers precise needs. A way to solve or reduce uncertainty and waste of time and money is to present those customers a preliminary version of the planned system as early as possible in the development process.

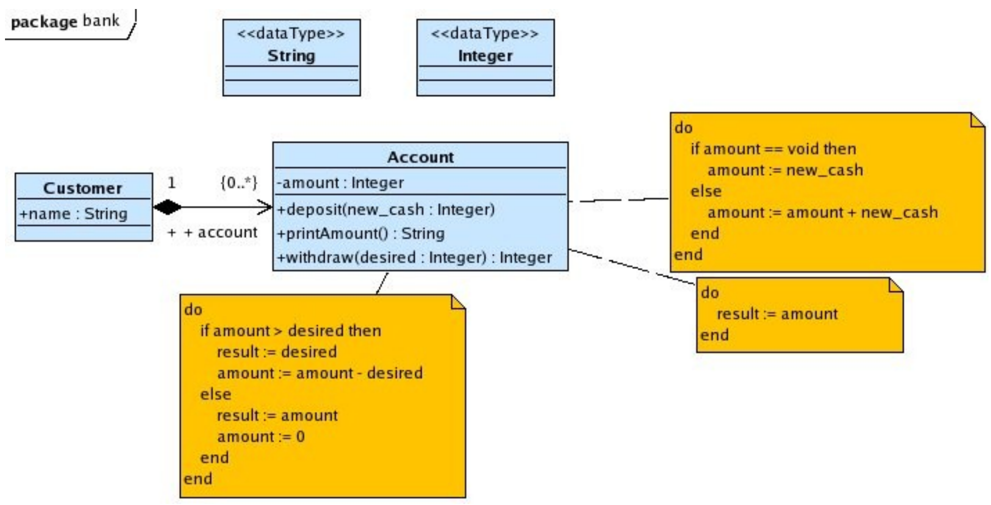
For software engineers, the access to their analysis model under a operative form in order to permit the users to manipulate a simplistic implementation of the main features.

4.1 A simple example

As an illustration of such a prototyping use of UMLAUT tools suite, we present a (very restricted) sample of an accounting model and its use in an even simpler prototype.

An analysis model can be used “as it is” which means the prototype could link to the all primitives the model defines (classes, operations, attributes). Or you can add some behavior to this model by giving a operative body to class operations: the code must be written in Kermeta language syntax and embedded in UML comment attached to the corresponding operation.

Our example presents such a behaviored model.



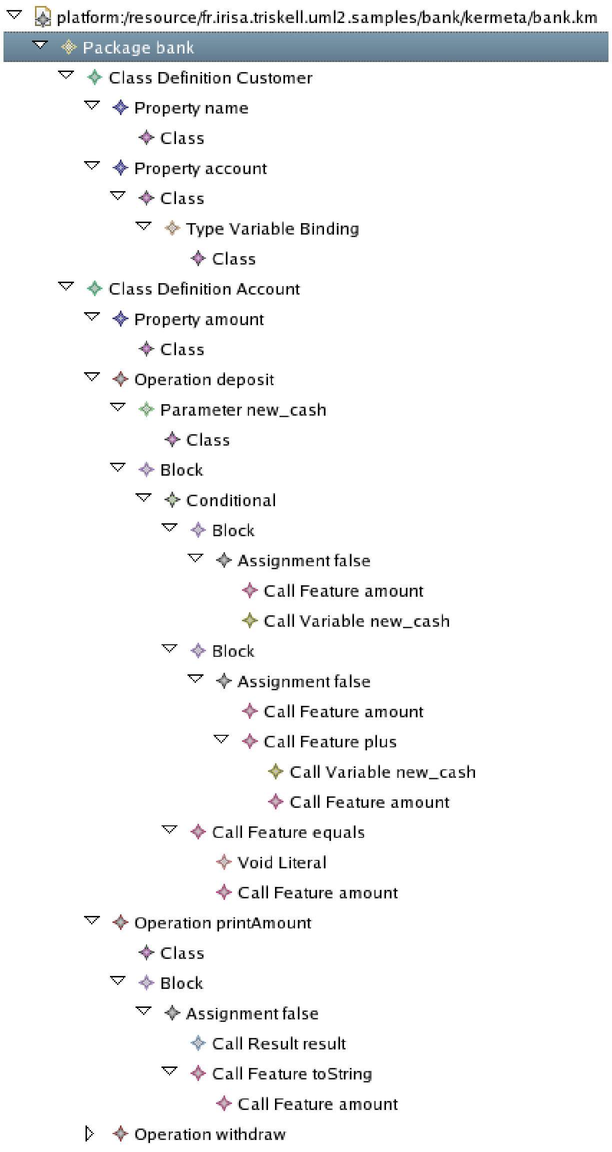
```
do
  if amount == void then
    amount := new_cash
  else
    amount := amount + new_cash
  end
end
```

```
do
  result := amount
end
```

```
do
  if amount > desired then
    result := desired
    amount := amount - desired
  else
    result := amount
    amount := 0
  end
end
```

4.2 Executable version of the model

Applying the *uml2km* tool on our UML model (`bank.uml2`), we get a Kermeta executable model in a XMI form: `bank.km`. You can have a look in it through the Ecore "reflexive editor", as shown above.



```
platform:/resource/fr.irisa.triskell.uml2.samples/bank/kermeta/bank.km
  Package bank
    Class Definition Customer
      Property name
        Class
      Property account
        Class
          Type Variable Binding
            Class
    Class Definition Account
      Property amount
        Class
      Operation deposit
        Parameter new_cash
          Class
        Block
          Conditional
            Block
              Assignment false
                Call Feature amount
                Call Variable new_cash
            Block
              Assignment false
                Call Feature amount
                Call Feature plus
                  Call Variable new_cash
                  Call Feature amount
            Call Feature equals
              Void Literal
              Call Feature amount
      Operation printAmount
        Class
        Block
          Assignment false
            Call Result result
            Call Feature toString
              Call Feature amount
      Operation withdraw
```

This Kermeta model can be transformed in a human readable text version: (`bank.kmt`):

```
/*
 * fr.irisa.triskell.uml2.samples/src/kermeta/bank.kmt
 *
 * a human readable version of the file generated by "uml2km"
 * for documentation of the "bank" example.
 */

package bank;

require kermeta

class Customer
{
        reference name : kermeta::standard::String
        reference account : kermeta::standard::OrderedSet<Account>[0..*]
}

class Account
{
        reference amount : kermeta::standard::Integer

        operation deposit(new_cash : kermeta::standard::Integer) is do
                if amount == void then
                        amount := new_cash
                else
                        amount := amount + new_cash
                end
        end

        operation printAmount() : kermeta::standard::String is do
                result := amount.toString
        end

        operation withdraw(desired : kermeta::standard::Integer) : kermeta::standard::Integer is do
                if amount > desired then
                        result := desired
                        amount := amount - desired
                else
                        result := amount
                        amount := 0
                end
        end
}
```

4.3 Use in a prototype

After getting the Kermeta version of your UML model, you can link any Kermeta program to it, giving you access to all the features and types defined in it.

As an example, we write a drastically simple bank back-office application which manipulates the elements of our Bank model: `prototype.kmt`.

```
@mainClass "bank::Main"
@mainOperation "main"

package bank;

require kermeta
require "bank.km"        // link to the executable model
require "utilities.kmt"

class Main
{
        reference customers : kermeta::standard::OrderedSet<bank::Customer>

    operation main() : Void is do
        // create our "Bank" :-)
               customers := kermeta::standard::OrderedSet<bank::Customer>.new

        // launche the interface
        var ui : UserInterface init UserInterface.new
        ui.initialize(customers)

                stdio.writeln("\n === Welcome to the 'BankOfMine' ===\n")

                // main loop
        var exit : kermeta::standard::Boolean
        from exit := false
                until exit
        loop
            exit := ui.exec()
        end

        // exiting
        stdio.writeln("\n === Thank you for using 'BankOfMine' services ===\n")
    end
}

class Bank
{
        operation initialize() is do
        end
}

class UserInterface
{
        reference customers : oset bank::Customer[1..*]
        reference utility : utilities::StringConverter

        operation initialize(new_bank : kermeta::standard::OrderedSet<bank::Customer>) is do
                customers := new_bank
                utility := utilities::StringConverter.new
        end

        operation exec() : kermeta::standard::Boolean is do
                printMenu
                var command : kermeta::standard::String init stdio.read("   what is your choice ? : ")
                if command == "A" then
                        self.addAccount
                        result := false
                else if command == "C" then
                        self.addCustomer
                        result := false
                else if command == "D" then
                        self.deposit
                        result := false
                else if command == "P" then
                        self.print
                        result := false
                else if command == "W" then
                        self.withdraw
                        result := false
                else if command == "Q" then
                        result := true
                else
                        stdio.writeln("\n >>>> You enter a wrong code <<<<<")
                        result := false
                end end end end end end
        end

        // interfacing operations
        operation printMenu() is do
                stdio.writeln("\nAvailable operations:")
                stdio.writeln("  A - Add a new account")
                stdio.writeln("  C - Add a new customer")
                stdio.writeln("  D - Make a deposit")
                stdio.writeln("  P - Print the current amount of an account")
                stdio.writeln("  W - Make a withdrawal")
                stdio.writeln("  Q - quit the 'BankOfMine'")
        end
        operation choiceCustomer() : bank::Customer is do
                var num : kermeta::standard::Integer init 0
                customers.each{ c |
                        stdio.writeln("  "+c.name+" - nÂ° "+num.toString)
                        num := num + 1
                }
                num := utility.string2Integer(stdio.read("    give the nÂ° of the customer : "))
                result := customers.elementAt(num)
        end
        operation choiceAccount() : bank::Account is do
                var customer : bank::Customer init choiceCustomer()
                var num : kermeta::standard::Integer init 0
                customer.account.each{ acc |
                        stdio.writeln("  "+customer.name+" account nÂ° "+num.toString)
                        num := num + 1
                }
                num := utility.string2Integer(stdio.read("    give the nÂ° of the account : "))
                result := customer.account.elementAt(num)
        end
        operation getAmount() : kermeta::standard::Integer is do
                result := utility.string2Integer(stdio.read("    give the amount to be processed : "))
        end

        // acting operations
        operation addAccount() is do
                var customer : bank::Customer init choiceCustomer
                if customer.account == void then
                        customer.account := kermeta::standard::OrderedSet<Account>.new
                end
                var account : bank::Account init bank::Account.new
                account.amount := 0
                customer.account.add(account)
        end
        operation addCustomer() is do
                var name : kermeta::standard::String
                name := stdio.read("    enter the customer name : ")
                var newCustomer : bank::Customer init bank::Customer.new
                newCustomer.name := name
                customers.add(newCustomer)
        end
        operation deposit() is do
                var account : bank::Account init choiceAccount()
                account.deposit(getAmount)
        end
        operation print() is do
                var account : bank::Account init choiceAccount()
                stdio.writeln("      amount = "+account.printAmount)
        end
        operation withdraw() is do
                var account : bank::Account init choiceAccount()
                stdio.writeln("      you got "+account.withdraw(getAmount).toString+" of money!")
        end
}
```

This application can be used under the advices of the project director (of the customer company) and/or some end-users. An trace of such a use could be like the sample above.

4.3.1 Launch

Run the `prototype.kmt` file as a Kermeta application.

You get:

```
 === Welcome to the 'BankOfMine' ===


Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
   what is your choice ? :
```

4.3.2 A new account

Add a customer named “Henry” to the 'BankOfMine':

```
 === Welcome to the 'BankOfMine' ===


Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
   what is your choice ? :
   what is your choice ? : C

    enter the customer name : Henry

Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
```

```
   what is your choice ? :
```

Add a account to him:

```
   what is your choice ? : A
  Henry - n° 0
    give the n° of the customer : 0

Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
   what is your choice ? :
```

4.3.3 Use the account

Make a deposit:

```
   what is your choice ? : D
  Henry - n° 0
    give the n° of the customer : 0
  Henry account n° 0
    give the n° of the account : 0
    give the amount to be processed : 50

Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
   what is your choice ? :
```

So Henry now has 50 of money on his first account.

We can print the account amount to check that:

```
   what is your choice ? : P
  Henry - n° 0
    give the n° of the customer : 0
  Henry account n° 0
    give the n° of the account : 0
      amount = 50

Available operations:
  A - Add a new account
  C - Add a new customer
  D - Make a deposit
  P - Print the current amount of an account
  W - Make a withdrawal
  Q - quit the 'BankOfMine'
   what is your choice ? :
```

The printing confirms this fact.

We may try to withdraw some money from this account:

```
   what is your choice ? : W
  Henry - n° 0
    give the n° of the customer : 0
  Henry account n° 0
    give the n° of the account : 0
    give the amount to be processed : 30
      you got 30 of money!

Available operations:
```

```
A - Add a new account
C - Add a new customer
D - Make a deposit
P - Print the current amount of an account
W - Make a withdrawal
Q - quit the 'BankOfMine'
 what is your choice ? : P
Henry - n° 0
  give the n° of the customer : 0
Henry account n° 0
  give the n° of the account : 0
    amount = 20
```

So the withdrawal works pretty good.

Then an end-user can ask us "*What if the withdrawal amount is higher than the amount of the account?*". It is a typical situation prototyping is useful for.

Try a withdrawal of 60, for example, and let the end-user have a look on how the system will respond to such a demand:

```
 what is your choice ? : W
Henry - n° 0
  give the n° of the customer : 0
Henry account n° 0
  give the n° of the account : 0
  give the amount to be processed : 60
    you got 20 of money!
```

So the withdrawal operation delivers as much of the desired money as the account has in amount. It is our choice from the analysis of the system we have done. May the customer of that future system doesn't agree with that choice. That is the matter of prototypes to permit engineers and project recipient to see the potential differences in approach of the problem and to discuss about those differences in order to eliminate mistakes.

4.4 The future of UML prototyping

Actually, the model under the scope must be translate into Kermeta code in order to be used by a program.

The prototyping will be more efficient and easier if the prototype application could "require" directly this model like any Ecore [meta]model.

Appendices