METAMILL™ 8.2

USER'S GUIDE

Metamill Software

Edited: 5-Jan-2018

Table of Contents

1. Introduction

What is Metamill?

Metamill™, UML™ CASE Tool

Thank you for choosing Metamill, a professional UML CASE tool. Whatever system or application you need to design, Metamill helps you to model it in UML. When your design is ready enough, you can generate source code in multiple languages from your model. You can import code to create new models from your existing code. Metamill is a standards-based, solid, easy to use tool ready for fast paced software design.

Metamill supports UML 2.4 - the latest version of UML standard. All 14 UML diagrams are supported, including profile diagrams. Model files also support the latest XMI standard, XMI 2.1. XMI being plain text XML it allows open access by external systems. Metamill packages can be managed, i.e. stored in its own XMI file, for example under version management system. Java, C++, ANSI C, C# and VB.NET source code can be imported to model and generated from a model, that is, round-trip engineered. Metamill uses code markers to preserve custom changes made in the source code. RTF and HTML documentation can be generated from models. A scripting language called MetamillScript can be used to manipulate model elements. Metamill is a fast, native Windows binary, written in C++.

Metamill 8 has gone through large overhaul of its internal model system and it is now better and more robust than ever. Version 8.2 introduces improved support for code engineering in C++11. Long awaited support for ADA and Python are also finally there, supporting both code generation and reverse engineering. ADA 2005 is the target level, also some ADA 2012 features have been implemented. Metamill 8.2 now supports Python 3 language and has improved support for C++11.

Metamill Software,
http://www.metamill.com

About This Document

This is Metamill User's Guide. It covers the basic functionality which is also declared in the Metamill Help (it can be invoked by pressing F1 while running Metamill). The application is meant to be as intuitive as possible and this document serves mainly as an introduction to UML. For advanced code engineering details, refer to chapter "How to generate code" and "How to reverse engineer code*. For details how code engineering can be tuned with constraints, refer to chapter "How to define attributes and operations" and "How to define constraints and tagged values". There exist many good books about software engineering and UML, which certainly are helpful in understanding modeling with Metamill and UML in general.

Bug reporting

Metamill is a complex software design application and big efforts are made to keep it bug-free. If despite of this, a bug is still causing gray hairs to you, please report it to support@metamill.com or using on-line support at www.metamill.com/support.html and explain with details how to reproduce it and under which circumstances it happened. Please include also information about your operating system and Metamill version and build number. Bugs are fixed as soon as they are identified, and a proper solution can be done. Improvement ideas and new feature suggestions are also welcomed.

Acknowledgments

Many thanks to all of you who helped with this release and especially for those who helped to identify, and correct issues found in earlier versions.

Latest evaluation version can be downloaded from Metamill website at:

www.metamill.com

2. Installation

Windows

To install Metamill to Windows, run the self-extracting archive mmill82.exe. It will uncompress the archive and starts the setup program automatically.

The setup program will issue a number of prompts. It is recommended to accept the default settings (just press OK, Yes or Next) unless you have reason to override them. You must explicitly agree the user license to install the program.

You also need to set Metamill Base Directory, see below.

If you need to uninstall Metamill, open Control panel, then double click `Add/remove programs', and then double click Metamill in the list.

Linux

To install Metamill to Linux, metamill82_ix86_64.tar.gz to a temporary directory and run the script setup.sh.

E.g.

```
% mkdir mtemp
% cd mtemp
% tar zxf metamill82_ix86_64.tar.gz
% ./setup.sh
```

The setup program will issue a number of prompts. It is recommended to accept the default settings (just press OK, Yes or Next) unless you have reason to override them. You must explicitly agree the user license to install the program.

If you need to uninstall Metamill, cd to the directory under which Metamill install directory is, and run uninstall.sh.

E.g. (if you installed Metamill in your $HOME):

```
% cd ~
% Metamill-8.2/uninstall.sh
```

Metamill Base Directory

When you start Metamill first time it will ask you to set the Metamill Base Directory. It is the directory under which all Metamill system and model files will be stored. (if you installed Metamill to Program Files, you may not have write access which is needed). Select a new directory with read/write access rights. If the directory you gave does not exist, it will be created. You can reset Base Directory later by pressing SHIFT-F7. Details of directories can be seen in Tools – Options.

Activating License

After you have successfully installed Metamill, you can activate it.

Run Metamill and wait few seconds until an "Apply License" dialog appears. Give the user id and license key and click OK to activate your registered version. You receive the user id and license key by e-mail soon after you buy the license (see www.metamill.com).

If you are evaluating Metamill, just click OK leaving the fields empty.

For more information about licensing options and for the latest version, see Metamill website at

www.metamill.com

Thank you for choosing Metamill.

3. What's New

What's New in Release 8.2

1) Improved support for C++11 (and C++14)

See page 11 for a short summary of new features.

Features in Previous Versions

Since release 8.1

1) Bug fixes and GUI improvements
2) Summon inner classes

Since release 8.0

1) Python 3 code generation and reverse engineering
2) Enhancements to GUI and model system
3) Partial diagram export

Since release 7.0

1) ADA 2005 code generation and reverse engineering
2) Support for UML 2.4
3) Refurbishing GUI libraries to modern standards
4) Internal overhaul of model system

Since release 6.2

1) Diagram element size alignment
2) Improvements in large data import
3) Timing diagram scaling problem fixed
4) Task cancel button

Since release 6.1

1) Diagram element alignment
2) Sequence diagram documentation improvement
3) Improvements in .mdl file import

Since release 6.0

1) Support for UML 2.3
2) All 14 UML diagrams supported
3) RTF (Word) document generation
4) Java annotations support
5) Experimental sequence diagram reverse engineering
6) Drag and drop attributes and operations
7) Fast XMI model file loading
8) Improvements in code engineering and GUI

Since release 5.0

1) Support for UML 2.1 and XMI 2.1
2) All 13 UML diagrams supported
3) Timing diagrams and Interaction Overview diagrams added
4) Automatic layout of diagram elements
5) VB.Net round-trip engineering
6) Support for generics in Java, C# and VB.Net
7) Rose .mdl files import/export
8) Browse diagrams by diagram type
9) CPP preprocessor for C/C++ code import

Since release 4.2

1) VB.Net reverse engineering
2) Support for keeping straight lines
3) Custom color relationships
4) C++ code engineering improvements

Since release 4.1

1) Support for workspaces
2) Support for .mdl - file import
3) Detailed hiding of attributes and operations
4) GUI performance improvements
5) Inherited methods in sequence diagrams

New Features in v8.1

Improved support for C++11 means that now you can import, reverse engineer and generate code using C++11 code.

Features in Previous version v8.1 / v8.0 / v7.x

Bugs corrected:
- 1825: Metamill base directory options
- 1825: Relationship moving fix
- 1825: InteractionOverview diagram follow problem
- 1825: Accessibility support for large text
- 1825: Python class documentation fix

Summon Inner Classes. Now it is possible to summon inner classes to a diagram. This is useful in ADA and Python models.

Python code generation and reverse engineering i.e. round-trip engineering is now possible. GUI enhancement means more speed and more robust interaction. Partial diagram export makes it possible to export bitmaps for only selected elements.

ADA 2005 code generation and reverse engineering i.e. round-trip engineering is now possible. ADA object orientation is adapted to UML model so that a package is presented as a class in the model. New types are defined under a package class as classes stereotyped <<type>>.

Support for UML 2.4 means Metamill is supporting the latest UML standard.

Refurbishing GUI libraries to modern standards means using latest GUI libraries and improving the overall user experience, still keeping it intuitive and keeping the designer

in mind. Class dialog has now a tab for inner classes which helps to examine the hierarchy of selected class.

Internal overhaul of model system has been made, lots of code was rewritten to make it simpler and more robust without compromising existing features and compatibillity.

Diagram element size alignment helps you to align diagram elements size. Select two or more elements and right click the element you want other elements to align with and select size alignment. All elements will be adjusted to same size.

Improvements in large data import means more feedback on importing large data sets. Huge data sets may take a long time to process.

Timing diagram scaling problem fixed means each lifeline now can maintain same scaling as long as t-min and t-max are given. This helps you to compare timing events between different lifeline objects.

Task cancel button allows you to cancel potentially time consuming tasks like code import and automatic backup.

UML 2.3 support / All 14 UML diagrams supported means Metamill now supports the latest OMG standard. UML 2.3 upgrade has been made in the model system and the profile system has been largely improved. With profile diagrams you can explicitly describe how to extend metaclasses by stereotypes.

RTF (Word) document generation allows you to generate documentation about your model and diagrams in RTF format. The generated document can be opened with Word or other tools that support RTF.

Java annotations support adds this important Java feature to Metamill. You can add annotations on Java code using tagged value ic_annotation. Reverse engineering also sets up the tagged value.

Experimental sequence diagram reverse engineering reads operation source code in C++ and Java and builds sequence diagrams. These diagrams help documenting and understanding existing code. More details in "How to Reverse Engineer Sequence Diagrams"

Fast XMI model file loading means at least 10-fold performance improvement in model loading. Mainly affects the Windows version.

Drag and drop attributes and operations feature helps you to manipulate class members. Now you can see members in model tree under classes and drag them to copy them under another class.

Improvements in code engineering and GUI means some existing bugs found since previous release have been fixed. GUI performance and usability has been improved. E.g. by right-clicking a container element you can choose to lock its inner elements so that automatic nesting gets disabled.

Other Important Features

Support for UML 2.1 and XMI 2.1 means major upgrade in UML support. Metamill now has Timing diagrams and Interaction Overview diagrams, thus it now supports all 13 UML 2.1 diagrams. XMI 2.1 means new XMI file format, upgraded from very old XMI 1.2

format. Metamill automatically transforms old models into this new format. Loading and saving is also much faster than in the previous version.

Automatic layout of diagrams allows you to layout all diagram elements in a diagram. It uses a layout algorithm developed by Metamill Software to visualize an inheritance tree and to place elements evenly with the inheritance trees. This is especially needed after importing existing code. All this in just push of a button (F12).

Support for generics in Java, C# and VB.Net allows you to reverse engineer and generate generics in above languages, just like templates which are supported in C++. Now you can also generate VB.Net code.

Rose .mdl files import/export means that you can import Rational Rose model files to Metamill and also export them. In other words, it means that Rational Rose can read exported Metamill models. This is valuable in prototyping where you design your initial models in Metamill, but must use Rose in your official documentation.

Browse diagram by diagram type lets you browse all diagrams in the model and easily locate them. You can filter the diagrams by diagram type.

CPP preprocessor for C/C++ code import enhances the macro system used in the earlier Metamill versions. Now you can declare macros in models and Metamill also preprocesses macros in your existing code.

VB.Net reverse engineering allows you to transform your VB.Net source code to Metamill UML models. Code generation is now supported too.

Keeping straight lines means that you can force associations and other relationship lines straight, i.e. to have sharp corners. Also, in this mode, you can explicitly set the start and end points on element's side. To make a line straight, right click mouse on the line and choose "Keep straight".

Custom color can be set to relationships via relationship properties' details – tab.

C++ code engineering improvements include improved throw statements handling and allowing initial values in C++ member variables. Throw statements are now stored in tagged value "ic_throw" instead of "ic_initm". To allow initial values in C++ variables, e.g. int LCODE = 1; add constraint "ce_allow_attr_init" to model root element.

Workspaces allow you to save the current set of open diagrams and also to remember which items were expanded in the model tree. Just press Ctrl+W to save the workspace. Metamill also remembers the model which you were editing and opens it automatically next time the application is launched. To not use workspaces, create empty new model and save the workspace without saving the new model.

Importing .MDL - files helps you to use model files saved by other UML tools. Just choose "Tools -> Import model -> Import .mdl" to import an .mdl file. Also diagrams are imported as much as possible. Now you can export .mdl files too.

Detailed hiding of attributes and operations means that you can use right mouse on a class to hide attributes, operations, or non-public members. You can also hide individual attributes or operations using vi_show taggedvalue with yes or no value. E.g. to not show an attribute, create taggedvalue "vi_show=no". To show it always, create taggedvalue "vi_show=yes". To tune generation of getters and setters refer to "How to Set Constraints and Tagged Values".

UML2.0 support is a major reorganization of the model system. Metamill now supports as closely as possible the latest UML standard. Activity and state diagrams have been

separated as the UML2 says. Sequence diagrams have been improved heavily. Old version supported old XMI format 1.2 only. Now Metamill supports UML 2.1 and XMI 2.1.

Integral method implementation code means that you can store all source code in the model. Use operation properties' Code - tab to write implementation details. Do not modify source code directly. Use this feature with caution, because the model files can easily become very large. By default, this option is not selected. Open the model properties to select it.

Visually nested elements are no more just visual diagram elements. The nesting is now reflected in the model as well. Drag and drop an element on top of another, and it becomes a nested element. Hold CTRL and drag the nested element outside the owner to put it back outside.

Generation of Get and Set methods automatically generate get and set methods for attributes.

ANSI C round-trip engineering is now possible, but as C is not an object oriented language, some features are different from other languages supported. E.g. in C reverse engineering, each file becomes a class and all global functions or variables become its members, when code is generated, all members become global variables and functions again.

Relaxed types mean that type ids are no more strictly bound to existing types. Now you can declare as complex types as necessary without the pain trying to support associated type elements. This eases the visual high-level design (where exact types need not yet exist) and also code generation where types can map one-to-one with generated source code types.

Code engineering improvements include support for typedefs, unions and structs, support for namespaces and import/using statements. Declare namespace with "namespace" tagged value and import/using statements with "import" tagged value. Declare plain typedefs as a subclass with stereotype "typedef" and a tagged value "typedefval" which is type contents, a class name becomes declared type's name. To declare "typedef class A { … } A" construct, just add constraint "typedef" for a class. See also Define Attributes and Operations.

Recursive code import means that you can now recursively import source code directories. This helps when importing large projects. See How to Reverse Engineer Code.

Model templates can be defined to be used every time a new model is created. Declare all code language specific data types (e.g. int, double) or any types or classes you need in every model. Model templates should be put into "Base directory for models" defined in "Options - Files" in order to be visible in the template dialog box.

GUI improvements include drag-n-drop from model trees to diagrams, find an element (press Ctrl+F), speed enhancements for diagram redrawing (refresh screen with Ctrl+L). Z-level for elements can be set with right mouse button.

MetamillScript now supports subroutines and more operations are available for element manipulation. See How to Write MetamillScripts.

4. Metamill Quick Start

Quick Start with Metamill

1) Set options if needed (like models base directory). See How to Set Options
2) Create new model. See How to Create a New Model.
3) Create new use cases under UseCaseView, logical design under DesignView and components under ImplView.

Also, have a look at Design guide to get (very) short introduction to software engineering.

Useful Tips

Save Workspace

You can use Ctrl-W to save the workspace, i.e. current model and currently opened diagrams. Next time you start Metamill the workspace is automatically loaded.

Drag-n-drop from model tree

After importing code there are many classes and relationships in the model, but which are not visible in any diagram. You can easily add existing model elements to an open diagram - just drag elements from model tree. You can also create new elements by dragging elements from tool palette.

Panning

You can "pan" diagram with your mouse: just hold shift key and start dragging anywhere in the diagram. The diagram moves with your mouse.

Moving system boundary label

You can move the system boundary label alone: hold control key and drag system boundary label. The label moves and the system boundary stays unmoved.

Showing icon mode in sequence diagram

You can show the icon mode of an object in sequence diagram: select object and edit properties, select stereotype that has icon presentation (interface, boundary, controller etc.) and select icon mode checkbox.

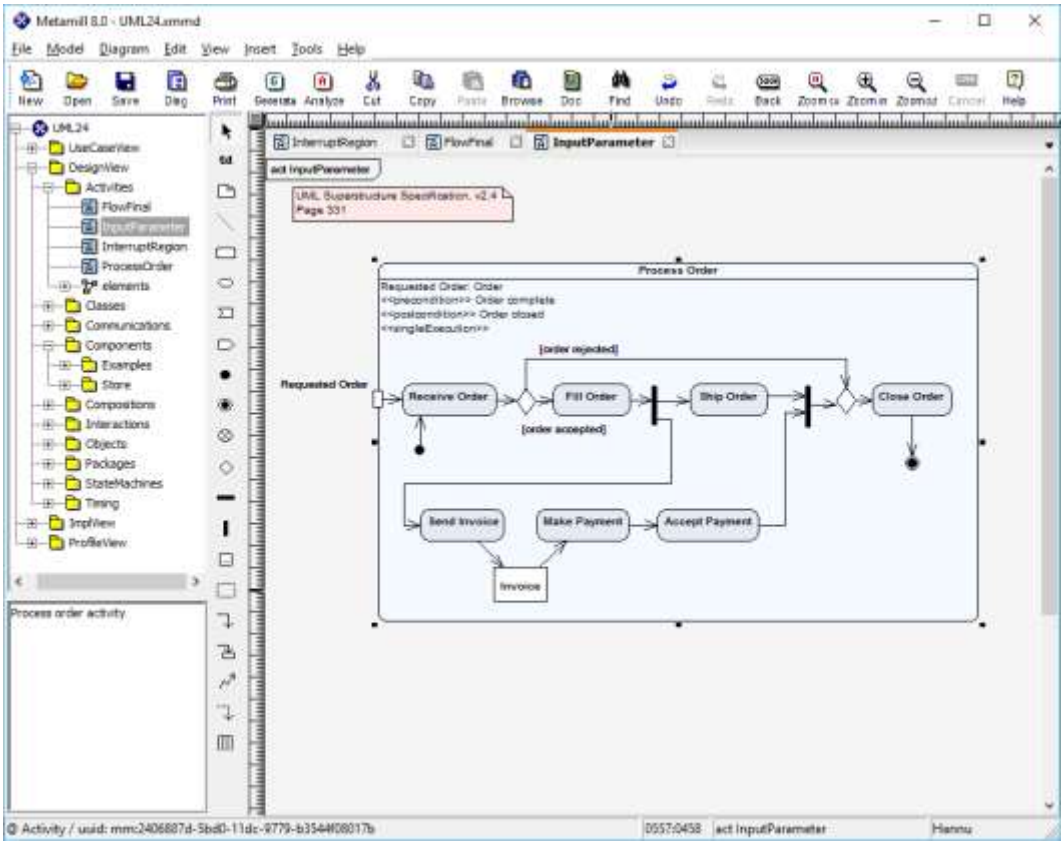
Suppress Metamill text from print header

Add tagged value "prn_header=simple" to model root element. To suppress it from HTML document, add "doc_header=simple".
(registered version only)

5. Metamill Menus

Metamill Main Screen

Below you can see the screenshot of Metamill's main screen. It is intentionally kept as simple as can be, to make using it intuitive.



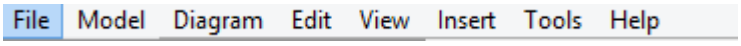
On the left you can see the model tree, which shows all model elements in the current model. Below the model tree there is a documentation window, which shows you the contents of documentation field when you select an element in the diagram. On the right there is a diagram canvas. Between the diagram canvas and the model tree there is a diagram toolbar which is different for each diagram type.

In the field at bottom you see messages triggered by different events, like information about last selected diagram element.

Toolbar

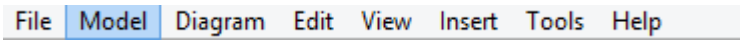
Tool	Key	Explanation
New model		Create a new model
Open model	Ctrl + O	Open an existing model
Save model	Ctrl + S	Save model
New diagram	Ctrl + N	Create new diagram, prompts for diagram type
Print diagram	Ctrl + P	Print diagram
Generate code	Ctrl + G	Generate code for selected classes
Reverse engineer	Ctrl + R	Reverse engineer selected classes
Cut	Ctrl + X	Cut selected elements
Copy	Ctrl + C	Copy selected elements
Paste	Ctrl + V	Paste copied elements to current diagram
Browse diagrams	Ctrl + B	Browse all diagrams filtering by diagram type
Generate document		Generate RTF or HTML document
Find model element	Ctrl + F	Find element from model
Undo diagram command	Ctrl + Z	Undo command performed on diagram
Redo diagram command		Do again undone diagram command
Go back		Go back to caller diagram (interaction use)
Switch zoom	F9	Switch zoom to close or far view
Zoom in	PAD +	Zoom in diagram
Zoom out	PAD -	Zoom out diagram
Cancel		Cancel current operation (code import etc.)
Help	F1	Show Metamill Help

File – menu



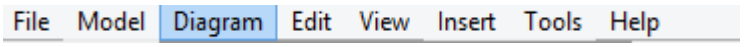
New Model		Close current and create new model
Open Model	Ctrl + O	Open existing model
Close Model		Close current model
Save Model	Ctrl + S	Save model
Save As		Save model with other name
Page Setup		Printer page setup
Print Preview		Show print preview
Print	Ctrl + P	Print diagram
Save Workspace	Ctrl + W	Save workspace
Recent Models		Open a recently opened model
Exit		Exit Metamill

Model – menu



Model Properties	Edit model properties (double click model icon)
Find Element Ctrl + F	Find element in the model by name
Summon Related	Associated Elements: fetch all related elements for selected element Inner Classes: fetch all inner classes

Diagram – menu

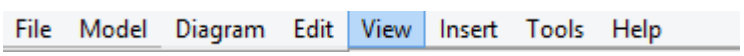


New Diagram Ctrl + N	New diagram under selected item in model tree
Close Diagram Ctrl + K	Close currently open diagram
Discard Diagram	Discard all changes to diagram and close it
Layout Diagram	Layout diagram, layout all or selected items
Browse Diagrams Ctrl + B	Browse diagrams by diagram kind
Export Diagram Ctrl + E	Export current diagram to a bitmap
Export All Diagrams F10	Export all open diagrams to a bitmap
Diagram Properties	Edit diagram properties

Edit – menu

Undo Ctrl + Z	Undo last diagram command
Redo Ctrl + Y	Do again last undone diagram command
Cut Ctrl + X	Cut selected diagram elements
Copy Ctrl + C	Copy selected diagram elements
Paste Ctrl + V	Paste diagram elements
Delete	Delete diagram elements (Ctrl+D = deep delete)
Select All Ctrl + A	Select all diagram elements
Element Properties	Edit diagram element properties

View – menu



Show Model Tree F4	Show or hide the model tree
Previous Diagram F5	Show previous open diagram
Next Diagram F6	Show next open diagram
Switch Zoom F9	Switch zoom to close or far view
Zoom In (+)	Zoom in diagram

Zoom Out (-)	Zoom out diagram
Refresh Ctrl + L	Repaint screen (for vi users)

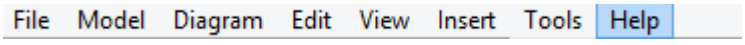
Insert – menu

Text	Insert text element
Note	Insert UML note element
Note Link	Insert note link, start dragging from a note
<elements>	Insert element (list depends on diagram type)
<relationships>	Insert relationship (list depends on diagram type)

Tools – menu

Generate Code Ctrl + G	Generate code from model (select in model tree)
Analyze Code Ctrl + R	Analyze code, i.e. reverse engineer code
Import Code Ctrl + I	Import external code to Metamill
Export Model Export XMI Export .mdl (Rose)	Export model Export XMI (select XMI 2.1, 2.0) Export to Rose model
Import Model Import XMI Import .mdl (Rose)	Import model Export XMI (supports XMI 2.1, 2.0) Imports Rose model
Generate Document	Generate documentation RTF, HTML or TXT
Run MetamillScript	Run MetamillScript script file
Options	Edit global options (alt+T and O)

Help – menu



Help Contents F1	Show Metamill Help
UML Element Reference	Show UML Reference
Apply License	Apply new license
Tip of the Day	Useful tips
About Metamill	Show product information about Metamill

6. Metamill Design Process

About Software Engineering

There is a multitude of different software engineering methods and processes and not one of them emerges to be clearly superior to others. However, the following eight phases can be seen vital to any software project:

1) Capture requirements
2) Design software architecture
3) Design subsystems
4) Write code
5) Unit test
6) Integration test
7) System test
8) Deliver system

The first three or perhaps the first four are interesting from the point of view of analysis and design. Note that the above list is made from point of view of software analyst and designer, not project management (that often concentrate on phases 1, 7 and 8 only, ignoring the rest).

These four phases together form a simplified design process, which is here called Metamill Design Process.

Metamill Design Process

This is not intended to be comprehensive software engineering process; it is here just to give an idea how things should be done. And, sometimes organization's official software engineering process is so heavy, that it is not followed in practice, or no written process exists. In these cases, it is better to use a simple process than not to use a process at all!

Capture Requirements
Design Software Architecture
Design Subsystems
Write Code

Note that the system analyst, software architect, software designer and programmer can be understood as roles. One software engineer may perform all the tasks.

In addition to the artifacts mentioned above, many different text documents can be produced simultaneously. The common separation of design documents can be seen below.

Functional specification:	use cases, sequence diagrams, statecharts
Architecture specification:	package diagrams
Technical specification:	class diagrams, sequence diagrams, statecharts
System Specification:	component diagrams, deployment diagrams

Capture requirements

The first phase of software engineering process is requirements capturing. This is done in close interaction with business analysts and if possible, with future users of the system. A system analyst defines a use case for each behavior the system must perform. The idea is to define what the system must do, not how it should be done.

Artifacts: use case diagrams, sequence diagrams

Design Software Architecture

A software architect analyses the problem domain and identifies different subsystems. For example, a TradeHandler could be a subsystem of Stock Exchange System, and TradeValidator could be a software component used by TradeHandler. Subsystems are defined as packages. At this level we do not talk about physical components, because they are physical implementations of subsystems or packages. It is good to keep that in mind, anyway.

After identifying the packages, the software architect designs interfaces for each subsystem. Software architect may design only main subsystem interfaces and give the rest to the software designer.

Common mistake is to think that no software architecture design is needed after a system architect has selected the high-level software platform and hardware for the new system. In fact, the design of software architecture is the most important phase of software design. Better not forget it!

Artifacts: package diagrams, component diagrams

Design Subsystems

This is where the actual implementation of the system is designed. Each subsystem can be designed separately from the others. A software designer designs the static view of the system by using class diagrams and the dynamic view using sequence diagrams and activity diagrams. Not everything should be put into diagrams, only the essential elements needed for explaining the implementation of the system. Use statechart diagrams to describe life-cycles of important objects. Activity diagrams can be used to describe the flow from activity to activity. Use collaboration diagrams and object diagrams to show relationships between instances of classes.

At this level it has to be decided, whether the subsystem is an active service (a server) or just a plain library. A subsystem may become a component. A component is a physical element, normally an implementation of a subsystem. Note that a component may use other components. Use component diagrams to describe the physical composition of the system; use class diagrams to describe how the system is implemented.

Artifacts: class diagrams, sequence diagrams, statechart diagrams, activity diagrams, component diagrams

Write Code

A programmer writes the code using actual implementation language. Metamill can be used for forward engineering and reverse engineering. Forward engineering means that the code can be generated from diagrams and reverse engineering means that diagrams can be constructed from existing code. Attempts to use reverse engineering cause sometimes more troubles than benefits and may lead to confusing models that are hard to understand. Reverse engineering is at its best in documenting old legacy code. The main purpose of visual modeling tool is to describe the system and make it understandable to other designers, not to use the tool as a programming language.

In some cases it is enough to generate only the initial code and then update the diagrams manually when the code evolves. However, it is recommended to adopt the forward engineering approach, that is, to update the diagrams, generate the code and to keep the code changes inside user blocks.

Artifacts: implementation source code

7. Round-trip Engineering

Forward Engineering

In forward engineering the implementation language source code can be generated from diagrams and the user makes changes to diagrams and then generates the code. Metamill has special user blocks in the generated code where changes can be made without losing them when the code is re-generated.

Reverse Engineering and Round-trip Engineering

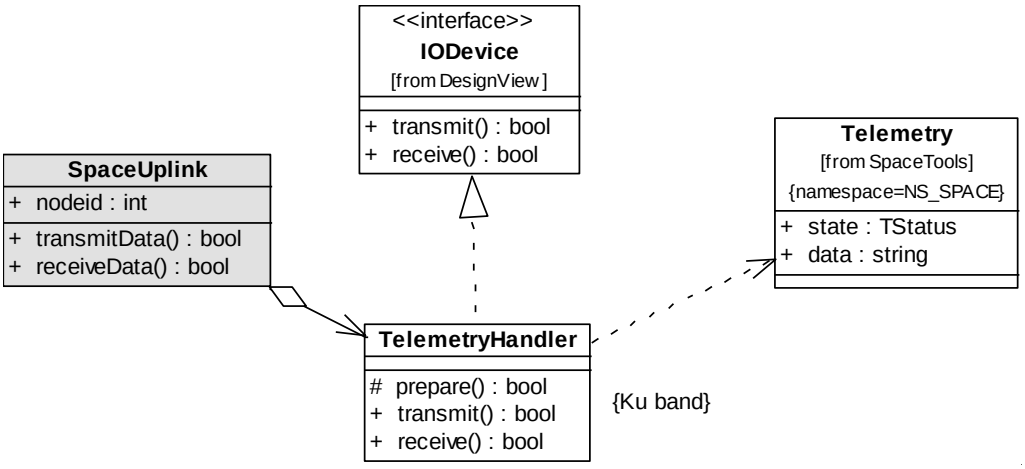
In reverse engineering the diagrams are generated from the source code. The source code is first analyzed and then diagrams are created. The problem with this approach is that it will often produce incomprehensible diagrams containing too much implementation details. In round-trip engineering the code is first generated from diagrams and then analyzed again using reverse engineering. Same problems arise as described above. However, reverse engineering suits very well for documenting old legacy code.

Round-trip Engineering with Metamill

See "How to Generate Code" and "How to Reverse Engineer Code"

8. UML Diagrams

UML Class Diagram



Overview

A class diagram illustrates the static design view of the system. It is one of the most important diagrams used in object oriented modeling. A class diagrams consist of classes and interfaces and relationships between them. The classes in class diagram usually have direct counter parties in implementation language. Of course, you can design classes that are in very high level of abstraction, just explaining their responsibilities. The high level approach is usually used at the beginning of design.

Creating a Class Diagram

To create a new class diagram, choose New diagram from the Diagram menu and then choose Class diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

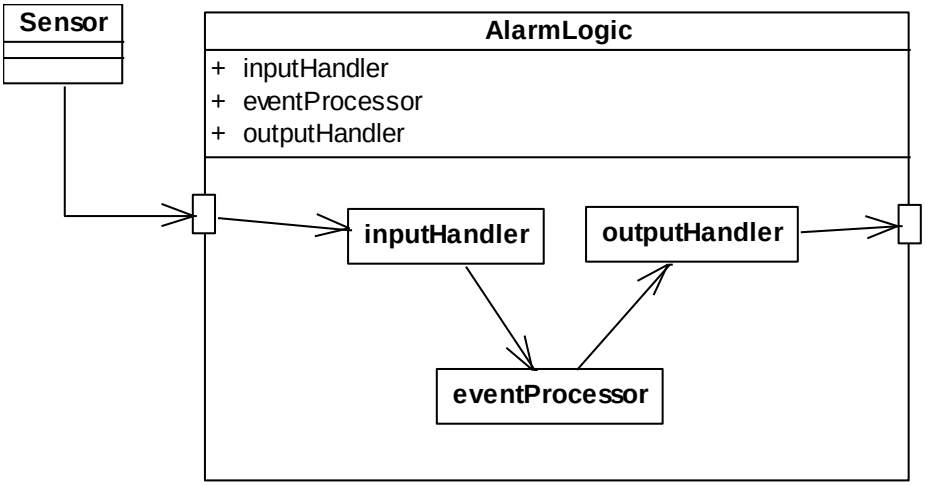
The following elements can be used in class diagrams:

Class
Interface

Relationships

See How to Create Relationships between Elements

UML Composite Structure Diagram



Overview

A composite structure diagram illustrates the inner structure of an element, usually a class or a component. In a composite structure diagram there can be ports on the edge of the element to describe "gates" to the inner structure of the element. Within the element itself you can visually explain the inner relationship of parts.

Creating a Composite Structure Diagram

To create a new composite structure diagram, choose New diagram from the Diagram menu and then choose Composite Structure diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Port

A port is a small rectangle on the edge of an element. All incoming flows can connect to a port.

Part

A part is a part of element. For a class it is an attribute.

Elements

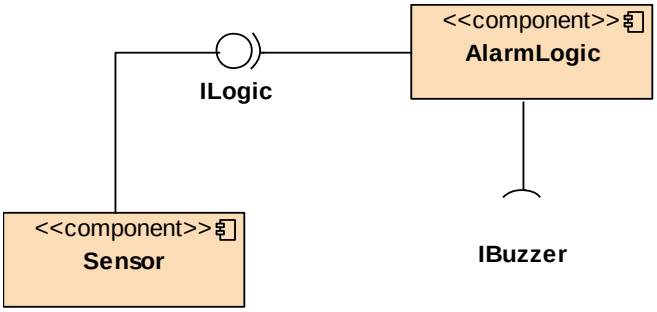
The following elements can be used in composite structure diagrams:

Class
Component

Relationships

See How to Create Relationships between Elements

UML Component Diagram



Overview

A component diagram is usually used in modeling physical aspects of the system. In other words, a component diagram describes the static implementation view of the system. With component diagrams you can visualize the implementation of component-based system. With UML2.x the concept of component is somewhat relaxed and component may be used in logical design as well. An artifact is a physical element, e.g. file, library etc.

Creating a Component Diagram

To create a new component diagram, choose New diagram from the Diagram menu and then choose Component diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

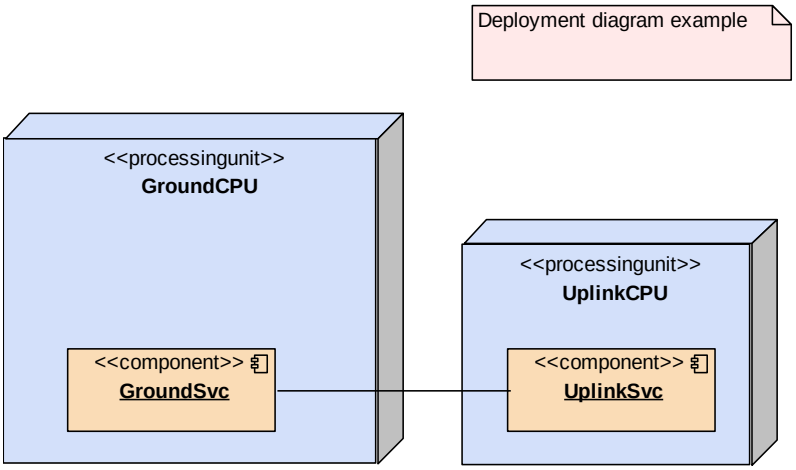
The following elements are used in component diagrams:

Component
Interface
Artifact

Relationships

See How to Create Relationships between Elements

UML Deployment Diagram



Overview

A deployment diagram shows the configuration of run time system. It shows the components and the nodes in which the components are running.

Creating a Deployment Diagram

To create a new deployment diagram, choose New diagram from the Diagram menu and then choose Deployment diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

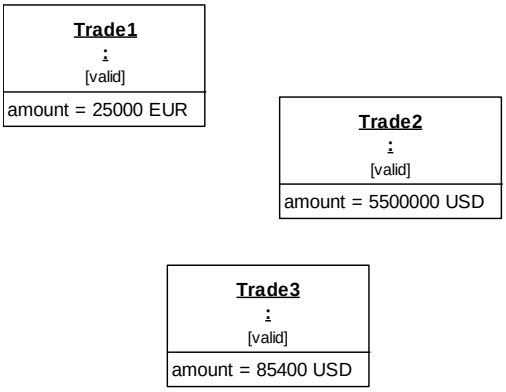
The following elements are used in deployment diagrams:

Node
Component
Interface
Artifact

Relationships

See How to Create Relationships between Elements

UML Object Diagram



Overview

An object diagrams shows a static design view or static process view of a system. It freezes a point of time and shows objects and relationships between them at that moment. Object diagrams can also be used to design static data structures.

Creating an Object Diagram

To create a new object diagram, choose New diagram from the Diagram menu and then choose Object diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

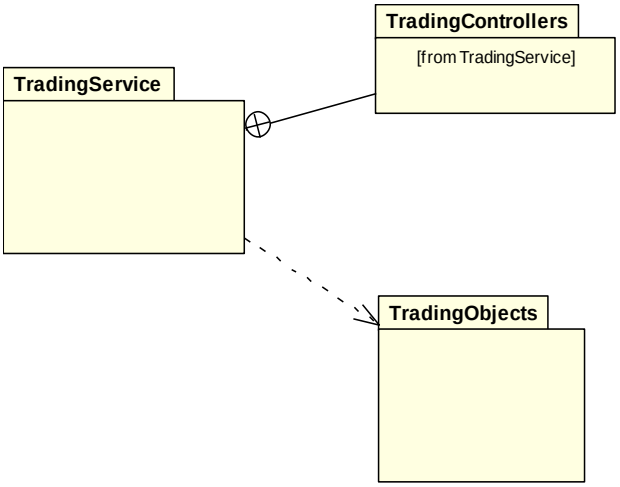
The following elements are used in object diagrams:

Object

Relationships

See How to Create Relationships between Elements

UML Package Diagram



Overview

Package diagram is used for designing the software architecture of the system. It consists of organizing the diagrams and elements in manageable groups and declaring their dependencies. A package is a subsystem or a library of classes that are semantically close to each other. A package diagram contains packages and their interfaces and relationships between them.

Creating a Package Diagram

To create a new package diagram, choose New diagram from the Diagram menu and then choose Package diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

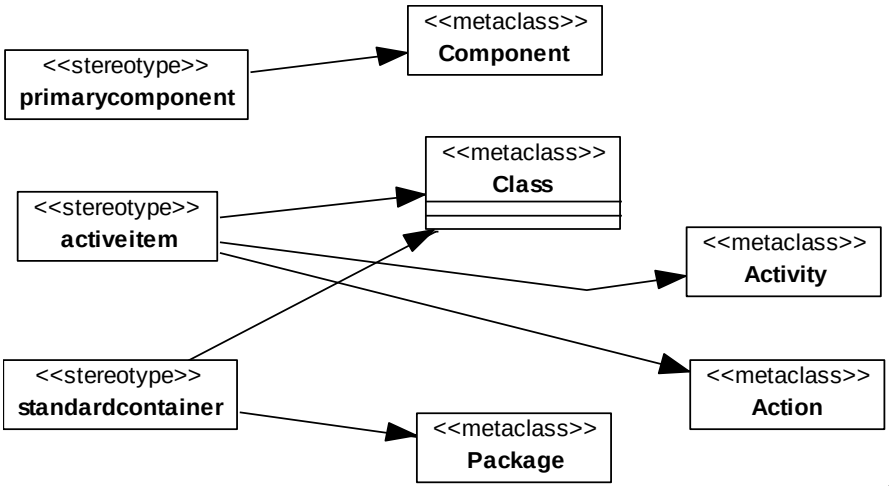
The following elements can be used in package diagrams:

Package
Interface

Relationships

See How to Create Relationships between Elements

UML Profile Diagram



Overview

Profile diagram is used for explicitly describing how stereotypes extend metaclasses. When a stereotype extends a metaclass the stereotype can be assigned to that metaclass. I.e. as above, metaclass Class is extended by two stereotypes, <<activeitem>> and <<standardcontainer>>. These both stereotypes will then be available in Class stereotype choice-menu.

For more details on profiles and stereotypes see How to Work with Profiles

Creating a Profile Diagram

To create a new profile diagram, in the model tree select by clicking the profile you want this diagram to belong to and then right-click mouse and choose New diagram. You can also use Diagram menu or toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

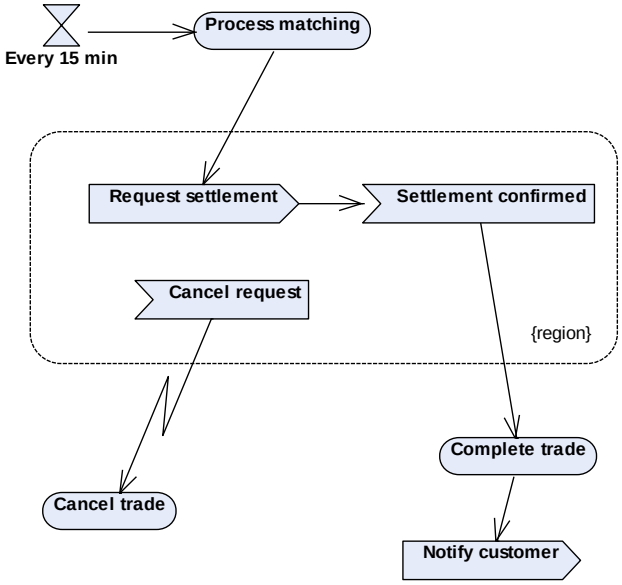
The following elements can be used in profile diagrams:

Profile
Stereotype
Metaclass

Relationships

See How to Create Relationships between Elements

UML Activity Diagram



Overview

An activity diagram describes the dynamic aspects of the system. In UML2 this has been separated from state model. An activity diagram consists of activities, actions and control flows between them. Also it can contain objects and object flows to and from them.

Creating an Activity Diagram

To create a new activity diagram, choose choose New diagram from the Diagram menu and then choose Activity diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

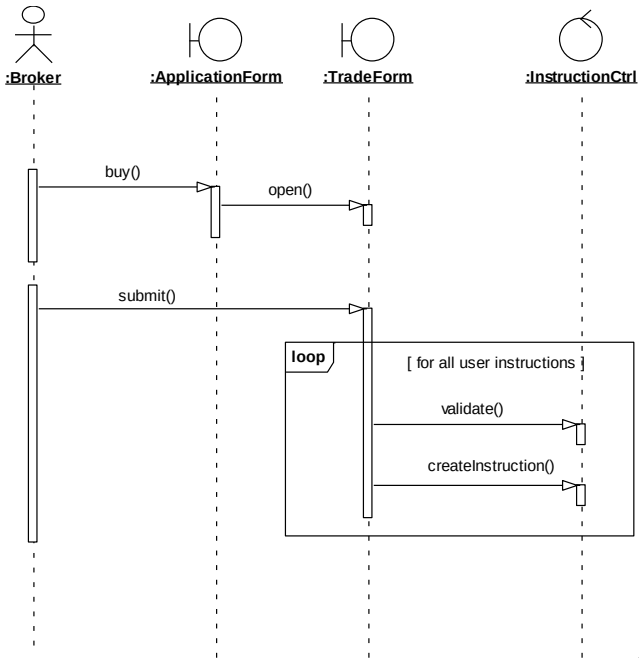
The following elements are used in activity diagrams:

Activity
Action
Control nodes
Object
Control flow
Object flow
Region
Swimlane

Relationships

See How to Create Relationships between Elements

UML Sequence Diagram



Overview

A sequence diagram describes the dynamic design view of the system. It consists of objects and time ordered messages between them. Messages can be synchronous or asynchronous. Combined fragments can be used to describe operation for a set of messages. E.g. loops, conditions, parallelism etc.

Creating a Sequence Diagram

To create a new sequence diagram, choose New diagram from the Diagram menu and then choose Sequence diagram from list of diagram types. You can also use toolbar New diagrams – icon as a shortcut to create new diagrams.

Elements

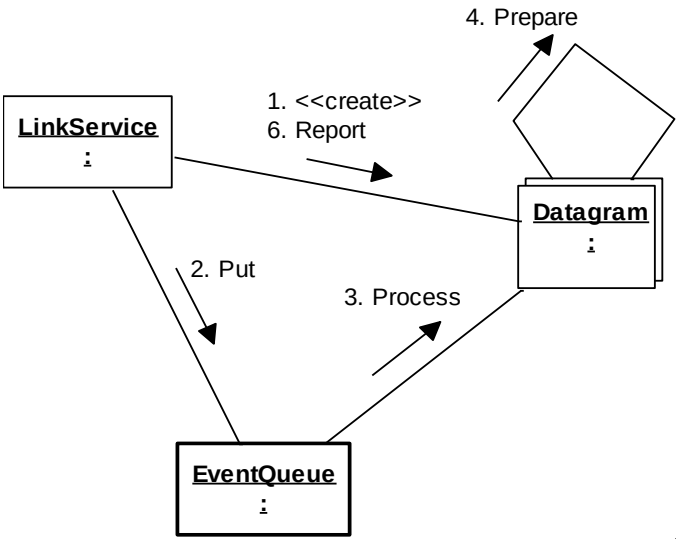
The following elements are used in sequence diagrams:

Object
Message
Focus of control
Combined fragment

Relationships

See How to Create Relationships between Elements

UML Communication Diagram



Overview

A communication diagram (pre-UML2 called collaboration diagram) is an interaction diagram that shows the structural 32eader32ation of objects and messages between them. Communication diagrams show dynamic aspects of the system. Going and coming messages can be shown on the links. These describe the events sent and received by objects. A dependency from object to a class can be used to show explicit class instantiation.

Creating a Communication Diagram

To create a new communication diagram, choose New diagram from the Diagram menu and then choose Communication diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

To show messages on the links, double click on the link to edit the link's properties and describe messages.

Elements

The following elements are used in communication diagrams:

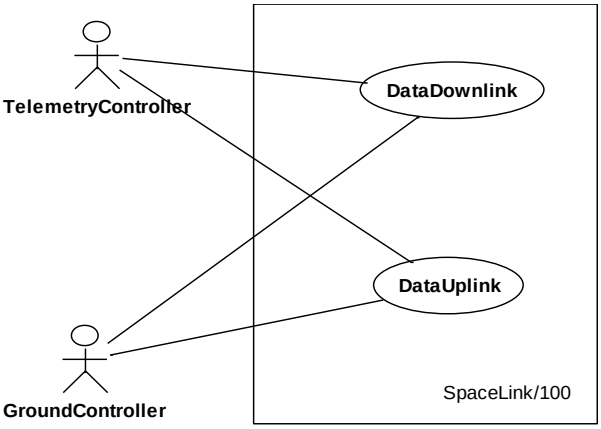
Object
Link

Link may contain messages coming from an object and going to an object.

Relationships

See How to Create Relationships between Elements

UML Use Case Diagram



Overview

A use case diagram describes the behaviors of the system. When designing a use case diagram, remember that it should describe what the system should do, not how it should be done.

Creating a Use Case Diagram

To create a new use case diagram, choose New diagram from the Diagram menu and then choose Use case diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

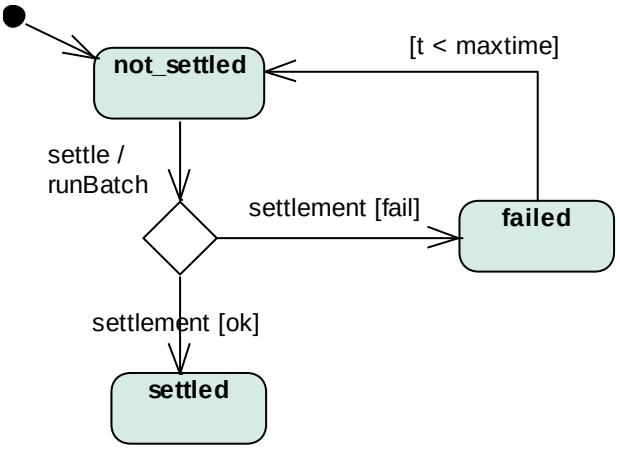
The following elements can be used in use case diagrams:

Actor
Use Case
System Boundary

Relationships

See How to Create Relationships between Elements

UML State Machine Diagram



Overview

A state machine diagram describes the dynamic aspects of the system. It consists of states and transitions between them. With state machine diagram you can describe all possible states an object may have during its lifecycle. You can also model different events that can cause state transitions.

Creating a State Machine Diagram

To create a new state machine diagram, choose New diagram from the Diagram menu and then choose State machine diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

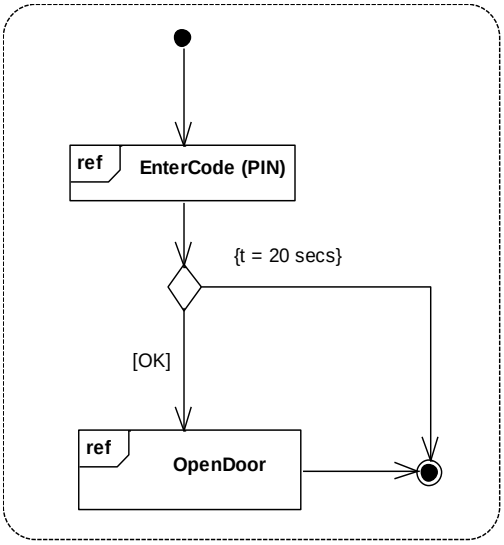
The following elements are used in state machine diagrams:

State
Transition

Relationships

See How to Create Relationships between Elements

UML Interaction Overview Diagram



Overview

Interaction Overview diagrams focus on the overview of the flow of control, thus describing high level interaction flows between more complex interactions shown only as icons. These interaction icons are called interaction use – elements. They can be understood as shortcuts to interaction diagrams which describe each interaction in detail.

Creating an Interaction Overview Diagram

To create a new interaction overview diagram, choose New diagram from the Diagram menu and then choose Interaction Overview diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

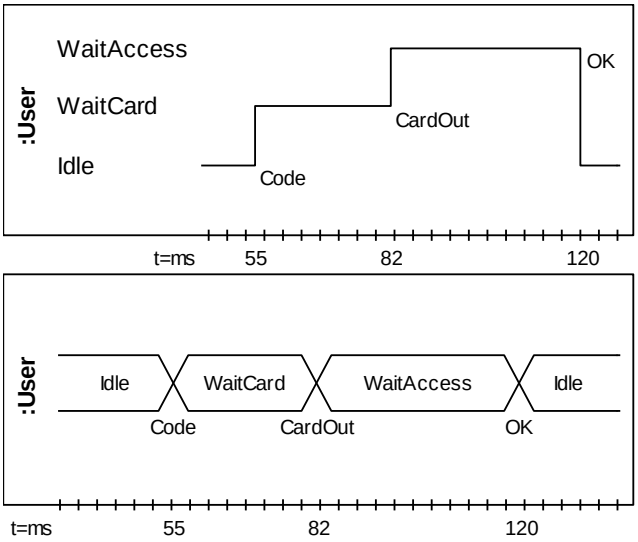
The following elements are used in interaction overview diagrams:

Interaction use
Control nodes
Control flow
Interrupt flow
Region
Swimlane

Relationships

See How to Create Relationships between Elements

UML Timing Diagram



Overview

Timing diagrams describe changes in a system's state on linear time axis. State lifelines are useful when illustrating discrete and continuous state changes, such as temperature or density. Value lifeline focus on showing points in time where the system's state changes.

Creating a Timing Diagram

To create a new timing diagram, choose New diagram from the Diagram menu and then choose Timing diagram from list of diagram types. You can also use toolbar New diagram – icon as a shortcut to create new diagrams.

Elements

The following elements are used in interaction overview diagrams:

State lifeline
Value lifeline

State lifeline and value lifeline are two different views of the same system. You can convert a state lifeline to value lifeline and vice versa.

9. How To

How to Create a Model

Create a New Model

Choose "New model" from File menu. Select model template which you want to use and click OK. If no templates are available in your model source directory, an empty model will be created. Three packages are created automatically: UseCaseView, DesignView and ImplView. Create under UseCaseView all your use cases and use case related sequence diagrams. All design diagrams should go under DesignView and implementation related diagrams like component diagrams and deployment diagrams under ImplView.

Once model is created, you can start creating new diagrams and elements. Finally, save your new diagram by choosing "Save model" from File menu. Choose a new filename for the model and click "Save".

Create Model Template

Create an empty model and set all details you want to have in any new model you create. E.g. primitive datatypes and stereotypes. Choose "Save as" from File – menu, and at the bottom select "Metamill Model Template" and type in a name for the new template and click Save to save it. Next time you create new model, a select box will appear and asks for the template. Note: save all templates under the model root directory, see How to Set Options

Edit Model Properties

Select "Model properties" from File menu or double click the root object (with Metamill icon) in the model tree to invoke the model settings dialog. These settings affect the current model only. See Set Model Properties. To modify settings for all models, see How to Set Options.

Model Tree

In the leftmost window you can see your model visualized as a tree: packages and under them elements and diagrams. When creating new elements or diagrams, select first the package you want the new element to belong to. Click mouse right button to invoke the Model Tree related pop-up menu. You can double click a diagram to open it.

How to Develop in Teams

Overview

Working as a team causes some additional requirements to the software used in development. Firstly, each team member must be able to design his own part of the model simultaneously with other designers without a risk of interfering others. And secondly, all team members must be able to see and to refer to other member's work. For these purposes, the concept of managed packages is introduced. Opened models are write-locked to prevent accidental interference between other users.

Managed Packages

A managed package is simply an independently stored package. But there are great benefits of having this. For example, single packages can be versioned under the version management system. Secondly, the main model may be made write-protected and only the packages under development can be made changeable.

Logical Paths

Logical paths in the model allow users to refer to same model from different physical installations. E.g. Model root directory may be different for different users, while they still point to the same model. See how to set options and model preferences.

How to Manage Model Elements

All model elements are stored in the internal model system. The visual elements in diagrams only point to the elements in the model.

Duplicate Element in Model

To duplicate a model element, follow the steps below:
1) Select the element on diagram you want to duplicate
2) Click mouse right button and select Deep Copy.

Note that the normal copy/paste only copies the link. The deep copy instead creates a new model element for the copy.

Delete Element from Model

To delete a model element, follow the steps below:
1) Select the element on diagram you want to delete from model
2) Click mouse right button and select Deep Delete.

Note that this removes the element from the model and all links to this element are no more valid. Hit CTRL-D for quick access.

Change Element Owner in Model

To move an element under another element:
7) Drag and drop the element in the model tree to new owner element.

Only packages can contain all kinds of elements.

To move multiple elements in one shot:
1) Select multiple elements in the model tree
2) Right click mouse and choose "Move"
3) Select new owner element
4) Right click mouse and choose "Move here"

Drag multiple elements from Model Tree

To drag multiple elements from model tree, follow the steps below:
1) Select multiple elements in the model tree
2) Move mouse to diagram empty space
3) Choose "Drag elements from tree".

Single elements can be just dragged from the model tree to diagram.

Summon related elements to diagram

If there is a relationship between elements and not all are visible, you can summon them from the model. To show related elements of a visible diagram element:
1) Select element in diagram
2) Right click mouse and choose "Summon related elements"

It will collect only directly connected elements, so you need to repeat this for summoned element to get elements from two hops away.

Find elements in diagram or model

To find a model tree element in the currently open diagram:
1) Select element in the model tree
7) Right click mouse and choose "Find in diagram"

To find a diagram element in the model tree:
1) Select element in the currently open diagram
2) Right click mouse and choose "Find in model"

Show diagrams under their direct container

Some diagrams must be stored under a container other than package. I.e. sequence diagrams and object diagrams must be stored under collaboration, activity diagrams must be kept under activities and state machine diagrams under state machines. By default all diagrams needing this sub-container are shown above the container, which makes it easier to find them in the model tree. However, you can opt to show it under its direct owner.

To show a diagram under its container, follow the steps:
1) Select diagram in the model tree
2) Right click mouse and choose "Keep under"
3) Refresh model tree if necessary

Drag and drop attributes and operations

You can copy attributes and operations by drag-and-dropping them from model tree to other model elements, and also to diagram elements.

To drag and drop an attribute or operation, follow the steps:
1) Expand a class in the model three to see the members
2) Drag an attribute or operation above a class in the diagram or other model element

How to Find Model Elements

You can search model elements using POSIX regular expressions.
Just press Ctrl+F, type in the regular expression and hit enter.

Examples

Regular Expression	Matches
MyClass.*	**MyClass**123, **MyClass**ifier, etc.
^.*Class.*	My**Class**, My**Class**123, Your**Class**123, etc.
MyClass[1|2]	**MyClass1**, **MyClass2**

See also How to Manage Model Elements.

How to Set Options

To edit common settings, choose Options from the Tools menu.

General-tab

Field	Explanation
User name	User name. The name is used in generated code.
Company	Company name. The name is used in generated code.
Automatic model backup with interval	Click this on to activate automatic model backup. Give the interval in minutes.
Backup directory	This is the directory under which the backup is saved. The backup file has ~B_ - prefix for the model file.
Local system directory (workspaces)	This is the directory where system files and workspaces are stored. Also code generation templates are here.
Always load managed packages	If checked, Metamill loads all managed packages when opening a model, without asking user's confirmation..
Small toolbar icons	If selected, explanatory text under toolbar icons is omitted.

Files-tab

Field	Explanation
Base directory for models	The models base directory is a default folder for all models. I.e. a new model is created under this directory by default. Give full path.
Managed packages base directory	Managed packages are stored under this directory by default. Can be overwritten with model specific settings.
Import code default directory	This directory is the default directory for importing external code.
Source code engineering base directory	The source root directory is the root directory for generated code. Give full path. Should not be same as import code default directory.
Document generation base output directory	This is the directory where the documents are generated by default.
Generate HTML	Click this to generate HTML documents. If not set, RTF document is generated.
MetamillScript base directory	This is the base directory for MetamillScript scripts.

Color-tab

Field	Explanation
Background	The diagram background color.
Element frames	The color used to paint element frames.
Element fill	The color used to fill elements.
Note fill	The color used to fill notes.
Relationships	The color used to paint relationships.
Package fill	The color used to fill packages.
State fill	The color used to fill states.
Component fill	The color used to fill components.
Artifact fill	The color used to fill artifacts.
Node fill	The color used to fill nodes.
Activity fill	The color used to fill activities.

Button 'defaults' restores Metamill default values.

Diagram-tab

Field	Explanation
Snap to grid	If selected, there is an invisible grid that forces the elements to be on the grid points.

With spacing	Size of the invisible snap-grid in pixels.
Export extension	Default file extension of export file
Diagram export base directory	Default directory where the exported files are stored.
Show page print hints	Visible printer page borders on diagram
Open diagrams maximized	If selected, diagrams open maximized by default
Lock inner elements by default	Lock elements so that they are not automatically set as inner elements when moved. Default for diagrams.
Switch shift+mouse direction	Changes the directions of scrolling when diagram is panned by holding shift and moving mouse
Straight lines by default	New lines are kept straight by default
Ask new diagram name	If checked, prompts name for a new diagram when new diagram is created. Otherwise default name is created automatically
Show UML2 diagram title	If checked, shows a polygon with diagram name on the left top corner of the diagram
Sort class members	If checked, sort class attributes and operations by visibility and name
Maximum class member width	Maximum class member width on single line. Members having more characters than this will be wrapped on multiple lines
Default layout max width	Maximum width for automatic layout algorithm. Single elements are not placed further to the right, but placed below the previous ones.
Element spacing	Distance between elements in automatic layout algorithm

Code-tab

Field	Explanation
Source editor	Source editor which is used to edit source when "Edit code" is called. E.g. c:\windows\notepad.exe.
Default engineering language	Language used by default when a new model is created.
	See model properties for model specific settings

How to Set Model Properties

To edit model's properties, double click root package (Metamill icon) or choose Model properties from the File menu.

General-tab

Field	Explanation
Model name	Model's name. Root item has this name. (note, this is not model file name, see below)
Documentation	Free format text about the model.

Files-tab

Field	Explanation
Model physical file	Model filename as it is saved on disk. (.xmmd file) Do not change this unless it is necessary. Normally file name gets updated when you save the model first time.
Managed packages directory	Directory under which the managed package files are stored (.xmmp files) Default: ${BASEMPACKSDIR} This means, base directory from Options. You can give e.g. ${BASEMPACKSDIR}\thismodel If you use ${MODELFILEDIR} it always points to the same directory where the actual model files is stored.
Source code directory	Directory under which the source code is generated. Default: ${BASESOURCEDIR}
Document generation output directory	Directory under which the HTML documents are generated. Default: ${BASEDOCUMSDIR}
MetamillScript directory	Directory under which the MetamillScripts are stored. Default: ${BASESCRIPTDIR}
Diagram export directory	Directory under which diagrams are exported. Default: ${BASEEXPORTDIR}

Tags-tab

Field	Explanation
Tagged values	Tagged values describing some additional features of the model.

Constraints-tab

Field	Explanation
Constraints	Model constraints.

Stereotypes-tab

Field	Explanation
Stereotypes	Declare here all stereotypes you want to use in the model. Default UML stereotypes are already included. **Note**: you can also explicitly declare stereotypes using profile diagrams.

Datatypes-tab

Field	Explanation
Datatypes	Declare here all data types you want to use in the model. E.g. if you use C++, you need int, long, string etc. Note: use model templates to store default data types.

Macros-tab

Field	Explanation
Use preprocessor	If checked, uses preprocessor to process macros and #-directives in source files. (for C/C++ code engineering)
Preprocessing macros	List macros used in code import and reverse engineering. E.g. MAX (a,b) a> b DECLARE_HANDLER int n = new myhandler() You can still use pre 5.x macros: If ends with "()", all parameters between `(` and `)' are ignored. If ends with ";" all text till next semicolon is ignored. E.g. Q_OBJECT() MM_XERP;

Code-tab

Field	Explanation
Engineering language	Implementation language used in code generation and code analyzing (reverse engineering).
C++ header ext	C++ header file extension (e.g. .hpp)
C++ impl. Ext	C++ implementation file extension (e.g. .cpp)
Ask C import	If selected, Metamill asks if a .h file should be imported as C++ or C. If not selected, C++ is used by default.
Doc in impl.	If selected, C\C++ documentation is generated in implementation file instead of header. For C++ affects only method documentation.
Code markers	**Full**: all code markers. **Medium**: only necessary code markers are generated. **None**: no code markers are generated. Note, that changes are not preserved during generation if code markers are not used.
Visual C++ class names	If checked, uses class names with C prefix in code engineering
C/C++ header pattern	(DIR)\(FILE).(EXT). DIR will be replaced with source directory, (FILE).(EXT) with file. Used in code generation.
Inc.path	Include hard coded path into #include. E,g, pattern (DIR)\headers\(FILE).(EXT) Will generate #include "headers/MyClass.hpp"
C/C++ impl. Pattern	(DIR)\(FILE).(EXT). DIR will be replaced with source directory, (FILE).(EXT) with file. Used in code generation.
C++ list	C++ list class for association attributes. Wild cards allowed: List? Generates ListMyClass ?List generates MyClassList list<> generates list<MyClass> list<*> generates list<MyClass*>
Java list	As C++ list. E.g. Vector
C# list	As above, but for C#
VB.Net list	As above, but for VB.Net
Attribute prefix	Prefix for generated attributes. E.g. m_ (can be empty)
Getters - with prefix	Check this to generate get – methods. Give get method prefix. E.g. get_mymethod

Setters - with prefix	Check this to generate set – methods. Give set method prefix. E.g. set_mymethod
Integral method implementation code	If selected, model will contain all source code! Note: model file can become very large. Use operation properties Code-tab to write implementation code. If not selected, write method implementation directly to source code within code markers. **Important**: avoid changing this in the middle of development, otherwise data may be lost!

How to Set Diagram Properties

To edit diagram's settings, double click empty canvas or choose Diagram properties from the Edit menu.

General-tab

Field	Explanation
Diagram name	Diagram's name.
Hide details	If selected, details like operation signatures are not shown. Also, only public members are shown.
Width	Diagram width (default 800)
Height	Diagram height (default 1100)
Dynamic size	Enlarge diagram automatically to show all elements
Fit to page	Fit to one page printing
Print page borders	Print page borders (corners)
Print zoom	Zoom used when printing (%100, %75, %25)
Lock inner elements	Lock elements so that they are not automatically set as inner elements when moved.
Show qualified name	If not selected, then only element's name is shown. Even inner class's owner class name is omitted.
Documentation	Free format text about the diagram.

Details-tab

Field	Explanation
Custom export file	Custom export file. Default is to use diagram's name with default export extension.

Layout max width	Overtakes default layout max width in common settings
element spacing	Overtakes default element spacing in common settings

How to Create Relationships between Elements

Relationships are lines or arrows between elements. There are different types of relationships: association, dependency, aggregate, composition, generalization, realization and extension. Not all relationship types are allowed between all elements.

Use composition when a class is fixed part of the other; if the relationship is not that strong, use simple aggregation. The dependency can be seen as "uses" – relationship. Realization is used for implementing an interface.

To create e.g. an association between two elements, choose Association from Insert menu or select Association – toolbar icon to start creating a new association. Start dragging with left mouse button the element you want the association to start from and release the mouse button on the element you want the association to end. If you release the mouse button on empty canvas, a control point is created and you can continue dragging.

Note: click mouse right button to cancel dragging.

To create other types of relationship, select the type you want and proceed as with association.

To create a new control point to a relationship, just drag the relationship with the left mouse button. To change properties of a relationship, double click it with left mouse button.

Association Dialog

General-tab

Field	Explanation
Label	Relationship's label.
Stereotype	Relationship's stereotype.
Source-End	Click this to open source end dialog
Target-End	Click this to open target end dialog
Elided	If not checked shows explicit navigation symbols
Documentation	Free format documentation about the relationship.

Details-tab

Field	Explanation
Line color	Color of the association line.

Tags-tab

Field	Explanation
Tagged values	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints. Declare no_include constraint to suppress generating #include statements in C++. Just add new constraint and replace *expression* with *no_include*.

Association End Dialog

General-tab

Field	Explanation
Role	Association end role name.
Visibility	Association end visibility
Aggregation	Association end aggregation
Multiplicity	Association end multiplicity
Ordered	Ordered flag
Navigable	Navigable flag, if selected, shows an arrow at this end
Documentation	Free format documentation.

Tags-tab

Field	Explanation
Tagged values	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Code-tab

Field	Explanation
Generate code	If not selected, no code is generated from this association end. I. e. allows you to write it manually within user blocks.
Custom code	If generation is on, you may type here what should be generated for this association end.

How to Define Attributes and Operations

Attributes and operations make the contents of a class. Double click the attribute or operation line and a dialog appears where you can set its properties. You can also type attributes and operations in free format. Just single-click the attribute or operation and wait two seconds (just like changing the label anywhere in Windows) and then type in the contents in UML specification format. This allows you to easily and quickly design the contents of high level classes. Set the constraints, like static, abstract etc., in the property dialog as described above.

Define Attributes

To define attributes, use the following format:

<visibility>$/ name : type = initialvalue

Examples:

Definition	Explanation
~ systemId : long	Variable with package visibility
+$ systemId : long = 2	Public static variable with default 2
#/ systemId : long = 1	Protected derived variable. Derived in UML means "calculated value".
- systemId : long	Private variable

Define Operations

To define operations, use the following format:

<visibility>$/ name(argument1 : type1, argument2 : type2) : returntype

Examples:

Definition	Explanation
+ run	Public method
+/ run : int	Public derived method with return type int
# run() : int	Protected method
-$ run() : int	Private static method
- run(name : string, id : int) : int	Private method with two arguments

If you define a method inline you need to give the implementation in the global user block of the header file. You may also define attribute or operation static typing a $ sign after the visibility code, e.g. +$ run() defines a static run() method. As stated in UML, derived here does not mean inheritance. A derived method means the method's return value is based on some calculations, i.e. not a direct value. A derived attribute means that attribute's value is calculated i.e. derived.

Set UML specific values (e.g. visibility) using the properties dialog Details – page. Set all code engineering specific values as constraints (e.g. inline).

Visibility

Code	Explanation
+	Public
#	Protected
-	Private
~	Package

Basic UML properties supported in Code Generation

Definition	Explanation	Applicable	Language
Classifierscope	Static	Attr/Oper	Java/C/C++/C#/VB
Frozen	Const, Final, ReadOnly	Attr/Oper	Java/C/C++/C#/VB
Abstract	Abstract, MustOverride, MustInherit	Oper/Class	Java/C++/C#/VB
Polymorphic	Virtual	Oper	C++/C#
Leaf	Final, Sealed, NotInheritable	Oper/Class	Java/C#/VB
Synchronized	Synchronized	Oper	Java
Query	Const method	Oper	C++/VB

How to Define Constraints and Tagged Values

Constraints and tagged values are UML extension mechanisms, which can be anything needed in the model domain. However, some constraints and tagged values are reserved for code engineering.

C/C++ Constraints

Definition	Explanation	Applicable	Language
auto	Auto attribute	Attr	C/C++
register	Register attribute	Attr	C/C++
mutable	Mutable attribute	Attr	C++

volatile	Volatile attribute	Attr	C++
transient	Transient attribute	Attr	C++
bitfield	Bitfield	Attr	C++
inline	Inline operation	Oper	C++
explicit	Explicit constructor	Oper	C++
no_include	Do not generate #include	Class/Asso	C/C++
pro_c	Pro/C source (.pc file)	Class	C
typedef	Typedef definition	Class	C/C++
static	Static struct	Struct	C/C++
no_impl	.cpp file is not generated	Class	C/C++
templ_no_incl	No #include "class.cpp"	Class	C++
no_getter	No getter generated	Attr	C++
no_setter	No setter generated	Attr	C++
no_getset	No getter or setter	Attr	C++

Java Constraints

Definition	Explanation	Applicable	Language
volatile	Volatile attribute	Attr	Java
transient	Transient attribute	Attr	Java
native	Native method	Oper	Java
strictfp	FP-strictness	Oper/Class	Java
static	Static class	Class	Java
no_package	Do not generate package	Class	Java
no_getter	No getter generated	Attr	Java
no_setter	No setter generated	Attr	Java
no_getset	No getter or setter	Attr	Java

C# Constraints

Definition	Explanation	Applicable	Language
event	Event	Attr	C#
readonly	Readonly attribute	Attr	C#
volatile	Volatile attribute	Attr	C#
internal	Internal operation	Attr/Oper	C#
new	New	Oper	C#
unsafe	Unsafe	Oper	C#
internal	Internal operation	Oper	C#
override	Override operation	Oper	C#
excplicit	Excplicit operator	Oper	C#
implicit	Implicit operator	Oper	C#
delegate	Delegate	Oper	C#
accessor	Accessor	Oper	C#
indexor	Indexor	Oper	C#
new	New class	Class	C#
partial	Partial class	Class	C#
internal	Internal interface	Interface	C#
no_getter	No getter generated	Attr	C#
no_setter	No setter generated	Attr	C#
no_getset	No getter or setter	Attr	C#

VB.Net Constraints

Definition	Explanation	Applicable	Language
dim	Dim declaration	Attr	VB.Net
withevents	WithEvents declaration	Attr	VB.Net
writeonly	WriteOnly member	Attr/Oper	VB.Net
shared	Shared member	Attr/Oper	VB.Net
notoverridable	NotOverridable method	Oper	VB.Net
overloads	Overloads method	Oper	VB.Net
overrides	Overrides method	Oper	VB.Net
overridable	Overridable method	Oper	VB.Net
default	Default method	Oper	VB.Net
property	Property	Oper	VB.Net
custom	Custom event	Oper	VB.Net
event	Event	Oper	VB.Net
extern	Extern member	Oper	VB.Net
delegate	Delegate	Oper	VB.Net
shadows	Shadows method/class	Oper/Class	VB.Net

ADA Constraints

Definition	Explanation	Applicable	Language
variant	Case variant	Attr	ADA
entry	Task entry	Oper	ADA
overriding	Overrides method	Oper	ADA
not overriding	Doesn't override method	Oper	ADA
null	Null procedure e.g. procedure A is null;	Oper	ADA
pure private	Type has no public forward declaration	Class	ADA
no_impl	.adb file is not generated	Class	ADA

Python Constraints

Definition	Explanation	Applicable	Language
after	Generate attribute after defs	Attr	Python

E.g. to make a C# method volatile: open method properties, select Constraints – tab and create a new constraint: volatile. Now the code generation system knows that the method has volatile constraint, and thus generates the method with volatile modifier.

To declare C++ friend classes and operations, add tagged value `friend'. Separate multiple friend – declarations with semicolon. E.g. friend = class MyClass;class MyOtherClass;

Tagged Values Supported in Code Generation

Definition	Explanation	Applicable	Language
friend	Friend declaration	Class	C++
vcflag	VisualC++ class flag	Class	C++

vc_preclass	VisualC++ class modifier (public ref class)	Class	C++
namespace	Namespace	Class	C++, C#, VB.Net
import	Import statements (i.e. Import, Using) ADA: import = Ada::Text_IO; ➔ with Ada.Text_IO; <BODY> separates .ads and .adb imports	Class	Java, C#, VB.Net, ADA, Python
using	C++: Using namespace ADA: using = Ada::Text_IO; ➔ use Ada.Text_IO; <BODY> as in import	Class	C++, ADA
includeh	#includes in .h/.hpp	Class	C, C++
includec	#includes in .c/.cpp	Class	C, C++
implements	Implements construct	Oper	VB.Net
handles	Handles event	Oper	VB.Net
externlib	Declares extern method	Oper	VB.Net
report_header	HTML report header	-	HTML
ic_header	Code generation custom block before class definition	Class	C++, Java, C, C#, ADA, Python
ic_global	Custom block after class definition	Class	C++, Java, C, C#, ADA, Python
ic_headerc	As ic_header, but for implementation file (.cpp)	Class	C++, C, ADA
ic_globalc	As ic_global, but for implementation file (.cpp)	Class	C++, C, ADA
ic_initm	Method initialization, including throws statement	Oper	C++, Java, C#
ic_throw	Method throws statement	Oper	C++, Java, C#
ic_condition	Condition as [condition]	Oper, Attr	C#, VB.Net
ce_allow_attr_init	Allow initial values in member variables	Attr	C++
forward	Forward class and function definitions. ADA: public and private forwards are separated with <PRI>	Class	C++, C, ADA
typeval	Type definition	Class	ADA
pragma	Pragma definitions	Class, Oper, Attr	ADA
pragma	Pragma=main does not create a class		Python
condition	Task entry condition	Oper	ADA
functempl	Function template	Oper	ADA
for	ADA for declaration. i.e. "for day use (Monday…	Class	ADA
caseopt	Case/when condition. e.g. caseopt = opt00 : AAA case opt00 is when AAA => …	Attr	ADA
renames	Renames a method	Oper	ADA
body	Simple task's body	Class	ADA
private	Override private type's public forward declaration	Class	ADA
discriminant	Task entry discriminant. e.g. discriminant = d → entry Q(d)(c : integer);	Oper	ADA

contract	Pre and post conditions	Oper	ADA
try	Try – except block for imports	Class	Python

Visual Constraints

Value	Explanation	Applicable
vi_nqualfrom	Non-qualified `from' text, i.e. short from text.	Class

Visual Tagged Values

Value	Explanation	Applicable
vi_show	vi_show=yes means: show item on diagram, value `no' means hide item.	Attribute, operation

Miscellaneous Constraints

Value	Explanation	Applicable
no_docgen	No document generation. Document generator skips the item and all its nested elements.	Element

How to Export Diagrams

Using clipboard

You can copy and paste diagrams using Windows clipboard. Follow the steps below:

1)	Select desired elements in a diagram. Use **ctrl – a** to select them all.
2)	Copy selected elements to clipboard using **ctrl – c**.
3)	Open your technical document (Word or other) and paste elements from clipboard to the document using **ctrl – v**. Use 'paste special' if needed.

Diagrams are copied as Enhanced Windows metafiles, which allow you to resize them on your technical document.

Exporting Diagrams

To export a diagram, choose Export from the main menu. A window opens which asks the export file name. The format in which the file is written is based on the file extension.

Supported file types are:

EMF	Enhanced Windows Metafile (also WMF will be saved as EMF)
BMP	Windows Bitmap Image
PNG	Portable Network Graphics Image
JPG	JPEG Image

- The default name for export file is based on diagram name and export file type extension.
- You can define custom export file for each diagram in the diagram properties.
- Use Export all – menu option to export all open diagrams.

Hint: if you define a custom export file and don't give any extension, the default export extension is used with the custom export file. The default export extension is defined in Options

If you select elements before export, only the selected elements are exported. This allows to extract parts of a large diagram.

How to Generate Code

There are two ways to generate code:

Generate code from diagram

To generate source code from diagram, follow the steps below:

1)	Open diagram containing elements you want to generate the code from.
2)	Activate elements you want to generate. If none is activated, all elements will be activated
3)	Select Generate Code from Tools – menu, or click Generate Code – icon on toolbar

You can generate code only from class diagrams. It would be nice to generate the whole system from use case diagrams, but that is only science-fiction. The sequence diagrams are good candidates for code generation, but not good enough. If so, they should be at the same level of abstraction as your implementation language. It is better to draw sequence diagrams to understand the domain, and use your favorite text editor to write the actual implementation logic using implementation language.

Generate code directly from model

1)	Select in the model tree the element you wish to generate code from.
2)	Right-click mouse and choose Generate code.

If you generate code for a package, all its sub-elements are generated as well.

Keeping changes in the source code

Make all changes inside user blocks, i.e. code markers like:

```
//#UBLK-BEG-GLOBAL
  this code here is not overwritten when generated…
//#UBLK-END-GLOBAL
```

Don't remove any of the user blocks, otherwise the code generation system won't be able to determine where you made changes.

The code will be generated in language you selected in model properties. See How to Set Model Properties to get more information about setting the language. See also How to Define Attributes and Operations.

Note also, that code inside custom blocks is not analyzed, i.e. reverse engineered. This allows you to declare code that is purely implementation specific and does not interfere with UML model.

Note: if you remove a method and then re-generate the code, the changes you made to the old method are copied to the end of generated file. This allows you to copy/paste the contents to a new method.

Integral code

All code including implementation code is generated from the model.
You can use the following tagged values:

ic_header – contents are generated before the class declaration
ic_global – contents are generated at the end of source file

ic_headerc – same as ic_header, but in the .cpp file
ic_globalc – same as ic_global, but in the .cpp file

ic_initm – constructor init and method throws clause, i.e. throws Exception
ic_condition – C# method condition i.e. [WebMethod]

How to Reverse Engineer Code

There are two distinctive situations where code is reverse engineered. Firstly, external old code is imported to Metamill and new elements are created to the model, secondly, existing elements can be updated from the generated code.

Importing external source code

To import external old code by reverse engineering it, follow the steps below:

1)	Create empty class diagram where you want to add reverse engineered elements. You can also import code to an existing diagram; existing classes are preserved. NOTE: if you have many classes to import, it is recommended to import directly to model and later drag elements to diagrams. To do this, do not check 'Import to diagram' checkbox when starting the code import.
2)	Select "Import code" from Tools – menu. Choose files to be reverse engineered. New elements are created in the model and added to current open diagram if 'Import to diagram' is checked.

	Note: you can import multiple files by pressing shift – key simultaneously when selecting files. Alternatively you can import also subdirectories. See "Scan with pattern" below.

Import root

When importing code, you need to give import root directory. This directory will be considered as a root for all package definitions. Package is resolved from the filename.

E.g.

Import root: c:\mysrc\data
File to import: c:\mysrc\data\tools\misc\MyTokenizer.java

Will become a class: tools.misc.MyTokenizer and will be created under DesignView package.

Missing packages will be created as necessary.

Note: you need to give the same import root if you import again to the same model. You can import as often as you want, existing classes are updated with imported data.

Search rules

You may declare search rules for code import. This mainly affects the way class' base classes are assigned.

E.g. you may give: Tools::*;System::*

This means that when Metamill finds a class deriving from HashMap, it first searches for a class Tools::HashMap or even Tools::Anything::HashMap. If there is no such class, then System::HashMap is searched and so on.

Scan with pattern

You can recursively import subdirectories. Check "Scan with pattern" checkbox and give search pattern (e.g. *.java, i.e. all java files found under the given import root directory). You can also give a subdirectory where to start the recursive scan. This is needed when you want to import recursively only a single package while keeping the original import root.

Check boxes

Import class details	If you uncheck this, only class names will be reverse engineered. This is needed when you want to import large amounts of data, but want only the type names. E.g. when importing Java – toolkits.
Recurse subdirectories	Uncheck this if you want to ignore subdirectories.
Resolve dependencies	If checked, Metamill tries to build dependency relationships based on the code content. C++ and Java only.

Import to diagram	If checked, imported classes are added to currently active class diagram.

Synchronizing model with source code

To update current model elements by reverse engineering a related source file, follow the steps below:

1)	Select the class you want to update from source file. If you select a package all sub-elements are analyzed as well.You can select class in the model tree or a class in the diagram.
2)	Select Analyze Code from Tools – menu. Source code for selected classes is analyzed and classes are updated. Note: to analyze a class you need a related source file, which you can obtain by generating the code from diagram.

By default, the code is reverse engineered in language you selected in model properties. See How to Set Model Properties to get more information about setting the language. You can also choose custom language for each class in your diagram by setting class properties.

To know more about how attributes and operations are constructed see How to Define Attributes and Operations.

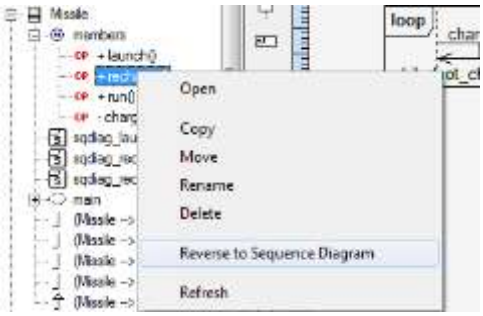
Processing Macros

Sometimes the source code contains user defined macros that are not part of the standard language (e.g. Q_OBJECT). It is possible to ignore them or declare substitute values for them and thus avoid error messages when reverse engineering the source code. See How to Set Model Properties to get more information about declaring macros.

How to Reverse Engineer Sequence Diagrams

Reverse engineering sequence diagrams means analyzing method contents, the actual implementation code, and then automatically creating a sequence diagram which gives a high-level understanding on method calls. Its purpose is not to be one-to-one with the implementation language code. Sequence diagrams can be created by method basis. Currently supported languages are C++ and Java.

Reverse Engineer Sequence Diagram



In model tree, select class and expand its members so that you can see attributes and operations. Select operation, right mouse click and select "Reverse to Sequence Diagram". If reverse engineering is successful and method calls are detected, then the sequence diagram is created and opened as current diagram. Method calls and entities must be known to the model, i.e. must be stored in the model. Unknown method calls are ignored. Also, class implementation code must be generated or if integral model is used, stored in the method implementation code.

If you don't want to keep the created diagram, you can easily delete it by right mouse clicking empty area of the diagram and then choosing "Delete".

How to Generate RTF or HTML Documentation

To generate documentation from the model

1)	Select a package you want to generate in the model tree. (usually root)
2)	Select `Generate document' from Tools – menu
3)	Type in the directory under which the document will be created
4)	Click OK to generate

RTF Document

By default the file name is document.rtf. Diagrams are embedded as PNG images.

HTML Document

File mmindex.html contains the master index of all elements.
File pmindex.html contains the package index of all packages.

Diagrams are generated in PNG format.

It is not possible to alter the way the document is generated. For this purpose, use MetamillScript or generic XML tools to produce custom documentation.

You can override the report header text by creating a tagged value "report_header" to the model root.

How to Write MetamillScript – scripts

MetamillScript is a simple scripting language which can be used to manipulate Metamill models.

Please refer to chapter Writing Metamill Script for more information.

How to Import/Export XMI

Import XMI

To import an external XMI file:

1)	Select a package under which you want to import the new elements
2)	Select `Import model → Import XMI' from Tools – menu
3)	Choose a file you want to import
4)	Click OK to import

Contents of the package are destroyed and the elements imported from the file are created under the package.

Metamill automatically detects the XMI version. Supported versions are 2.1 and 2.0.

Export XMI

To export a package to an external XMI file:

1)	Select a package which you want to export
2)	Select `Export model → Export XMI' from Tools – menu
3)	Choose a file you want to create by exporting
4)	Click OK to export

How to Import/Export Rose .MDL Files

Import .mdl

To import a Rose .mdl file:

1)	Select model root
2)	Select `Import model → Import .mdl' from Tools – Import model – menu
3)	Choose file you want to import
4)	Click OK to import

Contents of current model are destroyed and elements imported from the file are created under the package. Also diagrams are imported as much as possible.

Export .mdl

To export a Rose .mdl file:

1)	Select a package which you want to export
2)	Select `Export model → Export .mdl' from Tools – menu
3)	Choose a file you want to create by exporting
4)	Click OK to export

How to Work with Profiles

Using profiles is a UML way to describe metaclass extensions. This means mainly telling to the system which stereotypes are available for which metaclass classes. Profiles are created under ProfileView package.

NOTE: it is also possible to create stereotypes "in an old way" in model properties stereotype tab. You may want to avoid creating explicit profile diagrams, because model files will be slightly bigger when using them.

Creating a Profile

To create a profile:

1)	Select ProfileView
2)	Right click mouse and select New profile

To create a profile diagram:

1)	Select a profile under ProfileView
2)	Right click mouse and select New diagram

Stereotypes and Metaclasses

Under profiles you can create profile diagrams in the same way. In a profile diagram you can explicitly describe which stereotypes extend which metaclasses. When you create a stereotype, you can give documentation, tagged values and constraints. These are copied to the element which the stereotype is assigned to. It helps

You have this kind of item in elements which have stereotype capability:

You can choose the stereotype from the drop down menu, which also contains standard UML stereotypes. When you click 'profile' – button, you can select stereotypes which are explicitly described in the profile diagrams.

How to Buy License

To buy a license, follow the steps below.

1)	Download the evaluation version of Metamill (at www.metamill.com, also look for Metamill at shareware providers)
2)	Install and verify that the evaluation version runs on your machine.
3)	Go to Metamill web site at www.metamill.com and choose Buy License.
4)	Follow the instructions on web pages to buy the license (all major credit cards accepted).
5)	The user id and license key will be delivered to you by e-mail soon after you have completed the purchase.
6)	Start evaluation version and type in the user id and license key.
7)	Restart Metamill and the full, unlimited version is ready for use.

Don't forget to give all necessary information when filling the order. Note also, that processing the secure credit card order may take some time, please do not hang up until it's finished. If you have problems with your order, please see Support for information about how to get help. See also the FAQ at Metamill Website.

Benefits of Buying a License

* You do not use the software illegally
* You can add unlimited number of elements to models
* You get free on-line support
* You get future improvements and bug fixes for free
* You can decisively say Metamill is part of your IT strategy
* You support development to make Metamill even better

Validity between Releases

Your license is valid for all minor releases (like 8.1 and 8.2 etc.) published after the main release, that means you get possible improvements and bug fixes to it for free. For future major new releases (like 9.0) you need a new license if not stated otherwise (usually with reduced price).

See Metamill website at www.metamill.com for up to date information.

10. Reference

Actor

actor

An actor is an external user of the use cases. An actor represents a human, external device or another system that interacts with the system. Actors are used in use case diagrams.

To create an actor, choose Actor from Insert menu or use Actor – toolbar icon to create a new actor. Place it to the canvas by clicking the left mouse button. To change actor's properties, double click the actor.

General-tab

Field	Explanation
Name	Actor's name
Stereotype	Actor's stereotype.
Icon mode	Icon mode or box-mode for Actor
Autosize	If selected, size is automatically fit
Documentation	Free format documentation about the actor.

Tags-tab

Field	Explanation
Tagged value	Tagged value describing some additional features of the actor. Note: semicolon `;' marks end of line.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Use Case

A use case describes what the system does. No implementation related issues should be put into use cases, since they are analysis tools and they explain what is required from the system. Using use cases you can describe also a subsystem, a class or an interface. Use cases are used in use case diagrams.

To create a use case, choose UseCase from Insert menu or use UseCase – toolbar icon to create a new use case. Place it to the canvas by clicking the left mouse button. To change use case's properties, double click the use case.

General-tab

Field	Explanation
Name	Use case's name
Stereotype	Use case's stereotype.
Icon mode	Icon mode or box-mode for use case
Autosize	If selected, size is automatically fit
Documentation	Free format documentation about the use case.

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)
Diagram	Shortcut to a linked diagram. (diagram opens on double click)
Extension points	Use cases extension points. Put each extension point on its own line.

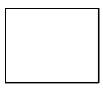
Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

System Boundary



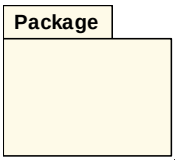
System boundary illustrates the boundary between the system and the actors. Actors see the system as a black box. System boundaries are used in use case diagrams.

To create a system boundary, choose SystemBoundary from Insert menu or use SystemBoundary - toolbar icon to create a new use case. Place it to the canvas by clicking the left mouse button. To change system boundary's properties, double click the system boundary.

General-tab

Field	Explanation
Name	Name of the system the use cases belong to.
Documentation	Free format documentation about the system.

Package



A package represents a group of elements that are semantically close to each other. In general packages are just folders where elements are stored. Package structure can be designed in package diagrams.

To create a package in package diagram, choose Package from Insert menu or use Package - toolbar icon to create a new package. Place it to the canvas by clicking the left mouse button. To change package's properties, double click the package.

You can create a package directly to the model using the model tree view. Point the package you want the new package to appear under and click right mouse button, then select 'New package'.

General-tab

Field	Explanation
Name	Package's name.
Stereotype	Package's stereotype.
Global	Select package as global if the package is used by every other package. Do not draw dependencies to a global package.
Autosize	If selected, size is automatically fit
Documentation	Free format documentation about the package.

Details-tab

Field	Explanation
Managed package XMI file	If this package is a managed package, this is the file name for the package file.

Generate code for owned classes	If not selected, no code is generated for owned classes.
Fill color	Fill color for visual diagram element (empty = default color)
Diagram	Shortcut to a linked diagram (diagram opens on double click)

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

The tabs below are present only for managed packages:

Stereotypes-tab

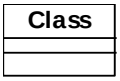
Field	Explanation
Stereotypes	Stereotypes that belong to a managed package.

Datatypes-tab

Field	Explanation
Datatypes	Data types that belong to a managed package.

Note: if model is not read-only these types will be propagated to the model root automatically. It is better to always add stereotypes and data types directly to the model, if it is writable.

Class



A class is a basic modeling element, which describes common operations for a set of objects. Classes are used in class diagrams.

To create a class, choose Class from Insert menu or use Class - icon as a shortcut to create a new class. Place it to the canvas by clicking the left mouse button. To change class's properties, double click the class.

General-tab

Field	Explanation
Name	Class's name.
Stereotype	Class's stereotype.
Icon mode	If selected, the icon presentation is used if it is supported.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the class.

Details-tab

Field	Explanation
Multiplicity	Multiplicity of the class. I.e. number of instances of class at maximum can exist.
Abstract	If selected, the class is abstract. I.e. it cannot be instantiated.
Leaf	If selected, the class is leaf. I.e. it cannot be derived from.
Active	If selected, the class is active. Active class has its own thread of control and thus can perform in parallel with other threads.
Synchronized	If selected, the class is marked as synchronized
Parameterized class	If selected, the class is a parameterized class. In C++ this means the class is a template class.
Parameters	If the class is a parameterized class, here are the parameters.
Visibility	Class visibility
Fill color	Fill color for visual diagram element (empty = default color)

Attributes-tab

Field	Explanation
Visible	If selected, Attributes - compartment is visible.
Attributes	Attributes of class. See attribute dialog below.

Operations-tab

Field	Explanation
Visible	If selected, Operations – compartment is visible.
Operations	Operations of class. See operation dialog below.

ExtraData-tab

Field	Explanation
Visible	If selected, ExtraData – compartment is visible.
ExtraData	Arbitrary additional data. Normally used to explain responsibilities of the class. You can type anything, the format is not restricted.

Tags-tab

Field	Explanation
Tagged value	Tagged values. Note: semicolon `;' marks end of line.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Code-tab

Field	Explanation
Custom language	Custom engineering language. If not selected the default engineering language from options is used.
Custom source file	Custom source file. If not selected, the default file name is used based on settings in options. This is used in code generation and code reverse engineering. In C/C++ this is the header file.
Custom impl. file	Not used in Java, C#, VB.Net In C/C++ this is the implementation file.
Generate code	If not selected, no code is generated for this element.

Interface

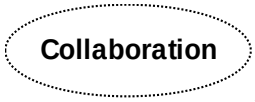
Interface

Interface is an abstract element that describes the access interface to a subsystem or to a set of classes collaborating together. Interface should have only static members and abstract operations.

To create an interface, choose Interface from Insert menu or use Interface - toolbar icon to create a new interface. Place it to the canvas by clicking the left mouse button. To change interface's properties, double click the interface.

Interface is just a stereotyped class, see Class

Collaboration



A collaboration describes a structure of collaborating elements, each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.

General-tab

Field	Explanation
Name	Collaboration's name.
Stereotype	Collaboration's stereotype.
Autosize	If selected, size is automatically fit
Documentation	Free format documentation about the collaboration

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)
Diagram	Shortcut to a linked diagram (diagram opens on double click)

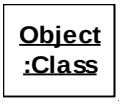
Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Object



Object is normally understood as an instance of a class. In high-level sequence diagrams the design classes are not yet known and the object represents something in the real world or inside the system. Objects are used in sequence diagrams.

Each object has a lifeline where time flows from up to bottom. Messages are attached to the object lifelines.

General-tab

Field	Explanation
Name	Object's name.
Class	Class of the object or an actor. Can be unspecified.
Stereotype	Class's stereotype.
Icon mode	If selected, the icon presentation is used if it is supported.
Show class name	If selected, the class name is shown after object's name.
Multiobject	If selected, presents a set of objects. For example, if one object interacts with many transaction objects in the same way, the transaction may be declared as multi-object.
Active class	If selected, the object is active. It has its own thread of control. Rendered with thick boundary.
Documentation	Free format documentation

Values-tab

Field	Explanation
State	Object's state in free text (only in object and communication diagrams)
Visible	If selected, the attribute values are shown. (only in object and communication diagrams)
Attribute values	Attribute values
Fill color	Fill color for visual diagram element.

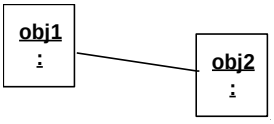
Tags-tab

Field	Explanation
Tagged value	Tagged values. Note: semicolon `;' marks end of line.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Link



A link is a relationship between objects. Links are used in collaboration and object diagrams. Messages may be shown on a link - these are same messages which are used in sequence diagrams, they are only presented here in a different way.

General-tab

Field	Explanation
Stereotype	Link's stereotype at the far end by default.
Documentation	Free format documentation about the link.

Details-tab

Field	Explanation
Fill color	Fill color for diagram line (empty = default color)

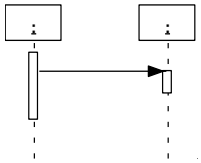
Messages-tab

Field	Explanation
Going messages →	
Procedural	Procedural message, shown with stick arrow
Asynchronous	Asynchronous message, shown with half-arrow
(contents)	Message contents. Add all going messages to this field. Use number indicating the order of messages if needed. E.g. 1. setPriority(1) 4. readData(n)
Coming messages ←	
Procedural	Procedural message, shown with stick arrow
Asynchronous	Asynchronous message, shown with half-arrow
(contents)	Add all coming messages here.

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Message



A message is the specification of communication from one object to another. There are always a sender and a receiver. The receiving of a message can be understood as an incoming event, or just as a member function call in implementation language like C++ or Java ™. Messages are used in sequence diagrams.

Message to self

The sender and receiver may be the same object. In this case the arrow is drawn back to the object. This may be understood as an object triggering its own event or calling object's own member function.

Return message

Returning a value to a caller may be explicitly described using return message. In implementation language this usually means return value from a function. See also below the focus of control. A return message can be added only to the end of a focus of control.

Asynchronous messages have no focus of control - they are considered as events. Start and end of asynchronous messages may differ, to show the time it takes to send the message. Open message properties to set timing constraints.

Create Message

This is a message with <<create>> stereotype. Object life-line starts at this message.

Destroy Message

This is a message with <<destroy>> stereotype. Object life-line ends at this message.

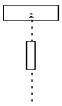
General-tab

Field	Explanation
Label	Message's label.
Documentation	Free format documentation about the class.

Details-tab

Field	Explanation
Message type	
Call	If selected the message is synchronous and it blocks the caller.
Send	If selected the message is asynchronous and doesn't block the caller.
Time observation	Source = Time at the source end (e.g. t=n) Target = Time at the target end (e.g. t=n+2) Constraint appears in { }
Fill color	Fill color for diagram line (empty = default color)

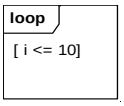
Focus of Control



Focus of control is a thin rectangle on a lifeline. It describes the period of time during which the object is performing an action. Focus of control starts when a message starts and ends when the message has been performed. The return messages are used to explicitly mark the end of focus. Focuses can be nested and stacked on top of each other child focuses being slightly aligned to the right. Focuses of control are created automatically when synchronous messages are inserted in a diagram.

Create return message starting from a focus and drag it down the lifeline to adjust the size of focus of control.

Combined Fragment



Combined fragment is a visual aide for describing operations on elements contained in the fragment. It is drawn as a rectangle with operator on the top left corner.

Below are the properties of combined fragment:

General-tab

Field	Explanation
Operator	Operator to apply for the elements inside rectangle
Operands	Any condition which guides the behavior of the operator Use dot (".") in a single line as partition delimiter. e.g. [x>1] . [else]

State

A state is understood as a condition or situation during the lifecycle of an object. A state may wait for some event, it may perform some activity. States are used in statechart diagrams.

There are five different types of states:

Field	Explanation
Initial state	Initial state where transitions can only start
Final state	Final state where transitions can only end
Shallow History	History state is a special state to remember the last substate that was active
Deep History	History state which can be used globally in a state machine
State	Normal state between initial and final states

Below are the properties of a state:

General-tab

Field	Explanation
Name	State's name.
Stereotype	State's stereotype.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the state.

Advanced-tab

Field	Explanation
Visible	If selected, advanced - compartment is visible.
Actions, activities and events	Advanced behavior of state. Type actions and events in the following format: entry / doEntry() - one action per row.
Fill color	Fill color for visual diagram element (empty = default color)
Diagram	Shortcut to a linked diagram. (diagram opens on double click)

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

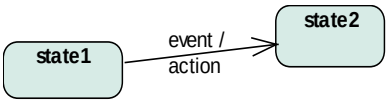
Choice

Choice specifies alternate paths of transitions taken based on some condition. A choice usually has one incoming transition and many outgoing transitions. On each outgoing transition a guard condition is described. These conditions are evaluated when the flow enters the branch.

General-tab

Field	Explanation
Name	Choice label
Documentation	Free format documentation.

Transition



A transition is a relationship between states. It describes conditions that cause state changes. An event may trigger state transition and the guard condition checks if an action must be performed. Transitions are used in state machine diagrams.

Format:

Event [guard condition] /

Action

General-tab

Field	Explanation
Event	Event is a stimulus that can trigger state transition
Guard	Guard condition is a boolean expression and if it is true the action is performed.
Action	Any atomic computation like operation or signaling
Stereotype	Transition's stereotype.
Documentation	Free format documentation about the transition.

Details-tab

Field	Explanation
Fill color	Fill color for diagram line (empty = default color)

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

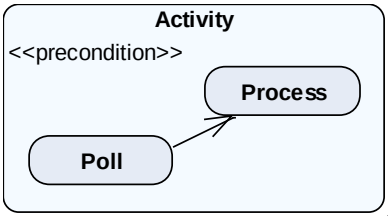
Fork/Join

Fork and Join are synchronization bars used for describing parallel flows of control. The Fork divides the flow into many simultaneous flows and the Join then joins these flows into one flow again. The Fork may have only one incoming transition and many outgoing transitions. Similarly, the Join can have many incoming transitions and only one outgoing transition.

General-tab

Field	Explanation
Name	Synchronization element label
Documentation	Free format documentation.

Activity



An activity is the specification of parameterized behavior where individual elements are actions. Activities are used in activity diagrams.

Below are the properties of an activity:

General-tab

Field	Explanation
Name	Activity's name.
Stereotype	Activity's stereotype.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the state.

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)
Diagram	Shortcut to a linked diagram (diagram opens on double click)

Advanced-tab

Field	Explanation
Details visible	Show activity details on screen
Activity details	Activity details. Here you can describe for example preconditions and postconditions.

Tags-tab

Field	Explanation
Tagged value	Tagged values.

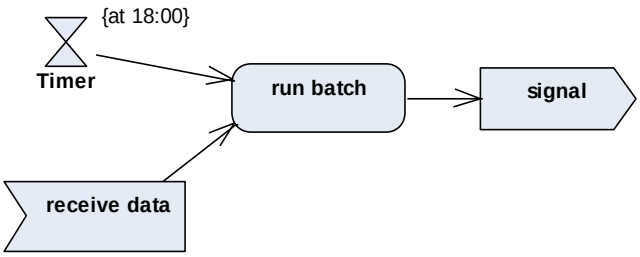
Constraints-tab

Field	Explanation
Constraints	Constraints.

Action

Action

An action is the fundamental unit of behavior specification. Actions are used in activity diagrams.
In the following diagram are show Accept time event action (hour class), accept event action (convex rectangle), call operation action and send signal action.



Below are the properties of an action:

General-tab

Field	Explanation
Name	State's name.
Stereotype	State's stereotype.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the state.

Advanced-tab

Field	Explanation
Action kind	Possible values: Undefined Accept event Accept time event Send signal Broadcast signal Call behavior Call operation Raise exception
Details visible	Show action details on screen
Action details	Action details.
Effect	Action's effect, e.g. when receiving events
Fill color	Fill color for visual diagram element (empty = default color)

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

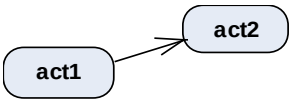
Control Nodes

Control nodes are used in activity diagrams. From left to right: Decision/Merge, Fork/Join, Initial node, Activity final and Flow final.

General-tab

Field	Explanation
Name	Synchronization element label
Documentation	Free format documentation.

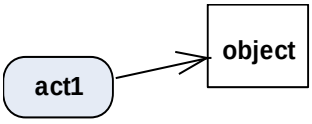
Control flow



Control flows describe flows of control between control nodes. They are used in activity modeling.

See How to Create Relationships between Elements

Object flow



With Object flows it is possible to show which objects are involved in a flow of control.

See How to Create Relationships between Elements

Interrupt flow

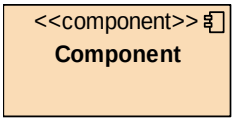
An interrupt flow, also called interrupting edge, describes an exception handler for a control node.

See How to Create Relationships between Elements

Swimlane

Swimlanes can be used for partitioning the activity diagram into divisions. One division may describe, for example, one business unit which will perform activities in that division. Swimlanes are thick vertical bars that divide the diagram.

Component



A component is a physical part of the system that can realize a set of interfaces. A component is easy to replace with another component as long as it implements properly its interfaces. A component can be an executable, a library, a source or data file or any other physical part of the system. See artifact below for a physical piece which has no clearly defined interface and is more atomic than a component.

Below are the properties of a component:

General-tab

Field	Explanation
Name	Component's name.
Stereotype	Component's stereotype.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the component.

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)
UML 1.x icon	Show icon with protruding blocks as defined in UML 1.x
Instance	Instance, show name underlined

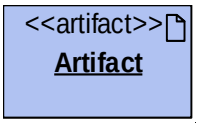
Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

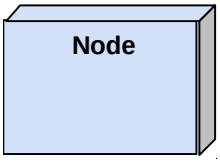
Field	Explanation
Constraints	Constraints.

Artifact



An artifact is the specification of a physical piece of information. Examples of artifacts are model files, source files, libraries, e-mail messages. An instance of artifact is the single copy of the object, which is actually stored as a single file or single library. Instances of artifacts are shown with name underlined. An artifact has similar properties as components.

Node



A node is a physical part of the system that executes components. One node may contain multiple components. Deployment diagrams are useful to describe the functioning of a distributed system where system is distributed across multiple processors or across network.

Below are the properties of a node:

General-tab

Field	Explanation
Name	Node's name.
Stereotype	Node's stereotype.
Autosize	If selected, element size is adjusted automatically.
Documentation	Free format documentation about the node.

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)
Instance	Instance, show name underlined

Tags-tab

Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Note: resize the node large enough to allow you to insert components into it.

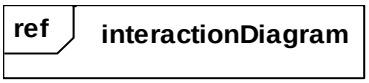
Region

A region is a simple boundary element which can be used to visually group elements. You can move region's label by holding CTRL and moving mouse.

General-tab

Field	Explanation
Label	Region's label
Documentation	Free format documentation about the region.

Interaction Use



An interaction use is a shortcut to interaction. Its main purpose is to reference interaction diagrams, but it can reference any kind of diagram.

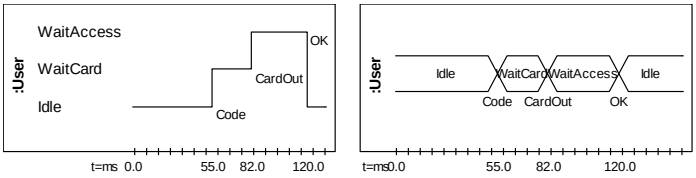
General-tab

Field	Explanation
Diagram	Referenced diagram name, use select – button to set.
Call parameters	Call parameters for diagram call
Documentation	Free format documentation about the interaction use.

Details-tab

Field	Explanation
Fill color	Fill color for visual diagram element (empty = default color)

State lifeline / Value lifeline



There are two kinds of lifelines used in timing diagrams. State lifelines (picture above left) are useful when illustrating discrete and continuous state changes, such as temperature or density. Value lifeline (above right) focus on showing points in time where the system's state changes.

Below are the properties of state and value lifelines:

General-tab

Field	Explanation
Lifeline name	Name of lifeline (vertical).
Lifeline type	State lifeline or value lifeline
Show event names	Show name of each event on the lifeline
Documentation	Free format documentation about the node.

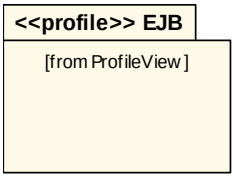
States-tab

Field	Explanation
Lifeline states	List of lifeline states State properties: State name Initial state. Check to make current an initial state
Initial state	Name of initial state, set by state properties

Events-tab

Field	Explanation
Lifeline events	List of events, sorted by transition time Event properties: Event name: name of the event Transition time: time when the event occurred Transition to state: state transition Note: state transition must point a state in States list.
Time unit	Time unit shown on the bottom left corner
Force min/max	Force explicit minimum and maximum times for lineline. If not check these are automatically calculated using event data.

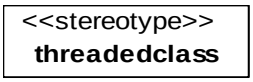
Profile



Profile is a container for metaclass extension declarations, profile diagrams, stereotypes, metaclasses and extension associations between the last two.

Profile is a similar container as package.

Stereotype



Stereotype are used for extending metaclasses and are used explicitly only in profile diagrams. In the above case, the actual stereotype will be <<threadedclass>>.

Below are the properties of a stereotype:

General-tab

Field	Explanation
Name	Stereotypes's name.
Documentation	Free format documentation about the stereotype.

Tags-tab

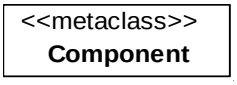
Field	Explanation
Tagged value	Tagged values.

Constraints-tab

Field	Explanation
Constraints	Constraints.

Note: documentation, tagged values and constraints are copied when the stereotype is assigned to an element.

Metaclass



Metaclass describes a metaclass like Component or Class. They are used explicitly only in profile diagrams.

Below are the properties of a metaclass:

General-tab

Field	Explanation
Name	Metaclass's name.
Documentation	Free format documentation about the metaclass.

Available metaclass names are:

Metaclass	Note
Action	
Activity	
Artifact	
Association	Includes aggregations, compositions etc.
Attribute	
Class	
Collaboration	
Component	
ControlFlow	
Dependency	
Generalization	
Interface	
Link	
Node	
Object	
ObjectFlow	
Operation	
Package	
PseudoState	e.g. initial state, deep history, shallow history, etc.
Realization	
State	
Transition	
UseCase	

Extension

Extension describes the association between a metaclass and stereotype. Stereotype extends the given metaclass. Extensions are used only in profile diagrams.

Below are the properties of an extension:

General-tab

Field	Explanation
Label	Extension's name.
Documentation	Free format documentation about the extension.

11. MetamillScript

Overview

MetamillScript is a simple scripting language which can be used to manipulate Metamill models. This implementation contains the basic set of commands. Please note that the idea of MetamillScript is not to be another full scale programming language (like Perl, Python), its purpose is to make simple custom model manipulations possible. If you need detailed access to the model, it can be done by accessing the model files which are open standard XMI, i.e. XML files.

Commands

script

<script> = script <identifier> '('<arguments>')' <declare> begin <block> end.

Script starts with a keyword 'script' followed by script name. The script has fixed arguments which are (String rootid). Contents of script are put between 'begin' and 'end.'

```
script testscript(String rootid)
begin
end.
```

sub

<sub> = sub <identifier> '(' <subarguments> ')' begin <block> endsub

Subroutine starts with a keyword 'sub' followed by subroutine name. Arguments can be by-value or by-reference. By-value means that changes made in the subroutine are not propagated to the actual variable, by-reference means variable is referenced to actual variable and all changes affect the actual variable. To declare an argument by reference, start the declaration with keyword ref. Arguments are by default by-value. Element types must be declared by-reference. Core types, like String and Number can be declared by-reference or by-value.

```
sub setname(ref Package mypack, String name)
begin
  set mypack.name to name;
endsub
```

declare

<declare> = declare

Before the script contents there is an optional declare section. Here you can declare global variables or set script output file.

```
script testscript(String rootid)
declare
 set OUTPUT to "myoutput.out"
 var String myGlobalVar init "abc";
begin
end.
```

var

<var> = var <type> <identifier> <varinit>

Variable definitions are done using 'var' command. There exist two primitive types, String and Number, which are initialized with 'init <value>' statement.

```
var String name init "Mars";
var String name init planet.name;
var Number moons init 2;
```

A type can be also a meta-element. A meta-element type Element is a base class for other meta-elements and can be used polymorphically. Concrete meta-elements are e.g. Package, Class and Attribute. If a type is a meta-element type, 'init' must not be used, but 'load' or 'create'. Use 'load' to load the element from model. Use 'create in <packageid>' to create a new meta-element to the model under given package.

```
var Package rootpack load rootid;
var Class planet load "123232-234234-324234-43";
var Class moon create in rootid;
```

set

<set> = set <identifier> to <setexpression>

Command 'set' is used to alter the variables. Simple expressions are allowed. Strings support operator '+' to concatenate strings. Numbers support operators: '+', '-', '*' and '/'. Expressions are processed from left to right, parenthesis are not supported. A number value can be assigned to a string variable, but no string can be assigned to a number.

```
set name is planet.name;
set planets is "";
set planets is planets + ", " + planet.name;

set ratio to 100 * submass / totalmass;
set ratiostr is "Ratio: " + ratio;
```

remove

<remove> = remove <identifier>

Command 'remove' removes an element with the given identifier from the model. Note, if an element is removed, also all its owned elements are removed. E.g. if you remove a package, all elements under it are removed as well.

```
remove "123232-234234-324234-43";
remove planet.uuid;
```

for

<for> = for each <type> <identifier> in <list> do <block> endfor

Command 'for' is used to iterate through meta-element lists found in the model. The <type> must be one of the metaelement types (Element, Class, Package,...) If the type is Element all elements in the list are processed, on the other hand, if the type is e.g. Class, only the class elements are processed even if the list contains also other elements. The <list> must be a list taken from a metaelement using e.g. 'elementlist' property operation. To check if the current item is first or last item to process, constructs _iter.first and _iter.last can be used.

```
for each Element current in mpack.elementlist do
  if current is not _iter.last then
  endif
endfor
```

if

<if> = if <conditions> then <block> [else <block>] endif

This is a command for conditional branching. Condition 'is a' can be used to check polymorphically if an element is of given type. Condition 'is' tests the equality (this is same as 'is equal', but the 'equal' can be omitted). Condition 'is above' test if a number is above other number and condition 'is below' tests if it's below other number.

The conditions can be concatenated using using operators 'and' and or'. Expressions are resolved from left to right. Also negation operator 'not' is supported. An optional else block may follow the main block.

```
if current is a Package then
endif

if planet.name is "Jupiter" and processed is not "done" then
else
  if planet.name is "Mercury" or mass is above 5000
  endif
endif
```

downcast

<downcast> = downcast <identifier> to <type> <identifier>

This command can be used to downcast the base metaelement to a given metaelement type. E.g. if the list iterator is of type Element it can be downcasted to Package. For the resulting package, all operations allowed for a package can be used. Note however, that you must be sure that the element is really of the downcasted type, otherwise an error will be produced.

```
if current is a Package then
  downcast current to Package mpack;
endif
```

call

<call> = call <identifier>(<uuid>)

Call command can be used to invoke the script recursively. The <identifier> must be the script's name. If used, it must be guaranteed that no indefinite recursion occurs.

```
call myscript(mpack.uuid);
```

print

<print> = print <printexpression>|println <printexpression>

Print command can be used to print out the information in the model. There are two commands: 'print' and 'println'. They are otherwise identical, but println adds an end-of-line character after the expression. Operator + is supported to concatenate strings. Also variables of Number type can be put in print expressions; they are automatically converted to strings.

```
print planet.name;
println "Planet " + planet.name + " mass is " + totmass;
```

return

<return> = return

Command 'return' returns from a subroutine before the normal end in endsub.

```
sub launch(Number count)
begin
  if count is above 0 then
    return;
  endif
endsub
```

exit

<exit> = exit

Command 'exit' stops processing of the script.

Element Operations

The table below shows supported core element operations.

Element type	Modify	Operation	Explanation
Element	X	name	Name (plain name)
		uuid	Uuid identifier
		eltype	Element type (e.g. Package)
	X	documentation	Element's documentation
		packageid	Uuid of package which owns the element
		tostring	Converts element to UML format string
	X	visibility	Element's visibility (public, protected,..)
	X	multiplicity	Element's multiplicity
	X	isroot	Is root
	X	isleaf	Is leaf
	X	isabstract	Is abstract
	X	isactive	Is active
	X	isordered	Is ordered
	X	ispolymorphic	Is polymorphic
	X	isderived	Is derived
	X	isgenerate	Is generate (generate/analyze code for class)
	X	changeable	Changeable (frozen)
	X	ownerscope	Ownerscope (classifier, "static")
	X	targetscope	Targetscope
	X	concurrency	Concurrency (guarded)
	X	stereotype	Stereotype
	X	type	Type (e.g. attribute type)
	X	value	Value (e.g. attribute default value)
	X	extradata	Extra data (class extradata)
		packageid	Package uuid of element's owner package
		ownerid	Element's owner's uuid (may be class)
		isreadonly	Is readonly element
	X	codelang	Element's engineering language (C++, Java, C#, C)
	X	sourcefile	Element's engineering source file
	X	namespace	Namespace (C++; C#)
		innername	Element's name with owner class (Owner::Inner)
		qualifiedname	Element's name with package information
Package		elemcount	Number of elements owned by the package
		elementlist	Element list containing owned elements
Class		classlist	Class list containing inner classes
		parameterlist	List of class parameters (template parameters)
		attributelist	List of class attributes
		operationlist	List of class operations

Example of element operations can be seen below:

```
print  myclass.name + ":  " + myclass.docum;
```

Where operation `name' returns the element's name and operation `documentation' returns its documentation.

MetamillScript Example

```
-- this is an example which shows how to use MetamillScript commands.
-- file: calc.mils

script calc(String rootid)
declare
  set OUTPUTFILE to "xx.out";
  var Number level init 0;
  var Number total init 0;
  var Number totclass init 0;
begin

  var Element melem load rootid;
  if melem is not a Package then
    println "Root is not a package";
    exit;
  endif

  var Package mpack load rootid;

  for each Element current in mpack.elementlist do

    if current is a Package then
      downcast current to Package spack;
      set level to level + 1;
      call calc(spack.uuid);
      set level to level - 1;
    else

      set total to total + 1;
      if current is a Class then
        set totclass to totclass + 1;
      endif

    endif

  endfor

  if level is 1 then
    var Number ratio init 0;
    set ratio to 100 * totclass / total;

    println "Total number of elements  : " + total;
    println "Total number of classes   : " + totclass;

    println "Ratio class/element       : " + ratio + "%";
    println;
  endif

end.
```

MetamillScript Grammar (simplified)

```
<start> = <subroutines> <script>
<subroutines> = <sub> | <sub> <subroutines>
<script> = script <identifier> '('<arguments>')' <declare> begin <block> end.
<declare> = declare
<declarations> = <set>|<var>|<declarations>
<var> = var <type> <identifier> <varinit>
<set> = set <identifier> to <setexpression>
<remove> = remove <identifier>
<for> = for each <type> <identifier> in <list> do <block> endfor
<if> = if <conditions> then <block> endif

<downcast> = downcast <identifier> to <type> <identifier>
<call> = call <identifier>(<uuid>)
<print> = print <printexpression>|println <printexpression>
<is> = <conditem> <isexpr> <conditem>
<isa> = <identifier> <isexpr> a <type>
<exit> = exit
<sub> = sub <identifier> '(' <subarguments> ')' begin <block> endsub

<isexpr> = is|is not|is <isdetail>|is not <isdetail>
<isdetail> = <equal>|<above>|<below>
<varinit> = load <uuid>|init <value>|create in <identifier>

<conditions> = <condition>|<condition> <boolexpr> <conditions>
<condition> = <is>|<isa>
<conditem> = <identifier>|<type>|<value>
<boolexpr> = and|or
<block> = <expressions>
<commands> = <command>;|<command>;<commands>
<command> = <var>|<set>|<remove>|<for>|<if>|<downcast>|<call>|<print>
<setexpression> = <resolvable>|<resolvable> <setoperator> <setexpression>
<printexpression> = <resolvable>|<resolvable> + <printexpression>
<resolvable> = <identifier>|<value>

<setoperator> = +|-|*|/
<arguments> = <stringtype> rootid
<list> = elementlist
<type> = <stringtype>|<numbertype>|<elementtype>
<stringtype> = String
<numbertype> = Number
<elementtype> = Element|Package|Class|Interface|Attribute|Operation|Parameter|
            Actor|UseCase
<uuid> = uuidstring
<identifier> = anystring
<value> = anystring
```

12. Features and Support

Features Summary

Main Features

Feature name	Explanation
Use Case diagrams	Use Case diagrams for system requirements capture. Elements: actors, use cases, system boundaries.
Package diagrams	Package diagrams for software architecture design. Package diagram is a role name for class diagram. Elements: packages, interfaces, relationships.
Class diagrams	Class diagrams for static system design. Parameterized classes (templates) are supported. Elements: classes, interfaces, relationships.
Composite Structure diagrams	Composite Structure diagrams for static design of element's inner structure. Elements: classes, ports, parts
Object diagrams	Object diagrams for static system design. Elements: objects, links.
Communication diagrams	Communication diagrams for dynamic system design. Elements: objects, links, messages.
Sequence diagrams	Sequence diagrams for dynamic system design. Elements: objects, messages.
State Machine diagrams	State Machine diagrams for dynamic system design. Elements: states, transitions.
Activity diagrams	Activity diagrams for dynamic system design. Elements: action states, transitions, branches, joins, forks, swimlanes.
Component diagrams	Component diagrams for static physical system design. Elements: components, interfaces, relationships.
Deployment diagrams	Deployment diagrams for distributed physical design. Elements: components, nodes

Interaction Overview diagrams	Interaction Overview diagrams for high-level view of interaction relations. Elements: Interaction uses
Timing diagrams	Timing diagram for presenting changes in system's state on linear axis. Elements: State and value lifelines
Profile diagrams	Profile diagrams for describing metaclass extensions by stereotypes. Elements: stereotypes, metaclasses.
Extensibility Mechanisms	Stereotypes, tagged values and constraints for extending UML™. Supports profiles.
Export diagrams	Export diagrams to EMF, BMP, PNG and JPG. Also partial export is supported..
XMI Import	Import XMI files. XMI 2.1, 2.0
XMI Export	Export XMI files. XMI 2.1, 2.0
Rose .mdl file import	Import Rose .mdl - model files.
Rose .mdl file export	Export Rose .mdl - model files.
Workspaces	Save workspace i.e. current model and currently opened diagrams.
Code Generation C++	Generate C++ source code with code markers (keeping changes).
Code Generation Java™	Generate Java™ source code with code markers (keeping changes).
Code Generation C#	Generate C# source code with code markers (keeping changes).
Code Generation ANSI C	Generate ANSI C source code with code markers (keeping changes).
Code Generation VB.Net	Generate VB.Net source code with code markers (keeping changes).
Code Generation ADA	Generate ADA source code with code markers (keeping changes).
Code Generation Python	Generate Python source code with code markers (keeping changes).
Reverse Engineer Java™	Analyze Java™ source code to model.
Reverse Engineer C++	Analyze C++ source code to model.
Reverse Engineer C#	Analyze C# source code to model.
Reverse Engineer ANSI C	Analyze ANSI C source code to model.

Reverse Engineer VB.Net	Analyze VB.Net source code to model.
Reverse Engineer ADA	Analyze ADA source code to model.
Reverse Engineer Python	Analyze Python source code to model.
UML 2.4 Meta model system	Meta-element system based on UML 2.4
XMI 2.1 support	Generate and read XMI 2.1 files
HTML document generation	Generate HTML documentation about model.
RTF document generation	Generate RTF documentation about model. It can be opened with Word or other tool supporting RTF.
MetamillScript - scripting language	A scripting language for custom model access.

Miscellaneous Features

- parameterized classes (templates)
- tabbed dialogs for settings,
- mouse right click short-cuts,
- model tree for easy diagram access,
- copy/paste to clipboard,
- fit to page printing,
- auto-size elements,
- command undo/redo,
- element alignment
- diagram panning,
- element type color-settings,
- and many more..

Technical Support

There is a free technical on-line support for Metamill users. Before sending anything, please read the FAQ (frequently asked questions) at www.metamill.com. If that doesn't give answers, please go to Online Support. Explain with details the problem and submit your query. You can also send e-mail to: support@metamill.com

Don't forget to give your return e-mail address, otherwise no answer can be sent. Registered users have priority over the evaluation version users.

More up-to-date information about support is available on our web site at www.metamill.com.

Note: If you had problems purchasing the license on-line, please see web pages above for how to get help.

Appendix A

License agreement

You ("Licensee") should carefully read this license agreement before using this software. Your use of Metamill ("Software") indicates your acceptance of all terms and conditions of this license agreement.

IF LICENSEE DOES NOT ACCEPT THESE LICENSE TERMS, NO LICENSE IS GRANTED TO THE SOFTWARE, AND LICENSEE SHOULD DESTROY ANY OBTAINED COPIES OF THE SOFTWARE.

1. Grant of license

This License Agreement permits you to use the software product identified above. Each user of this software must be covered either individually, or as part of a group multi-user license. The evaluation version of the software can be used only for limited time and for evaluation purposes only.

2. Ownership

This agreement gives you only a right to use the software. The author of the software owns the software. This Agreement does not grant you any intellectual property rights in the Software.

3. Evaluation and Registration

This is not free software. Subject to the terms below, you are hereby licensed to use the evaluation version of the software for evaluation purposes without charge for a period of 30 days. If you use this software after the 30 day evaluation period a license fee is required. License can be bought on our web site at www.metamill.com. After payment you will be sent a license key which you can use to activate your copy of the software. Quantity discounts are available, as described in the web site mentioned above. Unregistered use of the software after the 30-day evaluation period is in violation of international laws.

4. Registered Version

One registered copy of the software may be used by a single person who uses the software personally on one or more computers. You may access the registered version of the software through a network, provided that you have obtained individual licenses for the software covering all users that will access the software through the network. For instance, if nine different users will access the software on the network, each user must have its own license, regardless of whether they use the software at different times or concurrently. Multi-user license can be installed for as many users as indicated in the license. For example, a multi-user license indicating maximum of 10 users can be installed for 10 users at maximum. Site license may be installed for any number of users in one site. Geographically separated sites need their own site licenses. To use team working features, a multi-license must be purchased.

5. University Version

Same limitations apply for university version as registered version with addition that the license will expire after specific number of months from purchase. The university license will be valid until the date seen in the About box of the software. The university version is available only for universities and other educational organizations and is meant only for educational use. Normal registered version must be purchased if the software is used in any commercial use.

6. Beta Version

If the software is identified as BETA it indicates it is experimental software and the same rights and restrictions apply to it as to evaluation software. Licensee acknowledges that Software identified as BETA is experimental and may have defects or deficiencies.

7. Distribution

IT IS STRICTLY PROHIBITED TO DISTRIBUTE THE REGISTERED VERSION OF THE SOFTWARE WITHOUT PRIOR EXPLICIT WRITTEN PERMISSION FROM THE AUTHORS.

Distributing the evaluation version is allowed under the following limitations. You are licensed to make as many copies of the evaluation version of this software as you wish; give exact copies of the original evaluation version to anyone; and distribute the evaluation version of the software and documentation in its unmodified form. It is specifically prohibited to distribute license keys that may be used for registration with the evaluation version of the software.

8. Modifying

You may not alter, modify, translate, reverse-engineer, disassemble or in any other ways attempt to discover the source code of the software. Also you cannot give anyone else permission to alter the software.

9. Governing Law

This agreement shall be governed by the laws of the Grand Duchy of Luxembourg.

10. Disclaimer of Warranty

THIS SOFTWARE AND THE ACCOMPANYING FILES ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY OR ANY OTHER WARRANTIES WHETHER EXPRESSED OR IMPLIED. IN NO EVENT WILL THE AUTHORS OF THE SOFTWARE, OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, EVEN IF A REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY THIRD PARTY. NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. ANY LIABILITY OF THE SELLER WILL BE LIMITED EXCLUSIVELY TO PRODUCT REPLACEMENT OR REFUND OF PURCHASE PRICE.

Metamill User’s Guide