SimWeb

IST-2001-34651

Exploring Innovative eBusiness Models using Multi Agent Simulation

Document Name:	**MASDT Design Specification**
Document Date:	*Febraury 4th, 2004*
Document Owner:	*SimWeb Consortium*
Document Author:	*Jesús Delgado, CIMNE*
	Maite López-Sánchez, iSOCO
	Noyda Matos, iSOCO
	Xavier Noria, iSOCO
Contributions:	*Nigel Gilbert, UNIS*
	Stephan Schuster, UNIS
Deliverable Number:	*D34.2 MASDT Design Specification*
Work Package:	*WP34*
Deliverable Type:	*Technical Document*
Version	*1*

Abstract

This document describes the design specification for Simma, the Multi Agent Simulation Development Tool (MASDT) we plan to implement. In particular, this document corresponds to deliverable "D34.2 MASDT Design Specification" .

Keyword List: SimWeb. Technical report. Participatory design. MASDT. Requirements. Validation Cases.

COPYRIGHT AND CONFIDENTIALITY NOTICE

Index

1 Introduction

This document specifies the design of Simma, the multi-agent development tool we are building at SimWeb.

This design is based on the Simma requirement specification, which is documented in *Simma MASDT Specifications*, available on-line in the members area of the SimWeb project page.

2 Design Principles

The design of Simma has been guided by these general principles:

- ***Easy of use for end-users.*** Simma should be a useful tool for end-users. Simulations should be easy to run, study, and interact with.

- ***Easy of use for model-builders.*** Model-builders should get common problems solved by the tool in an easy way, such as automatic looping, visualization of model states, etc. Software contracts should be as non-intrusive as possible, and preferably avoided, to let model-builders focus in the logic of their models.

- ***Portability of the application.*** Simma has to be portable among principal operating systems. This imposes constraints on eligible technologies.

- ***Portability of models.*** Models developed with Simma need to be portable among different installations, no matter the platform. This influences the way we represent models. Care has to be taken with platform specific issues, as conventions about new lines in text files, file names, etc.

3 Technology

Simma will be written in Jython[1]. Jython is an interpreter of Python implemented in Java which provides an excellent integration between Java and Python classes. Jython combines the ease of Python scripting with the ubiquity and power of Java libraries and the Java Virtual Machine.

In this document we shall use Python to refer to the language in which Simma is going to be implemented. Jython will be just used in contexts where Java and Python are intermixed.

[1] http://www.jython.org

Choosing Jython has positive implications for us, Simma developers:

- Development using a scripting language is typically faster, the resulting code is more compact and implementations usually adapt better to changes in requirements. Python is an object-oriented programming language in addition, hence standard object-oriented techniques are available to build a modular, clean application.

- From Jython we are allowed to use any pure Python or Java libraries we need. Not only that, we offer that range of choices to model-builders consequently. There are lots of freely available libraries to solve lots of problems.

- Python is a dynamic language, we have at our disposal meta-programming techniques that may result in automatisms in benefit of model-builders.

Python is a clean general-purpose programming language with a core easy to grasp. To develop models with Simma model-builders will need to program, but the knowledge of professional programmers shall not be required to write models up to certain complexity. At the same time, however, Jython offers to experienced developers a full-featured programming environment.

We have designed this application having in mind model-builders with different degrees of programming experience. Our aim was to find a good balance, trying to offer a way to make easy things easy for beginners, but being careful about the introduction of unnecessary limitations for developers.

The graphical user interface will be implemented using Swing from Jython. This seemed a good choice: The Java Swing library offers a professional look and feel, and fulfils the requirement about the portability of the application.

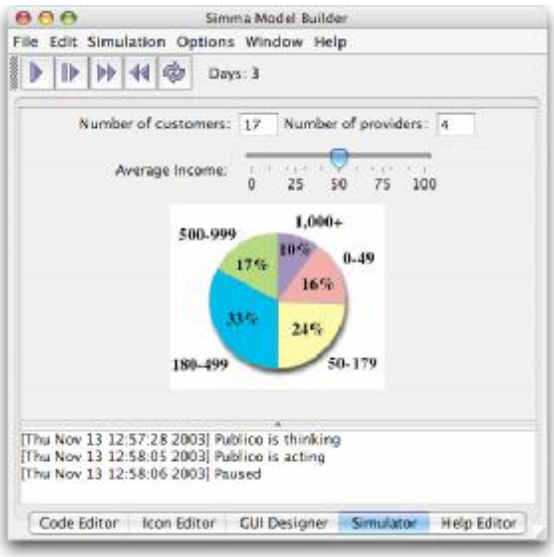
Technically it would be possible to write Simma in Java and work with models written in Jython. Developing Simma in the same language models will be written in, however, avoids an unnecessary communication layer with its implied frictions. Working with Jython we get in addition the benefits we mentioned earlier.

4 Main Modules

4.1.1 Graphical User Interface

The graphical user interface of Simma, which in our jargon does *not* include the graphical interface of simulations, was designed and sketched in the validation cases for Simma included in the document *Simma MASDT Specifications*, available on-line in the members area of the SimWeb project page.

The components implementing this interface come from a more or less direct mapping from the appearance of the application. Menus, buttons, tabbed panes, ..., we will not get into much detail here. It is enough to recognize this set of classes as a distinguished subsystem of the application.:

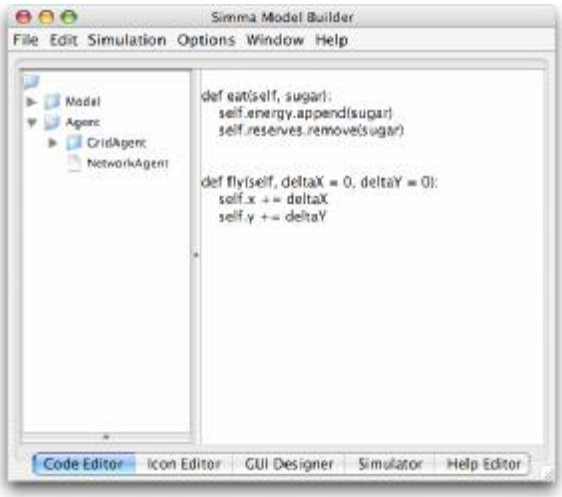


There are a few changes with respect to the interface however:

- The console with output in the model-builder's simulator will be a free-floating window instead of some embedded pane, which will most likely be better suited for expected amount of output.

- It turned out that the combo to group ticks didn't help as much as we initially thought, and made the resulting interface unnecessarily complicated. Model-builders will be able to include widgets which display data for groups of ticks anyway.

- In the design meetings we realized the code editor pane we initially planned might need a rethink though. The code editor pane was initially devised as this mock up suggests:



In the upper left corner there is a class tree, representing an inheritance tree and serving as interface to create subclasses. The main zone was supposed to be reserved just for methods, a partial

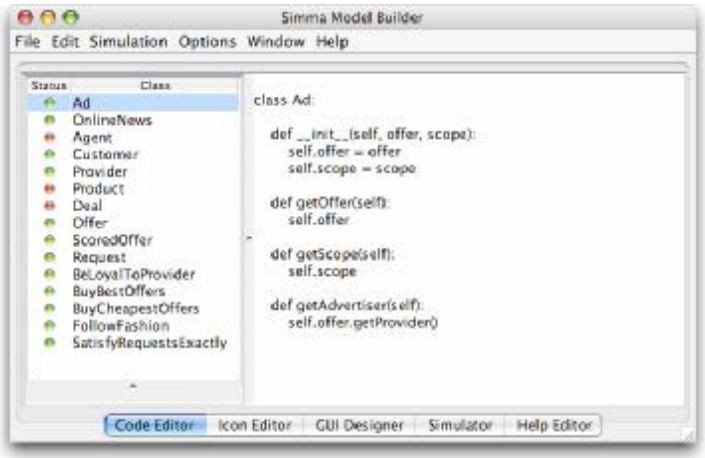
implementation of the class we would munge somehow to get a valid class definition. We founded the following main drawbacks to this interface:

- We think managing inheritance just in the class tree does not feel natural for children of classes outside Simma, such as those in an external library. Those classes would inherit from some others not seen in the class tree.

- There is no suitable place for import statements, a must for Python programming. Some new zone or dialog would need to be added.

- We expect models to be rather flat in that regard indeed. If that was true, the tree wouldn't provide too insight, that space could be better leveraged.

There are a few solutions to that. Thinking in the model-builder we agreed the best choice would be in our opinion to simplify the interface and let the model-builder enter code following standard Python layout in the text area to the right, and let that left part just show the list of user-defined classes and some mark that informs the model-builder about syntax correctness:



If the model-builder selects a class whose code is not well-formed, Simma will indicate the errors somehow in the editor pane, maybe in a gutter.

The true abstraction behind those "classes" is what in Python is known as "modules", which is more or less a set of statements in one file which can contain more than one class definition. The implementation of the tool shall not assume one class definition per item, but the documentation will most likely suggest somehow that simplification.

Packages, in the Python sense, shall not be supported, classes of the same model will share a single namespace and model builders will be able to refer to any class in the model definition from any other one without the need of explicit imports.

The *jEdit Syntax Package*[2] will be used as embedded editor. The icon and help editor would be embedded applications as well, but they have yet to be chosen. Their communication with our classes will depend on their interface.

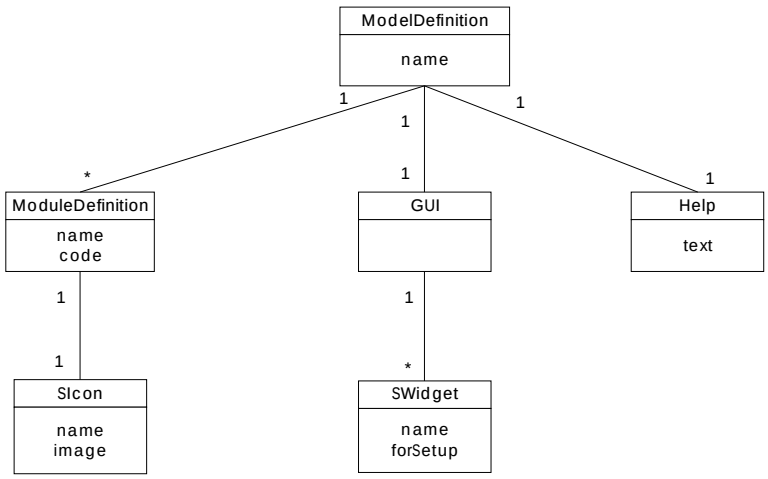
4.1.2 Model Definition

From our perspective model definitions are an aggregation of components defined by model-builders to build their models. That is what we need to represent in our software.

From the model-builder's view a model is essentially comprised of:

- A set of class definitions.
- A GUI to visualize simulations.
- A help document.
- Software dependencies.

We shall represent them in a rather direct manner, as the following class diagram depicts:



Most probably some support for external resources will need to be provided as well, but we have not get into details about this yet. We believe the architecture we have defined will allow us to just plug in that support.

4.1.2.1 ModelDefinition Class

The ModelDefinition class acts as a container, and is the root object we plan to serialize (see Persistence below).

[2] http://syntax.jedit.org/

The libraries bundled with Simma will be available out of the box in the code, but model-builders might want to use third party software. The GUI offers a way to associate libraries to models, and those dependencies will be somehow registered in the model definition class, though the very library files will not.

It will be Simma's responsibility to make the registered libraries available at runtime as long as they are properly installed and configured when models are executed.

4.1.2.2 ModuleDefinition Class

As its name suggest, the class ModuleDefinition encapsulates what a module definition consists of in Simma, which is basically a string with source code, and an associated icon.

Simma will provide some pre-made classes. In particular, in a model definition there must be a child of the class SModel, the one Simma will use to perform model simulations. That's the main bridge between the Simma simulation engine and the model. Simma will try to figure out by class tree inspection which is the one among the ones in the model definition.

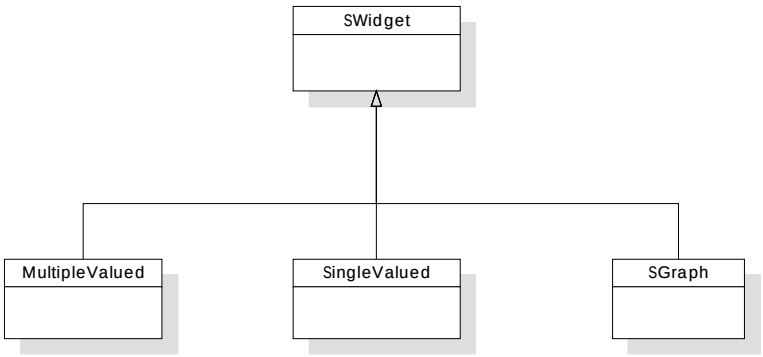
Simma's simulation engine follows discrete steps. Each possible state of a model's life triggers especial methods of the model class the model builder will implement if something needs to be done: onCreation, onSetDefaults, onStart, onEachStep, onPause, onEnd, onRestart.

4.1.2.3 Classes For Widgets

Simma will offer three kinds of widgets to model-builders:

- ***Swing-derived widgets.***
    - ***Single valued widgets.*** These are widgets to read/write single values. For that to make sense the single values need to be kind of global to the model, condition we have relaxed to setters/getters of singletons.
    - ***Multiple valued widgets.*** These are widgets to display collections. For instance, a grid might display the current set of instances of providers and customers.
- ***Graphs.*** These are widgets which offer standard statistical-related graphs.

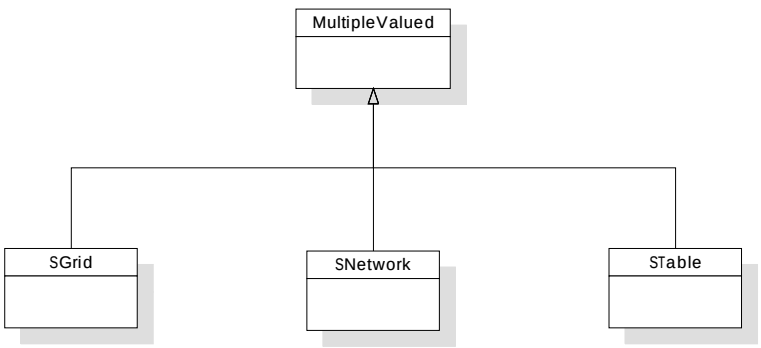
They all will inherit from a base SWidget class:



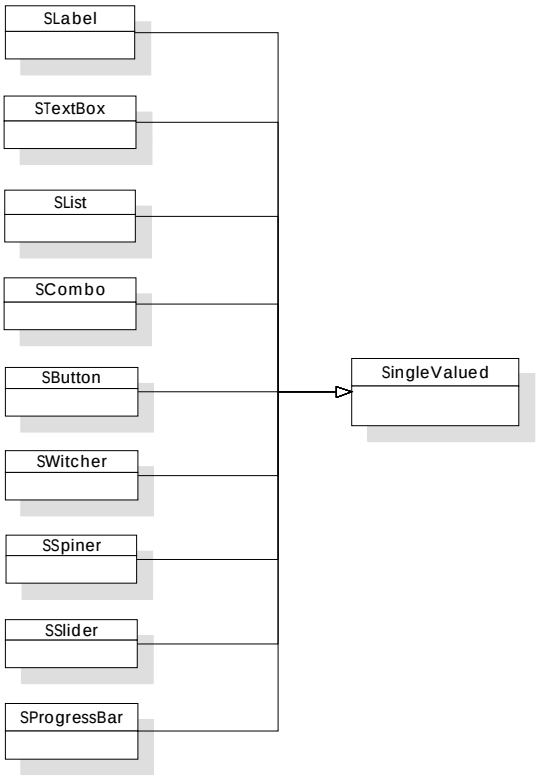
Each widget will have a name that the model-builder will use to refer to them and a flag that says whether it is a widget of setup or runtime. Some globally visible GUI manager will provide access to the defined widgets from code.

4.1.2.4 Swing-derived widgets

Although this is not included in the following diagrams, each custom widget class will in turn be somehow related to its corresponding Swing class, either by inheritance or by composition. Thus, for example, SLabel could inherit from JLabel, and SingleValued.

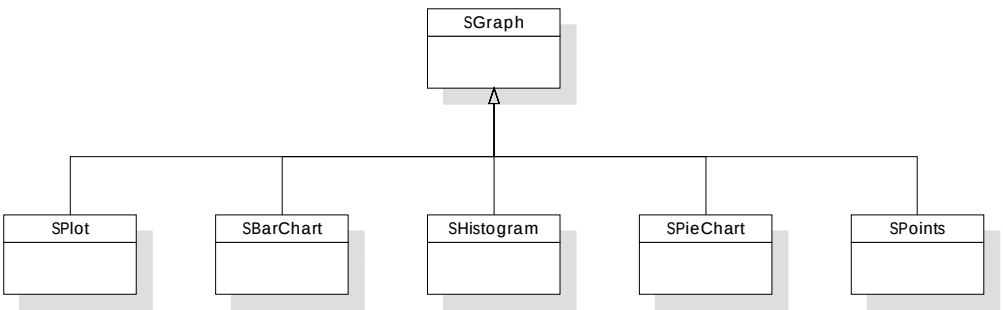


Leveraging the dynamic nature of Python we shall offer a uniform interface to read/write from/to these components, hiding the actual method calls.



4.1.2.4.1 Graphs

The ultimate graphical widgets will depend somehow on the graphical library we choose. Any library provides standard displays, though, so we shall certainly offer them:



4.1.3 Simulation

We call *simulation* to a model execution. Simulations can happen either in the simulator pane of the model-builder view of the application, or in the main pane in the end-user view. The only difference being that in model-builder mode the standard output will be displayed in a text area.To

avoid code duplication, we designed GUI-related stuff with the goal of being able to use the same class for both usages.

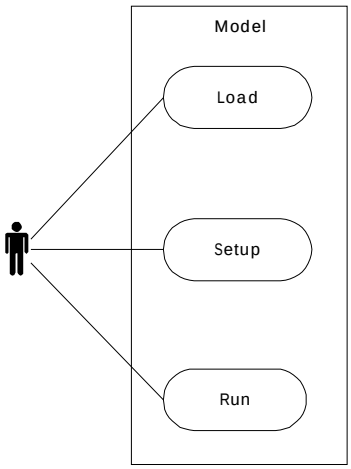
A few global objects are responsible of important services in Simma:

- ***PersistenceManager.*** A persistence manager will be responsible to store and retrieve models and simulation movies from disk.

- ***ConsistencyChecker.*** This class knows how to check up to a certain point the formal consistency of a model. Verifies there is a distinguished class which is the model, checks all the classes and methods mentioned in the GUI definition do exist[3], and so on

- ***Director.*** The director is responsible of the step count, is aware of the current state of the simulation, and manages automatic method calling.

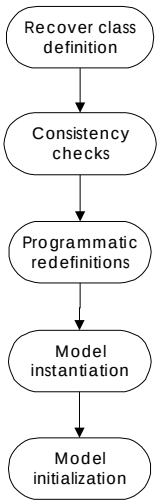
Once global services have been set up a model to be executed passes through three main stages:



4.1.3.1 Model Loading

When Simma first loads a model for execution, either through the persistence layer in end-user model or from memory in model-builder mode, several steps are performed:

[3] This step is needed because widgets and classes/methods are decoupled in the definition phase by design.

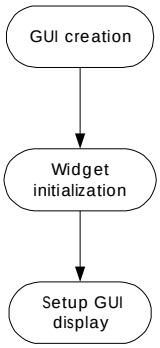


These are the descriptions for each one:

- ***Recover class definitions.*** The source code of the classes defined by the model-builder is read and dynamically compiled.
- ***Consistency checks.*** The consistency checker ensures everything looks right.
- ***Programmatic redefinitions.*** Original class definitions may eventually be modified programmatically at this point to add some magic behind the scenes. This step does not alter the original source code.
- ***Model instantiation.*** The constructor of the distinguished model class is called.
- ***Model initialization.*** The method that initializes a model before the simulation properly starts is invoked.

4.1.3.2 Model Setup

After the model is loaded, Simma shows a Setup GUI where the user could change the initial values of some attributes. To this end the application performs the following steps:



These are the tasks performed in each one:

- ***GUI creation.*** Setup widgets are created.
- ***Widget initialization.*** Models have a chance to initialize setup widgets with suitable default values.
- ***Setup GUI display.*** Make setup widgets visible to let the user interact with them.

4.1.3.3 Run model

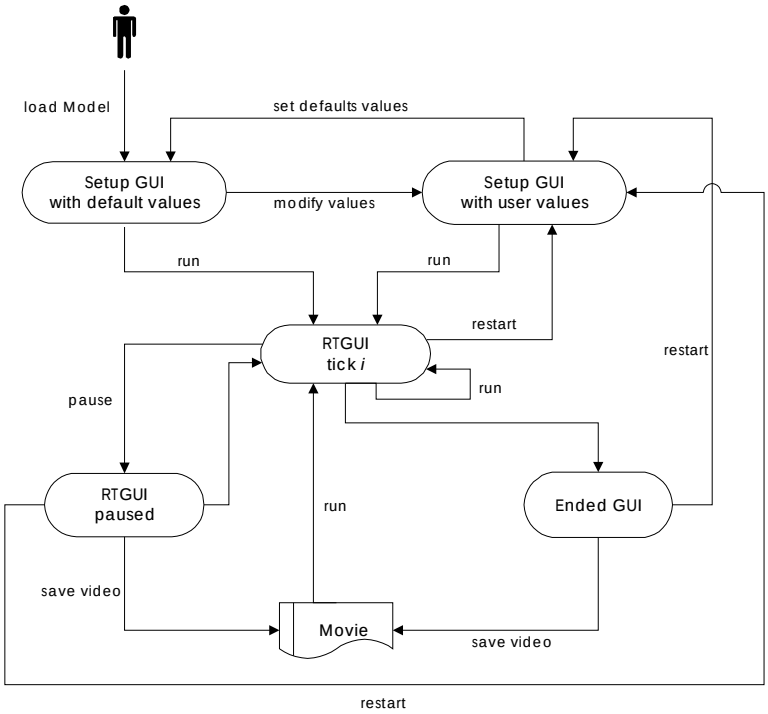
At this point the model is initialized and it is ready to be executed by the user. Simma performs the following steps in that case in a loop that may end in several ways:

- Executes the method onEachStep of the model instance.
- Updates the RTGUI (Run Time GUI).

4.1.3.4 Simulation steps

The simulation is the model execution where the user loads the model, change the setup values and run it. The simulation occurs in a period of time that we measure in basic units called steps. In this section we will describe all the options that the user can perform during the simulation process.

The following diagram shows the main flow of a simulation.



First, as we saw before, the model is loaded and Simma shows the Setup GUI with the default values set by the model. At this moment the user can start the simulation with these values or change them.

The simulation process is controlled by the Director. The Director controls the evolution of the model evolution and for each step he calls the onEachStep method of the model.

During the model simulation the user can perform the following actions:

- ***Pause the model.*** This option stops the model simulation until the user restarts it. The execution performed until that step can be saved into a movie at this point.

- ***Restart the model.*** The current simulation is stopped and the user goes back to the Setup GUI that contains the initial values of that particular simulation, which may be different from the model defaults. This allows the user, at any moment, to adjust the model parameters and restart the simulation with some other setup.

- ***Run the model.*** Whether paused or stopped, the simulation goes.

When the model execution ends the user have the following options:

- ***Save the simulation.*** The user will save the results of the simulation process. All the Runtime GUI showed in each tick will be saved as a movie somehow (see Simulation Movie Playing below).

- ***Restart the model.*** The user goes back to the Setup GUI.

4.1.4 Simulation Movie Playing

The classes in this module know how to play a simulation movie.

At this point it is not totally clear how movies will be represented. Either Simma will save simulation movies as real movies, or it will save the necessary data to let some custom engine reproduce the display. In either case the widgets of the simulation interface will be read-only.

4.1.5 Persistence

Since models and simulations need to be portable among different installations of Simma care has to be taken in the form they are stored in disk.

The persistence layer is the unique mean of communication between components that need to be saved and the file system. From a end-user's view a model will be seen as a single file, a model will be self-contained, though external resources as third party libraries shall not be bundled into.

Model storage will be implemented using standard Java serialization mechanisms.

5 Glossary

Model. In our context a *model* is an abstract representation of some dynamic phenomena in the form of a computer program. Simma is an integrated tool to develop and execute multi-agent models.

Model setup. Models normally have some initial parameters. A *model setup* gives particular values to them, after that the model is ready for execution.

Simulation. A *simulation* is the execution of a model with some particular parameters. A simulation can be paused and it usually features some output widgets that display observable data about the state of the model.

Step. Simulations run following a sequence of discrete iterations called *steps*. They normally represent units of time, days for instance.

Simulation Movie. A *simulation movie* is a record of the output of some simulation. A particular simulation can be saved in the disk as a movie, loaded in Simma afterwards, and played.

Model-builder. The *model-builder* develops models for the end-user using Simma. She encodes the model itself in Jython and designs its visual interface with the facilities provided by the tool.

End-user. The *end-user* is the person that uses Simma to play with models written by the model-builder. He uses the tool typically to study some model under different setups, observing its evolution, and comparing different recorded simulations.