Cookbook for Developers of ArgoUML

An introduction to Developing ArgoUML

Edited by Linus Tolke
Markus Klink

Cookbook for Developers of ArgoUML: An introduction to Developing ArgoUML

by Linus Tolke and Markus Klink

Abstract

The purpose of this Cookbook is to help in coordinating and documenting the development of ArgoUML.

This version of the cookbook is loosely connected to the version 0.24 of ArgoUML.

Table of Contents

Change Log

This will also be a log of major design decisions. A major design decision is a decision that changes responsibilities or functions of the subsystems.

Table 1. Changes done

When	What	Who
2007-01-18	Added instructions on how to configure Subversion. (See Section 2.2, “Preparations”).	Linus Tolke
2007-01-07	Change to the process of verifying issues to allow the reporter to more freedom. (See Section 9.3, “Roles Of The Workers”, Section 9.5, “How to verify an Issue that is FIXED”, Section 9.6, “How to verify an Issue that is rejected”, and Section 9.7, “How to Close an Issue”).	Linus Tolke
2006-12-30	Change how we handle problems in the JRE and other SW not delivered with ArgoUML. (See Section 9.8, “How to relate issues to problems in dependencies”).	Linus Tolke
2006-10-07	Removed mentions of CVS.	Linus Tolke
2006-10-01	Change the description on how to check out and build with Eclipse to fit the Subversion set up (and Eclipse 3.2). (See Section 2.8, “Setting up Eclipse 3”).	Linus Tolke
2006-09-30	Change to describe the new release build mechanism based on subversion (See Section 2.11, “Making a release”).	Linus Tolke
2006-09-12	Removed the manual tests. Removed the description on how to publish the documentation. Updated the download and build section for Subversion. (See Chapter 2, *Building from source*).	Linus Tolke
2006-07-14	Added a section on how to run all JUnit test cases from within Eclipse. (See Section 2.8.5.1, “Running all JUnit test cases from within Eclipse”).	Linus Tolke
2006-06-24	Added explanation on different kinds of releases and how we work with them. (See Section 9.9, “How to Create Stable Release”).	Linus Tolke
2006-06-19	Added a description on how to run JUnit tests from within Eclipse. (See Section 2.8.5, “Running JUnit test cases from within Eclipse”).	Linus Tolke
2006-05-28	Renamed subprojects to dependencies.	Linus Tolke
2006-05-14	Split and moved the CVS chapter. One part goes into the Standards for coding chapter and one part goes into a newly created appendix. (See Section 7.2, “When Committing to the Repository”, and Appendix B, *Repository Contents*).	Linus Tolke
2006-05-14	Moved the Terminology into the Cookbook. (See Terminology in the ArgoUML project).	Linus Tolke
2006-05-13	Change to the introduction to the ArgoUML project to reflect that there are several Tigris projects involved. (See Chapter 1, *Introduction*).	Linus Tolke
2006-05-13	Moved the User Manual section from chapter 7 to chapter 10. (See Section 8.7, “User Manual Plans”). Moved the rest of chapter 7 to appendix.	Linus Tolke
2005-05-01	Removed old description on how to make releases. Added instructions on how to run the installers.	Linus Tolke

When	What	Who
2006-04-30	Change to the explanation on how to use Eclipse. (See Section 2.8, “Setting up Eclipse 3”).	Linus Tolke
2005-10-29	Change to the instructions on how to build ArgoUML to describe how it works with the argouml-mdr project. (See Chapter 2, *Building from source* and Section 2.8.1, “Checking out through Eclipse”).	Linus Tolke
2005-07-22	Removed the /modules/junit. (See Section 2.11, “Making a release”).	Linus Tolke
2005-07-19	Change to the descriptions of the Model, Diagrams, and Persistence subsystems (See Section 5.1, “Model”, Section 5.3, “Diagrams”, and Section 5.5, “Persistence”).  **Design decision - Bob Tarling 2005-07-14** The Diagrams subsystem does not store any data. All data it works on is stored in the Model subsystem.	Linus Tolke
2005-07-18	Change to the short list of subsystems and responsibilities. (See Chapter 5, *Inside the subsystems*).	Linus Tolke
2005-06-18	Restructured: all main chapters are now in seperate files. No content changes.	Michiel van der Wulp
2005-06-15	Change to how internationalization is done. Subprojects. (See Section 5.14, “Internationalization”).	Linus Tolke
2005-06-12	Change to the description on how to set up an Eclipse environment. (See Section 2.8.1, “Checking out through Eclipse”).	Linus Tolke
2005-06-11	Change to how to make an announcement. (See Section 2.11, “Making a release”).	Linus Tolke
2005-05-23	Change to release building description. (See Section 2.11, “Making a release”).	Linus Tolke
2005-05-06	Added instructions on how we work with sub-projects. (See Section 2.5.2.2, “Working in a subproject”).	Linus Tolke
2005-05-01	Change to the tools for releases. (See Section 2.11, “Making a release”).	Linus Tolke
2005-04-29	Added a diagram explaining Explorer. (See Section 5.18.3, “Details of the Explorer Implementation”).	Michiel van der Wulp
2005-03-10	Change to process for verifying issues. Any release after the one where the issue is fixed can be used for verifications. (See Section 9.5, “How to verify an Issue that is FIXED”).	Linus Tolke
2005-03-06	Change to description of how to build. `src` directory is now involved. (See Chapter 2, *Building from source*).	Linus Tolke
2005-03-01	Removed the modules component. (See 6).	Linus Tolke
2005-02-01	Change to Model subsystem. (See Section 5.1, “Model” [39], Section 5.3, “Diagrams”). Added the Persistence subsystem. More work is needed. (See Section 5.5, “Persistence”).	Linus Tolke
2005-01-30	Change to the description on how to use the Model subsystem, the ModelFacade does not exist anymore. (See Section 5.1.6, “How to work against the model”).	Linus Tolke
2005-01-29	Change to Model subsystem chapter. Removed references to UmlEventPump and clearified how to remove elements using the UmlFactory. (See Section 5.1.3.2.1, “ How do I register a listener for a certain type event ” and Section 5.1.6, “How to work against the	Linus Tolke

When	What	Who
	model”).	
2005-01-26	Change to requirement of JDK version support. (See Section 3.3.1, “Choice of JRE: ArgoUML will support any JRE compatible with a Sun specification of any JRE from Sun that has not begun the Sun End of Life (EOL) process. ”).	Linus Tolke
2005-01-07	Added copyright notices to the files.	Linus Tolke
2004-12-30	Change the default year in the copyright notices. Yes, I am a little early. (See Chapter 7, *Standards for coding in ArgoUML* and Section 2.8, “Setting up Eclipse 3”).	Linus Tolke
2004-11-01	Change to the way we generate documentation. The FILENAME.id files are no longer used. (See Section 2.7, “Generating documentation”).	Linus Tolke
2004-10-29	Change to the description on how to generate documentation. Better explanation of how it works. (See Section 2.7, “Generating documentation”).	Linus Tolke
2004-10-19	Change to How to Contribute. Changed some spelling errors in cookbook.in while at it. (See Section 1.3, “How to contribute”).	Linus Tolke
2004-10-11	Changes to description of module loader making the new module loader a fact. (See Section 5.19, “Module loader” and Section 6.2, “Modules and PlugIns”).  **Design decision - Linus Tolke 2004-10-11** New imperative Module loader.	Linus Tolke
2004-09-17	Change to the description on how to extend ArgoUML. Now module loader described. (See Chapter 6, *Extending ArgoUML*).	Linus Tolke
2004-09-16	Changed the meaning of RESOLVED/LATER. (See 6 f and g in Section 2.11, “Making a release”, and Section 9.2.2, “Resolutions”).	Linus Tolke
2004-09-15	Change to design of new module loader. (See Section 5.19.2, “Design of the new Module Loader”).	Linus Tolke
2004-08-17	Deemphasized the layers and instead describe the subsystems in groups according to the MVC-pattern. (See Chapter 4, *ArgoUML Design, The Big Picture* and Chapter 5, *Inside the subsystems*).	Linus Tolke
2004-08-17	Change to the definition of the priorities. Now they are defined in terms of how much release blocker they are. (See Section 9.2.1, “Priorities”).	Linus Tolke
2004-08-02	Added rationale for not using RESOLVED/REMIND or RESOLVED/LATER (See Section 9.2.2, “Resolutions”).	Linus Tolke
2004-07-28	Reorganized the description on how to use Eclipse 3. Added instructions on how to use the Eclipse JUnit test runner. (See Section 2.8, “Setting up Eclipse 3”).	Linus Tolke
2004-07-25	Added this Change Log. (See Change Log).	Linus Tolke

Chapter 1. Introduction

1.1. Thanks

We, the authors, would like to take the opportunity to thank everyone involved in the creation of this documentation, and especially the people behind setting up the DocBook environment. In particular thanks go out to Alejandro Ramirez, Phillipe Vanpeperstraete and Andreas Rueckert. Thank you!

1.2. About the project

ArgoUML is an open source project, so it depends on people that volunteer to work on it. Especially in the area of development there is still so much to do!

This Cookbook is dedicated to everyone interested in taking part in the ArgoUML project as such and should help to transfer the knowledge from the old experts to them. Please feel free to discuss the ArgoUML project and this Cookbook on dev mailing list [mailto:dev@argouml.tigris.org]!

The ArgoUML project is hosted at Tigris. For best use of the Tigris platform, the ArgoUML project is spread over several Tigris projects with the argouml project as the top project. Each Tigris project has its own list of developers, source repository, issuezilla, web site, set of mailing lists, ... but they are released together.

1.3. How to contribute

You can help, there are big tasks and small tasks waiting for you.

Here is a suggestion on how you could become part of the ArgoUML Project. This could be perceived as a ladder to climb but remember that if so it is firstly a ladder of levels of commitment and time spent by you. You get no prize for climbing higher, you just get more responsibility in the project.

1. Use ArgoUML.

2. Report bugs & suggest enhancements.

   There are bugs in ArgoUML. When you use ArgoUML you might encounter them where you least expect it. To help, make sure they are known about i.e. that there exists an issue in Issuezilla describing the problem. You need to be a registered user at Tigris to report bugs but notice that to add further comments to the issue you also need to have gotten a Role in the ArgoUML project.

   Like all good open-source projects, ArgoUML is evolving to meet the needs of its users. We can only do this if we know what users need, so please file enhancement requests.

   To help the developers prioritise work on bugs and enhancements, vote for the ones that you really care about.

   You need to be a registered user at Tigris to report bugs and enhancements (and to vote for them) but notice that to add further comments to the issue you also need to have gotten a Role in the ArgoUML project.

3. Subscribe to some of the users' mailing lists.

   Discuss how you use ArgoUML in your project and how you promote ArgoUML in your organization. You can also help other users with their ArgoUML-related problems.

There is one users' mailing list for each language project (Spanish, Swedish, Chinese, ...) for the users that prefer to discuss ArgoUML using that language. These are not currently (May 2006) very active but the main list, the English-speaking list at users@argouml.tigris.org-list, is.

In some cases there are also users' mailing lists for the specific subproject, for discussing these specific features or uses of them (cpp, andromda, ...). Prefer those if your discussion is for that area.

4. Apply for an Observer role.

   This shows that you are committed to the project and also allows you to comment on issues.

5. Familiarize yourself with the project and how we work.

   Suggestion on how to go about this:

   a. Read through most of the User manual and install and run the latest version of ArgoUML.

   b. Subscribe to the issues list.

      You will get updates on all issues so you can monitor what we are doing in the project. (It could be a lot of mails. If it turns out you don't like watching issues in this way, you unsubscribe!)

   c. Subscribe to the commits mailing list of the project that you are interested in.

      You will get updates on all changes that are done to code, documentation, and the web site. (It could be a lot of large mails. If it turns out you don't like watching what is going on in the project in this way you unsubscribe.)

   d. Read the process part of the Developers Cookbook at Chapter 9, *Processes for the ArgoUML project*.

      This will give you the idea of how the ArgoUML project attempts to release with good quality and especially how we use Issuezilla.

   e. Get the Observer role granted.

      You can apply for the Observer role in any or all of the argouml projects if you like. Choose the ones you are interested in.

      From this on you can comment on bugs yourself directly in Issuezilla.

      You can also verify issues according to the verification process (see Section 9.5, “How to verify an Issue that is FIXED”).

      This will help you understand the terminology used in the project and also gives you an idea of the current quality of ArgoUML and what needs to be done in the future.

      This is also a very low-commitment level task that could be completed in a couple of minutes (depending on your choice of issue).

   f. Read the rest of the Developers Cookbook.

      There is a lot of stuff discussed in here that is interesting for your understanding of the project and the code.

   g. Check out the source from subversion and build.

6. Subscribe to some of the dev mailing lists.

There is one dev mailing list for each project with the purpose of discussing the specifics for that project. Subscribe to as many of them as you'd like and take part.

The purpose of this is to see what the developers are discussing in the project.

Monitor the discussions and as soon as you see something discussed where you have an opinion, jump right in!

7. Familiarize yourself with the code.

   For this a good knowledge of Java is more or less a prerequisite.

   Suggestion on how to go about this:

   a. Take active part in the discussions on the dev-list.

   b. Solve issues registered in Issuezilla or fix other things that you want to pursue.

   c. Convince someone to commit your changes.

      Establish a relationship with one developer of the project you want to work with. That developer will check that your code reaches the quality level that we strive for in the project and obeys the design.

   d. Repeat.

      This can go on until the developer helping you knows that you have good knowledge of the project quality and design and the main problem for you two is that sending, waiting, committing, updating et.c. is extra work.

8. Apply for a Developer role in the project where you want to contribute.

   This allows you to do commits on your own and you can now increase the pace in which you are working while also increasing your responsibilities in the project.

   This role is granted by the Project leader after he is convinced that you have learned the enough about the project w.r.t.:

   - Understanding and accepting the goals.
   - Understanding where we are in the development process.
   - Understanding the terminology used in the project.
   - Understanding how we use subversion in the project.
   - Understanding the set of tools (ant, JUnit) and how to use them.

9. Focus your work in a specific area.

   Everybody has different interests and the best contribution is made when someone is allowed to pursue his own interests. Hopefully ArgoUML provides you with interesting challenges to your taste.

10. Accept responsibility for a specific area.

1.4. About this Cookbook

This document, the Cookbook for Developers of ArgoUML, is provided with the hopes of being helpful for the developers of ArgoUML when it comes to learning and understanding how ArgoUML work in order to improve on its functions and features. It can also be of interest for persons that wish to analyze the ArgoUML project for whatever purpose that may be.

1.4.1. In this Cookbook, you will find...

Information on how to compile ArgoUML. (Chapter 2, *Building from source*)

Information on how different features of ArgoUML are implemented and how they are to be used. (Chapter 4, *ArgoUML Design, The Big Picture* and Chapter 5, *Inside the subsystems*)

Information on how you should write extensions to ArgoUML. (Chapter 6, *Extending ArgoUML*)

Information that you, as a developer of ArgoUML, need to know about how to contribute. (Chapter 7, *Standards for coding in ArgoUML*, Chapter 8, *Writing Documentation in the ArgoUML Project* and Chapter 9, *Processes for the ArgoUML project*)

1.4.2. In this Cookbook, you will not find...

You will not find information on how to install and use ArgoUML.

You will not find information on what UML is and if or how you should use it in your project.

You will not find information on how to convince your project to use ArgoUML as a modeling tool.

1.5. Mailing Lists

All developers *MUST* subscribe to the mailing list for developers. Please find the details at: http://argouml.tigris.org/servlets/ProjectMailingListList
[http://argouml.tigris.org/servlets/ProjectMailingListList]

It is also recommended to join the commits and issues mailing lists. Both give you a good idea of what is going on. Developers should also work with Issuezilla registering or fixing problems found by themselves and others.

Chapter 2. Building from source

Building ArgoUML from source requires a SVN client, a current JDK (1.4 or later), and 350MB of free disk space. All other tools, including the Ant build tool upon which the build is based, are included in the project source tree. If you have these tools and are familiar with them, the next section contains quick instructions to build ArgoUML from source. For more detailed directions, see the following sections.

2.1. Quick Start

If you are using Eclipse 3.1 or later, see Section 2.8, “Setting up Eclipse 3” for quick setup instructions.

If you are using Windows, the follow commands will build ArgoUML from source and run it. If you using Unix/Linux, the comparable commands, modified slightly for your particular shell, should work.

```
C:\Work>svn checkout http://argouml.tigris.org/svn/argouml/trunk/src http://argoum
Password:  (give empty password if prompted)
C:\Work>set JAVA_HOME=C:\Programs\jdkwhatever
C:\Work>cd argouml\src_new
C:\Work\argouml\src_new>build run
```

A newly compiled ArgoUML will open in a new window.

Note

JDK 1.4 or later is required

That was the compact version for Windows + JDK. Modifying these steps slightly as appropriate for your shell should work on Unix/Linux systems as well.

If you don't understand these instructions or they don't work, please read the rest of the chapter for more detailed instructions on how to build ArgoUML.

2.2. Preparations

In order to develop the ArgoUML source it is absolutely mandatory that you work with ArgoUML from the source repository. How you checkout this is described in Download from the Subversion repository.

Notice that the source repository contains not only a set of source files but a complete development environment with required tools for working with ArgoUML.

2.2.1. What do I need to build ArgoUML?

These are the tools not included in the repository that you need to work with ArgoUML.

- A computer with an Internet connection and free disk space for your work.

  Around 150MB to download everything from the repository. (Currently September 2006 it is 147MB.) Around 200MB to download all and build the tool and the documentation. (Currently September 2006 it is 192MB.) 350MB should be enough to build it all (Javadocs, documentation, classes, ...). (Currently September 2006 it is 299MB.)

- Subversion for getting the files and committing source code updates. You can also use an IDE with a built-in subversion client.
- JDK, at least version 1.4.2 (includes the Java compiler)

For building the documentation from DocBook format, you also need the following tools:

- DocBook XSL style sheets.

  There exists rules in the `argouml/documentation/build.xml` for downloading this correctly.

- Jimi

  Used by FOP for including PNG pictures.

  Detailed instructions:

  1. Download the file `jimi1_0.zip` from java.sun.com [http://java.sun.com/products/jimi/].
  2. Extract the file `JimiProClasses.zip`. Most unzip applications allow you to specify the output location. If you are using one of these, this step can be combined with the next.
  3. If the previous step put the extracted file elsewhere, copy or move this file into the `argouml\tools\lib` directory.

2.2.2. Configuring Subversion

To reduce problems with line endings and to get the headers working, we use the properties on files in the Subversion repository according to the table below.

Table 2.1. Changes done

Property	Value	Comment
svn:keywords	Author Date Id Revision	All text files, Java files, XML files. This is for the Id-tag at the top of all source files and the $Date$ tag on some web pages.
svn:eol-style	native	All text files, Java files, XML files, i.e. almost all non-binary files.
svn:executable	*	This is for the tools of the development environment to work properly.

The properties described here are project conventions and should be applied to the files in the repository. This is normally only needed when creating new files because the existing files should have their properties set correctly..

Alas, Subversion and the Tigris set-up does not allow us in the project to enforce this from the Subversion server end. Instead we rely on each developer and the subversion client installation on each developer's machine to perform this correctly.

To get your subversion client to help you in this, make sure the configuration in your subversion client config file contains settings for this. Your subversion client config file is

- On Windows: `%APPDATA%\Subversion\config` where %APPDATA% is `C:\Documents and Settings\`*`your username`*`\Application Data`.
- On Unix/Linux: `$HOME/.subversion/config`.

Suggested settings:

```
[miscellany]
enable-auto-props = yes

[auto-props]
### The format of the entries is:
###   file-name-pattern = propname[=value][;propname[=value]...]
### The file-name-pattern can contain wildcards (such as '*' and
### '?').  All entries which match will be applied to the file.
### Note that auto-props functionality must be enabled, which
### is typically done by setting the 'enable-auto-props' option.
*.java = svn:eol-style=native; svn:keywords=Id Author Date Revision
*.properties = svn:eol-style=native
*.sh = svn:eol-style=native;svn:executable; svn:keywords=Id Author Date Revision
*.txt = svn:eol-style=native; svn:keywords=Id Author Date Revision
*.xml = svn:eol-style=native
*.zargo = svn:needs-lock=*

# Picture formats
*.eps = svn:needs-lock=*
*.jpg = svn:mime-type=image/jpeg; svn:needs-lock=*
*.png = svn:mime-type=image/png; svn:needs-lock=*
*.gif = svn:mime-type=image/gif; svn:needs-lock=*

*.pdf = svn:mime-type=application/octet-stream; svn:needs-lock=*
*.PDF = svn:mime-type=application/octet-stream; svn:needs-lock=*
```

2.3. The ArgoUML development environment

2.3.1. Which tools are part of the ArgoUML development environment?

These tools are provided by the development environment that you get when you check out from the repository.

- Ant, the tool to manage compiling and packaging.
- mdrant, the integration allowing to run mdr from ant.
- ANTLR, for regenerating the built-in parser.
- JUnit, for running the JUnit test cases.
- JDepend, for examining the code.

For building the documentation from DocBook format, these tools are also provided with the development environment that you get when you check out from the repository.

- Saxon for building documentation from DocBook format.
- fop for generating PDF versions of the DocBook format.

  To build a PDF file with the pictures included you need Jimi that is downloaded separately. See Jimi.

2.3.2. What libraries are needed and used by ArgoUML?

These libraries are provided in the development environment that you get when you check out from the repository. They are checked by the Java compiler when compiling, needed for running ArgoUML and therefore distributed with ArgoUML.

- MDR, the Netbeans Model Data Repository.

  This library provides services to manipulate both the UML metamodel and the users model. It includes services to serial/deserialize to/from XMI.
- GEF graph editing framework, available from gef.tigris.org [http://gef.tigris.org].

  If you would like the GEF sources for reference, please consult their web site for directions on checking them out.
- The OCL package to parse and run the Object Constraint Language things.

  Details about the package are available from SourceForge OCL Compiler [http://dresden-ocl.sourceforge.net/].
- log4j, a library with infrastructure for logs.
- antlrall, the run-time part of the ANTLR tool.
- MDR, the Netbeans Meta Data Repository.

  This is included in the files jmi.jar, jmiutils.jar, mdrapi.jar, mof.jar, nbmdr.jar, openide-util.jar.

2.4. Download from the Subversion repository

The easiest thing to do now, is just check out the whole *`argouml`* project. For this use the Subversion URL http://argouml.tigris.org/svn/argouml/trunk.

If you want to limit the download size, by not checking out the images in the `argouml/documentation` and `argouml/www` the repository directories you need to check out to work with ArgoUML are `argouml/lib`, `argouml/tools`, `argouml/src_new`, `argouml/src`, and `argouml/tests`. These are located at the subversion URLs `http://argouml.tigris.org/svn/argouml/trunk/lib`, `http://argouml.tigris.org/svn/argouml/trunk/tools`, `http://argouml.tigris.org/svn/argouml/trunk/src_new`, `http://argouml.tigris.org/svn/argouml/trunk/src`, and `http://argouml.tigris.org/svn/argouml/trunk/tests` respectively.

If you just want to build the documentation you check out the directories `argouml/lib`, `argouml/tools` and `argouml/documentation`. These are located at the subversion URLs `http://argouml.tigris.org/svn/argouml/trunk/lib`, `ht-`

`tp://argouml.tigris.org/svn/argouml/trunk/tools` and `http://argouml.tigris.org/svn/argouml/trunk/documentation` respectively

If you want to work with the web site you check out the directory *`argouml/www`*. This is located at the subversion URLs `http://argouml.tigris.org/svn/argouml/trunk/www`,

If you don't want to acquire a Tigris login to do this you can use the "guest" account with an empty password.

2.5. Build Process

The standard ArgoUML build process is driven by Apache Ant, and it is highly recommend that you stick to that. Some developers use the integrated build tools of Eclipse and NetBeans, but always make sure that your work compiles with the standard Ant build process. There are also some Java files generated by magic scripts in Ant that you need to create before opening with the IDE.

Ant is a tool written in Java developed for Apache that reads an XML-file with rules telling what to compile to what result and what files to include in what jar-file.

The rule file is named `build.xml`. There is one of those in every separate build directory (`argouml/src_new`, `argouml/src/`*`whatever`*, `argouml/documentation`, and `argouml/modules/`*`whatever`*).

2.5.1. How Ant is run from the ArgoUML development environment

For your convenience the ant tool of the correct version is present in the source repository of ArgoUML in the file `argouml/tools/ant-1.6.2/lib/ant.jar`.

It is possible to start ant with the command **../tools/ant-1.6.2/bin/ant *`arg`*** and in the modules **../../tools/ant-1.6.2/bin/ant *`arg`*** . On windows the command **..\tools\ant-1.6.2\bin\ant *`arg`*** runs the program `ant.bat`.

To keep you from having to write this and keeping track if you are working with a module or not there are two scripts (one for Unix and one for Windows) that are called `build.sh` and `build.bat` respectively present in most of the directories that contain a `build.xml` file. These two scripts run the equivalence of the above paths.

By setting JAVA_HOME to different values you can at different times build with different versions of JDK and Java.

To use different versions of Ant, you are responsible for installing your own version. Also, you must execute **/where/ever/you/placed/your/new/ant *`target`*** rather than **build *`target`*** .

2.5.1.1. Compiling for Unix

Here is what you need to do in order to compile and run your checked out copy of ArgoUML under Unix.

1. **JAVA_HOME=*`/where/you/have/installed/jdk`***

   export JAVA_HOME

   This is for sh-style shells like sh, ksh, zsh and bash. If you use csh-style shells like csh and tcsh you will instead have to write **setenv JAVA_HOME */`where/you/have/installed/jdk`*** .

2. Change the current directory to the directory you are building

   cd *`/your/checked/out/copy/of/argouml/src_new`*

3. Start Ant with no parameters to get a list of build targets with descriptions

   ./build.sh

4. Run ArgoUML using **./build.sh run**

If you change something, running the run target again will build and run.

In certain cases when something is changed in the argouml project you need to use the clean target to re-compile everything, since we have not set up dependencies correctly.

2.5.1.2. Compiling for Windows

If you do this from Cygwin you work just like for Unix.

1. **set JAVA_HOME=*`\where\you\have\installed\jdk`***

2. Change the current directory to the directory you are building

   cd *`\your\checked\out\copy\of\argouml\src_new`*

3. Start Ant with "-p" parameter to get a list of build targets with descriptions

   build -p

4. Run ArgoUML using **build run**

2.5.1.3. Customizing and configuring your build

It is possible to customize your compilation of ArgoUML.

If you issue the command **build list-property-files** you can see what files are searched for properties.

Don't change the `argouml/src_new/default.properties` file (unless you are working with updating the development environment itself). Instead create one of the other files locally on you machine. The properties in these files have precedence over the properties in `argouml/src_new/default.properties`.

Remember that if you do this, you have modified your development environment. To be sure that you will not break anything for anyone else when checking in things developed using this modified environment, remove these files temporarily for the compiling and testing you do just before you commit.

2.5.1.4. Building Javadoc

By running Ant again using **build prepare-docs** the Javadoc documentation is generated and put into `argouml/build/javadocs`.

2.5.1.5. Building one of the modules

If you want to run ArgoUML with modules enabled the `build.xml`s are set up to do this in two ways:

1. Test just one module
   a. Build ArgoUML, the package

      This is done with **ant package** in the `argouml/src_new`-directory.

   b. Run the module

      This is done with **ant run**-command in the `argouml/modules/`*`whatever`* -directory.
2. Test several modules together
   a. Build ArgoUML, the package

      This is done with **ant package** in the `argouml/src_new`-directory.

   b. Compile and install the modules

      This is done with **ant install**-command in each of the `argouml/modules/`*`whatever`* -directories.

   c. Start ArgoUML

      This is done with **ant run** in the `argouml/src_new`-directory.

   This will start ArgoUML with all modules available.

2.5.2. Developing in a subproject

This describes how to do development in one of the ArgoUML sub-projects.

If you are in a hurry:

```
C:\Work>mkdir argouml
C:\Work>mkdir argouml\build
```

Download and unpack the latest release of ArgoUML into this directory.

```
C:\Work>svn checkout http://argouml-XX.tigris.org/svn/argouml-XX/trunk argouml-XX
C:\Work>set JAVA_HOME=C:\Programs\jdkwhatever
C:\Work>cd argouml-XX
C:\Work\argouml-XX>ant run
```

An ArgoUML starts with the module from the subproject argouml-*XX* enabled.

That was the short version provided that, you are using Windows + JDK, you have ant installed, and the subproject in question does not require any of the ArgoUML tools to build.

If you don't understand this or it doesn't work read the rest of the chapter that describes why and how in more detail.

2.5.2.1. The sub-project's relation to ArgoUML

The purpose of a subproject to ArgoUML is to develop things that are run within ArgoUML. In ArgoUML we call them modules, in other tools they are called add-ins or plug-ins.

If you want to start working with a module of your own you could do it by letting the ArgoUML project leader set up a subproject to ArgoUML for you. The benefits are:

- You will inherit all the infrastructure from the ArgoUML project.

  This includes a site for your Subversion repository, mailing lists, web server..., a common way to set up the project, releases, bug fixes, static checks, and coding guidelines and license.

- You get a community of ArgoUML developers that might monitor your work and fix problems, especially problems caused by changes that the developer makes to the ArgoUML API.

The draw-backs are:

- You are forced to use the ArgoUML infrastructure

  Subversion, BSD license, coding guidelines.

- You are forced to make your module Open Source.

  This is actually a Tigris policy.

If you decide not to make your module a argouml subproject, you can still benefit from using a similar set up as described here but since you have your module repository elsewhere, some adaptations are necessary.

The sub-projects are developed close to the ArgoUML project and reside in a similarly-looking subversion repository. We try to provide a working set of tools and instructions to fit the whole set of projects. These tools are sometimes located in the argouml project and sometimes in the subproject. Also, to compile the module, you need the argouml interfaces, and to run it you need argouml in place. In most cases the argouml interfaces is argouml itself so this distinction is mostly formal.

There are two ways to get the argouml in place w.r.t. your module. The ArgoUML source way and the quicker ArgoUML distribution way.

Using the ArgoUML source way you check out the argouml project alongside the subproject you are going to work with and build it. If you are doing development in the argouml project too, if the subproject in question requires a tool from the argouml project, or if your modules is on the bleeding edge of argouml development and you can't wait for distributions, this is the preferred way. You will need to update and rebuild the argouml project regularly.

Using the ArgoUML distribution way, you check out only your module and then download the ArgoUML distribution and work against that. This is the approach described in the beginning of this section. You will need to download and replace the ArgoUML distribution whenever you need a newer version to work against. You could also, at any point, upgrade to the ArgoUML source way to get to the bleeding edge.

The build.xml ant configuration file in the subproject and the argouml main project are set up to allow for both of these ways.

2.5.2.2. Working in a subproject

Each subproject has its own web site with documentation and plans of the subproject.

The subproject has its own commits mailing list that you need to join to monitor the commits. It also has its own dev mailing list where the people working within that subproject discusses the subproject. Join both of these mailing list to see what is going on in the subproject!

The sub-projects could use their own Issuezilla database but could also be subcomponents in the ArgoUML Issuezilla. If it is a subcomponent you need to acquire an Observer role in the argouml project to work in a subproject.

2.5.2.3. Targets in build.xml in a subproject

The following targets have the same documented meaning in all sub-projects:

- clean - optional

  Removes files that are generated by running any of the other targets.

- compile - optional

  Compiles the code. The result is in `build/classes`.

- generate - optional

  This is a step that, if it exists, can be run before compile. The result of this is some files that is a pre-requisite for compile so the compile target runs this automatically.

- install

  This builds the whole module and copies it into the `ext` directory in the argouml installation.

  The purpose of the ext directory is so that argouml can be started with several different modules started at once.

- jar - optional

  This builds the whole module and puts the resulting jar file(s) in `build`.

- run

  This starts argouml with this module active.

  This is the way to start this module with the newly compiled source.

- tests - optional

  This runs all the JUnit test cases available in the module. This probably requires the `junit.jar` tool from the argouml project.

2.5.3. Troubleshooting the development build

2.5.3.1. Compiling failed. Any suggestions?

It might be that some other developer has made a mistake in checking in things that contain errors, or forgotten to check in some files in a change. Look at the last couple of hours on the developers mailing list [http://argouml.tigris.org/servlets/BrowseList?listName=dev]! It is probably on fire.

Another reason for problems is an unclean local source tree. This means that if you have updated different parts of your source tree at different times it might contain inconsistencies. If you suspect this, first try to fix it by doing **build clean** and **svn update** before trying to build again. If that doesn't work remove your checked out copy completely and get it all again.

Another reason might be that you have an `build.properties` or `argouml.build.properties` file that you have been working with earlier and that is doing something. If in doubt, remove those files.

If nothing helps, ask the developers mailing list [mailto:dev@argouml.tigris.org]!

2.5.3.2. Can't commit my changes?

You need to have a developer role in the ArgoUML project or in the subproject you are working with.. If you don't then you cannot do commits yourself. Discuss what you have done and how best to test it on the ArgoUML project developers mailing list or the developers' mailing list for your subproject. Get in contact with the active developers and urge them to commit it for you.

Furthermore the checkout of your copy needs to be done with your Tigris id that has the Developer role. If you for some reason have earlier checked out a copy as guest and then made modifications you may have problems. It is unknown if this is a problem in Subversion. Let the editor know if you find out.

2.6. The JUnit test cases

ArgoUML has a set of automatic test cases using JUnit-framework for testing the insides of the code. The purpose of these are to help in pin-pointing problems with code changes before even starting ArgoUML.

The JUnit test cases are residing in a separate directory and run from ant targets in the `src_new/build.xml`. They are never distributed with ArgoUML but merely a tool for developers.

By running the command **build tests guitests** in `src_new` these test cases are started, each in their own JVM.

Each test case writes its result on the Ant log.

The result is also generated into a set of files that can be found at `build/test/reports/junit/output/html/index.html`.

The test cases' Java source code is located under `argouml/tests/org/argouml`.

2.6.1. How to write a test case

Now this will make all you Java enthusiasts go nuts! We have both class names and method names with a special syntax.

2.6.1.1. About the Test case Class

The name of the test case class starts with "Test" (i.e. Capital T, then small e, s and t) or "GUITest" (i.e. Capital G, U, I, T then small e, s, t). The reason for this is that the special targets in `src_new/build.xml` search for test case classes with these names. If you write a test case class that does not comply to this rule, you still can run the test cases in this class manually by starting with **build run-with-test-panel**, but it wont be known and run by other developers and automatic build mechanisms so don't do it.

Test case classes that don't require GUI components in place have filenames like `Test*.java`. They must be able to run on a headless system. To make sure that this works, always run your newly developed test cases with **build tests** using JDK 1.4 or later.

Test case classes that do require GUI components in place have filenames like `GUITest*.java`.

We should try to get as many tests from a GUITest* class to the corresponding Test* class because the latter are run by automatic builds regularly.

Every class `org.argouml.x.y.z` stored in the file `src_new/org/argouml/x/y/z.java` should have a JUnit test case class called `org.argouml.x.y.Testz` stored in the file `tests/org/argouml/x/y/Testz.java` containing all the Unit Test Cases for that class that don't need the GUI components to run. Tests that do need GUI components to run should be part of a class named `org.argouml.x.y.GUITestz` stored in the file `tests/org/argouml/x/y/GUITestz.java`

If you only want to run your newly written test cases and not all the test cases, you could start with the command **build run-with-test-panel** and give the class name of your test case like `org.argouml.x.y.Testz` or `org.argouml.x.y.GUITestz` . You will then get the output in the window. You could run all tests in this way by specifying the special test suite `org.argouml.util.DoAllTests` in the same way.

Every test case class imports the JUnit framework:

```
import junit.framework.*;
```

and it inherits `TestCase` (i.e. `junit.framework.TestCase`).

2.6.1.2. About the Test case Method

Methods that are tests must have names that start with "test" (i.e. all small t, e, s, t). This is a requirement of the JUnit framework.

Try to keep the test cases as short as possible. There is no need in cluttering them up just to beautify the output. Prefer

```
// Example from JUnit FAQ
public void testIndexOutOfBoundsExceptionNotRaised()
        throws IndexOutOfBoundsException {
    ArrayList emptyList = new ArrayList();
    Object o = emptyList.get(0);
}
```

over

```
public void testIndexOutOfBoundsExceptionNotRaised() {
    try {
        ArrayList emptyList = new ArrayList();
        Object o = emptyList.get(0);
    } catch (IndexOutOfBoundsException iobe) {
        fail("Index out of bounds exception was thrown.");
    }
}
```

because the code is shorter, easier to maintain and you get a better error message from the JUnit framework.

A lot of times it is useful just to run the compiler to verify that the signatures are correct on the interfaces. Therefor Linus has thought it is a good idea to add methods called `compileTestStatics`, `compileTestConstructors`, and `compileTestMethods` that was thought to include correct calls to all static methods, all public constructors, and all other public methods that are not otherwise tested. These methods are never called. They serve as a guarantee that the public interface of a class will never lose any of the functionality provided by its signature in an uncontrolled way in just the same way as the test-methods serve as a guarantee that no features will ever be lost.

Example 2.1. An example without Javadoc comments

```
package org.argouml.uml.ui;
import junit.framework.*;

public class GUITestUMLAction extends TestCase {
    public GUITestUMLAction(String name) {
        super(name);
    }

    // Testing all three constructors.
    public void testCreate1() {
        UMLAction to = new UMLAction(new String("hexagon"));
        assert("Disabled", to.shouldBeEnabled());
    }
    public void testCreate2() {
        UMLAction to = new UMLAction(new String("hexagon"), true);
        assert("Disabled", to.shouldBeEnabled());
    }
    public void testCreate3() {
        UMLAction to = new UMLAction(new String("hexagon"), true, UMLAction.NO_ICO
        assert("Disabled", to.shouldBeEnabled());
    }
}
```

and the corresponding no-GUI-class:

```
package org.argouml.uml.ui;
import junit.framework.*;

public class TestUMLAction extends TestCase {
    public TestUMLAction(String name) {
        super(name);
    }

    // Functions never actually called. Provided in order to make
    // sure that the static interface has not changed.
    private void compileTestStatics() {
        boolean t1 = UMLAction.HAS_ICON;
        boolean t2 = UMLAction.NO_ICON;
        UMLAction.getShortcut(new String());
        UMLAction.getMnemonic(new String());
    }

    private void compileTestConstructors() {
        new UMLAction(new String());
        new UMLAction(new String(), true);
        new UMLAction(new String(), true, true);
    }

    private void compileTestMethods() {
        UMLAction to = new UMLAction(new String());
        to.markNeedsSave();
        to.updateEnabled(new Object());
        to.updateEnabled();
        to.shouldBeEnabled();
    }
}
```

2.7. Generating documentation

This describes how to generate the documentation for ArgoUML.

If you are in a hurry, here's the quick version:

```
C:\Work>svn checkout -N http://argouml.tigris.org/svn/argouml/trunk/src_new argoum
C:\Work>svn checkout http://argouml.tigris.org/svn/argouml/trunk/documentation htt
C:\Work>set JAVA_HOME=C:\Programs\jdkwhatever
C:\Work>cd argouml\documentation
C:\Work\argouml\documentation>build docbook-xsl-get   (first time only)
C:\Work\argouml\documentation>build defaulthtml
```

The chunked HTML versions of the Cookbook, Quick Guide, and User Manual are built and the results are placed in: `C:\Work\argouml\build\documentation\defaulthtml\cookbook`, `C:\Work\argouml\build\documentation\defaulthtml\quick-guide`, and `C:\Work\argouml\build\documentation\defaulthtml\manual` respectively.

2.7.1. Building documentation

To build the documentation, you will need to check out the whole `argouml/documentation` directory. as well as the `argouml/tools` directory which contains the tools needed (Ant, FOP, Saxon, etc) and the file `argouml/src_new/default.properties` which contains the current version and other project-wide settings. None of the other ArgoUML source directories are needed if you are just building the documentation.

The subdirectories of `argouml/documentation`, `cookbook`, `manual`, and `quick-guide` each contain one of the three books. The subdirectory `docbook-setup` contains two things. It contains the configuration files that control how the generation is done. It contains the XSL rules for all the generation. The subdirectory `images` contains all the required pictures for all the books.

There are separate build targets available for each output format, as well as a target that builds all possible output formats. Use the command **build -p** to get a complete list of targets.

For testing purposes while editing a manual, you probably want to build just a single output format for the one manual that you are working with. To suppress the building of the other manuals, you can define the following properties in your `build.properties` file:

```
skip-cookbook=true
skip-quickguide=true
skip-argomanual=true
```

When, in the `documentation` directory, you run **./build.sh defaulthtml** or one of the other targets that builds the documentation, all books are built.

What happens is (the target internal-dispatcher):

- The *`manual/argomanual.xml`* is copied by ant to *`manual/argomanual-generated.xml`* while doing substitution of tokens: (@VERSION@ to become the version as specified in `default.properties`)
- The file *`manual/argomanual-generated.xml`* is processed by the special `docbook-setup/create-imglist.xsl` XSL script that generates a list of included images.

- All included images are copied.

  The purpose of this is to avoid copying images not actually used in the document.

- The HTML is generated by processing the file *manual/argomanual-generated.xml* .

The file *manual/argomanual-generated.xml* is a temporary file that only exists while processing the XML. If you are editing the XML be sure to edit the file *manual/argomanual.xml* and not the temporary file.

2.8. Setting up Eclipse 3

Linus Tolke

If you are running Eclipse 3.1 or 3.2 we have a prepared and recommended setup committed into our repository. This setup is slightly different than the ant setup w.r.t. file tree structure of the checked out copy and some special magic is used in the ant scripts that is controlled from the special Eclipse configurations.

The prepared setup includes settings for Checkstyle within Eclipse that will be enabled if you have the Checkclipse Eclipse plugin installed.

You need to have Subclipse installed in your Eclipse to retrieve the files from the repository. See http://subclipse.tigris.org/install.html.

If any of these instructions don't work or could be improved in some way, please help in making them better by contacting the editor of the Cookbook.

2.8.1. Checking out through Eclipse

This instruction is if you want to use Eclipse to download the source and it takes you up to where you can start ArgoUML from the source.

We have Team Project Set files prepared to make the set up automatic.

Do the following.

1. Change to the SVN Repository Explorer perspective and select Add SVN Repository. Then enter the following url **http://argouml.tigris.org/svn/argouml**.

2. Download the initial Team Project Set file and save it somewhere.

   Suggested by browsing to `trunk/tools/eclipse` and checking it out as a project.

   This is only needed the first time and if you don't already happen to have an updated checked out copy of argouml laying around.

3. Do File => Import and select Team Project Set and press Next.

4. Browse to the file `argouml-core-projectset.psf` and press Finish.

   Now everything is automatic even though it takes a while. What happens is:

   a. Files are downloaded from the Tigris SVN server and put into Eclipse projects.

      It is around 30Meg that is downloaded so beware if you are on a slow connection.

b. The projects are built. First things that are special for ArgoUML, generated Java code, are created by Eclipse calling ant, then Eclipse kicks in and compiles the Java code.

c. Eclipse finds the Configurations that are also checked out.

5. Select Project => Clean... => Clean all projects and wait for everything to build again.

6. Verify that you can start ArgoUML from the debugger within Eclipse. You can do this by clicking on the little down-arrow close to the Debug icon in the tool bar, selecting Debug..., and finding ArgoUML in the list (under Java Application).

2.8.2. Eclipse to help with the ArgoUML coding style

This instruction is to set up Eclipse to work according to the ArgoUML Coding standards. If this is not done correctly you will most likely find that you will have to do a lot of manual edits every time Eclipse has touched the code. You have your tool working against you instead of for you.

The instructions here are for these settings to affect your Eclipse Workspace. If you have other projects in the same Eclipse Workspace you would probably want this for only the ArgoUML projects and that is possible although not explained here. We suggest you to consider having ArgoUML work in an Eclipse Workspace separate from your other projects.

- Code conventions.

  1. Select in the menu Window => Preferences.

  2. Then select Java => Code Style => Formatter.

  3. For Eclipse 3.2 you do: Select Java Conventions [built-in].

  4. For Eclipse 3.1 you do: Select Import and give the file `eclipse/eclipse-argo-formatter.xml` located in the argouml-core-tools project that you just checked out.

     This takes care of the Eclipse-built-in bug (https://bugs.eclipse.org/bugs/show_bug.cgi?id=104765).

  5. Press OK.

  This will give you Code Conventions exactly like the Sun Coding Style that we use in the ArgoUML project.

- New file templates.

  1. Select in the menu Window => Preferences.

  2. Then select Java => Code Style => Code Templates.

  3. Select Import and give the file `eclipse/eclipse-argo-codetemplates.xml` located in the argouml-core-tools project that you just checked out.

  4. Press OK.

  This will set up templates to be used when using the Eclipse functions for creating files, functions, types and Javadoc...

2.8.3. Eclipse to automatically find problems in the code

This instruction is to set up Eclipse to automatically find what, in the ArgoUML project, could be considered problems in the code.

You can apply these individually depending on what level of help you need in your coding. I (Linus Tolke) recommend that you set them all on the Warning level. This makes them visible for you. You can then decide to fix them or not depending on how you feel about the code you are working with.

The instructions on where to find the different settings are for Eclipse 3.2. If you use Eclipse 3.1 you might need to search through the tabs to find where the setting is since they were reorganized for Eclipse 3.2.

- Compiler compliance level.

  See in the menu Window => Preferences. Then select Java => Compiler. At the right hand side set Compiler compliance level: 1.4.

  Uncheck Use default complicance settings and set Generated .class file compatibility: 1.4, Source compatibility: 1.4, and Disallow identifiers called enum to Error.

  In the ArgoUML project we have decided to keep source compliance to JDK 1.4. (See Section 3.3.1, “Choice of JRE: ArgoUML will support any JRE compatible with a Sun specification of any JRE from Sun that has not begun the Sun End of Life (EOL) process. ”). This setting enables Eclipse to tell you where some JDK 1.5 features have crept in. At the same time we don't want to introduce identifiers that are in conflict with Java5.

- Find forgotten and incorrect Javadoc comments.

  In the menu select Window => Preferences. Then select Java => Compiler => Javadoc. Suggested settings for these tabs (Only things diverting from the Eclipse defaults are listed):

  - Javadoc => Malformed Javadoc comments: Warning, Private, Check all reports.
  - Javadoc => Missing Javadoc tags: Warning, Private, check Check overriding and implementing methods.
  - Javadoc => Missing Javadoc comments: Warning, Protected, uncheck Check overriding and implementing methods.

  These problems (except missing Javadoc comments) are also found by Checkstyle so if you are running Checkclipse (See Section 2.8.4, “Settings for Checkclipse”) put this in Ignore instead.

- Code that hides other code.

  In the menu select Window => Preferences. Then select Java => Compiler => Error/Warnings. Suggested settings for these tabs (Only things diverting from the Eclipse defaults are listed):

  - Potential programming problems => Serializable class without serialVersionUID: Ignore

    We don't use serialization in the ArgoUML project anyway.

  - Potential programming problems => Possible accidental boolean assignment: Warning.
  - Name shadowing and conflicts => Local variable declaration hides another field or variable: Warning, check Include constructor or setter method parameters.
  - Name shadowing and conflicts => Field declaration hides another field or variable: Warning.

- Find Code that shall be removed.

  In the menu select Window => Preferences. Then select Java => Compiler => Error/Warnings. Suggested settings for these tabs (Only things diverting from the Eclipse defaults are listed):

  - Unnecessary code => Local variable is never read: Warning.
  - Unnecessary code => Parameter is never read: Warning.
  - Unnecessary code => Unused or unread private members: Warning.
  - Unnecessary code => Unnecessary semicolon: Warning.
  - Unnecessary code => Unnecessary cast or 'instanceof' operation: Warning.
  - Unnecessary code => Unnecessary declaration of thrown checked exception: Warning.

2.8.4. Settings for Checkclipse

Checkclipse is a plug in for Eclipse which needs to be installed separately. It enables style checking according the rules set for the ArgoUML project.

Get the latest Checkclipse kit from SourceForge at http://sourceforge.net/projects/checkclipse and install it by unzipping into your Eclipse plug-ins directory and restarting Eclipse. (Checkstyle is bundled with Checkclipse so it's not necessary to install it separately, but the web site is http://checkstyle.sourceforge.net/ for reference.)

Most ArgoUML projects in Eclipse have their Checkclipse settings predefined which should be found as soon as you install Checkclipse, but if you need to set them up by hand, use the following instructions. These instructions are for Checkclipse 2.1.

In the Java perspective, select the project argouml, i.e. the icon at the top of the Package Explorer. Then, in the menu, select Project => Properties, select Checkclipse (appears only if Checkclipse is correctly installed) and then fill the fields like this:

- Enable Checkstyle - Checked.
- Set Project Classloader - Checked.
- Checkstyle Configuration File: `/argouml-core-tools/checkstyle/checkstyle_argouml.xml` in the argouml-core-tools project.
- Checkstyle Properties File: `/argouml-core-tools/checkstyle/checkstyle.properties` in the argouml-core-tools project.

Leave the rest of the fields at their default (empty). The File Filters are defined on an additional preferences tab rather than in separate file as in earlier versions of Checkclipse. If this isn't populate with the saved values from SVN you can add individual files to the exclude list as you encounter them, but generally any machine generated source file (JavaLexer, JavaRecognizer, etc) should be excluded from the checks.

2.8.5. Running JUnit test cases from within Eclipse

Most of the JUnit test cases belong to the argouml-core-tests Eclipse project. The argouml-core-tests Eclipse project has its compile time dependencies set up to include the things needed to compile the test cases. This means that it is possible to compile the test cases and nobody will create tests that uses the insides of some subsystem that is supposed to be hidden.

The Model subsystem is separated in two parts:

- The interfaces and some bootstrap code in the argouml-core-model Eclipse project.
- The MDR implementation in the argouml-core-model-mdr Eclipse project.

There is also for test purposes a partly implemented Mock implementation of the model subsystem in the `org.argouml.model.MockModelImplementation`-class for the purpose of testing the interfaces and the bootstrap code but it requires the test cases to be written especially for that.

The tests are never to be compiled against the MDR-implementation but against the interfaces. This means that the MDR-implementation is not included in the project dependencies.

When it comes to running the tests, most of the tests require the Model subsystem working to succeed. To run the application with a working Model subsystem, a working implementation is needed so the tests require the MDR-implementation.

The simplest way to solve this is to:

1. Create the test (by right-clicking on any of the test classes and select Run as JUnit Test or Debug as JUnit Test).

   The test fails with a `java.lang.ExceptionInInitializerError` on the first reference to Model. If it doesn't then the perhaps the test case doesn't use the Model.

2. Select Debug... or Run... to get the configuration editing box. It has remembered the test case and it is selected.

3. Add the argouml-core-model-mdr to the classpath.

   Classpath tab, Select User Entries, Add Projects, Check argouml-core-model-mdr, make sure Add exported entries of selected projects and Add required projects of selected projects are checked, Press OK, Press Debug or Run. Eclipse will remember these changes to that configuration but you will have to do it all over again once for each new test case.

2.8.5.1. Running all JUnit test cases from within Eclipse

The description so far describes how to do this for a single test case at the time. If you want to run all test cases in one go from within Eclipse it is also possible. We have not prepared that for three reasons:

- Some of the tests fail when run in this way.

  The guess (Linus July 2006) is that this depends on the fact that Eclipse doesn't start each test in its own JVM and some of the tests relies on a fresh environment (empty models...). Let's hope that an upcoming version of Eclipse includes a function to allow us to specify that the tests are to be run each in its own JVM.

  Another cause might be that the tests relies on certain other files being in specified places and that the Eclipse set up doesn't provide that.

- Maintaining the Configuration.

  Since we have an Eclipse setup with a mismatch between the compile time dependenies and the run time dependencies this Configuration contains references to specific jar-files. This means a maintenance problem whenever changing version of a dependency.

- Selection of tests.

  In the project we keep all tests in the same tree, whether they are working or not. We distinguish between test classes with names starting with "Test", "GUITest", and neither of them. The ones starting with "Test" or "GUITest" are official tests that should be working. All others are tests that are never run automatically. They are perhaps used for other purposes. In Eclipse, there is no way to specify this so the description below will run all tests.

  Such a test is the DoAllTests-test suite that runs all tests so tests risk to be run twice.

If you want to test this, this is what you do:

- Right-click on the project argouml-core-tests.
- On the multilevel popup that opens, select Run As... JUnit Test or Debug As... JUnit Test.
- A Configuration is created on that project with the Run all tests in the selected project, package or source folder: checked and the configuration is started/launched.
- Stop the running Configuration.

  The newly Configuration has the same problem as every single test above with the model subsystem so it won't work.

- Press Debug... or Run... to open the Configuration editor.

  The newly created Configuration argouml-core-test is selected.

- Add the argouml-core-model-mdr project to the classpath.

  This is done by:

  - selecting the Classpath tab,
  - selecting argouml-core-tests under User Entries,
  - clicking Add projects, and
  - selecting the argouml-core-model-mdr project and press OK.

- Run all the tests by pressing Debug or Run.

2.9. Settings for NetBeans

Linus Tolke

Warning

It is unclear what version of NetBeans these settings work in. If you test it, let us know so that we can update the Cookbook.

The argouml style guides correspond to the following settings in NetBeans:

- In (Tools =>) Options => Editing => Editor Settings => Java Editor

  Tab Size = 8

- In (Tools =>) Options => Editing => Indentation Engines => Java Indentation Engine

  Add Newline Before Brace: False, Add Space Before Parenthesis: False, Expand Tabs to Spaces: False, Number of Spaces per Tab: 4 (Should probably be read as Number of Spaces per indentation level).

2.10. Settings for Emacs

Linus Tolke

These style guides correspond to the default Java settings in Emacs:

```
("java"
 (c-basic-offset . 4)
 (c-comment-only-line-offset 0 . 0)
 (c-offsets-alist
  (inline-open . 0)
  (topmost-intro-cont . +)
  (statement-block-intro . +)
  (knr-argdecl-intro . 5)
  (substatement-open . +)
  (label . +)
  (statement-case-open . +)
  (statement-cont . +)
  (arglist-intro . c-lineup-arglist-intro-after-paren)
  (arglist-close . c-lineup-arglist)
  (access-label . 0)
  (inher-cont . c-lineup-java-inher)
  (func-decl-cont . c-lineup-java-throws)))
```

2.11. Making a release

The purpose of this chapter is to simplify for the person that is actually doing the release work and to make sure that everything is done in the exact same way every time and nothing is forgotten.

The scripts involved have been developed and are mostly run on a Cygwin system. They will hopefully work on any UNIX system but most likely they will need some adjustments.

The scripts and tools used specifically for the build are maintained in the argoumlinstaller project. From the argouml project the files `argouml/src_new/build.xml` and other `build.xml` files are re-used.

Prerequisites (what you need to be able to do this):

- Subversion access to the argouml projects (to create the releases branch/tag). The projects involved are specified by `argoumlinstaller/build-release.sh`.

- Subversion access to the argouml-downloads project (to upload the result).

- A machine with 3GB of disk to use for this purpose (September 2006).

  This is probably the machine you use for your development if you are an argouml developer.

  The machine needs Internet access (it is not a small download and upload so at least 128KB Internet connection to keep the time reasonable < 2 hours), the correct version of Java installed (should be a JDK 1.4.2), SVN installed, Unix or Cygwin to be able to run the scripts.

- The argoumlinstaller and argouml-downloads projects checked out alongside eachother.

  If this is not in place from a previous release this is done using the commands

```
cd wherever
svn co http://argoumlinstaller.tigris.org/svn/argoumlinstaller/trunk argoumlinst
svn co http://argouml-downloads.tigris.org/svn/argouml-downloads/trunk argouml-d
```

  Note that the argouml-downloads checkout is large (almost 1.5 GB) and will take a considerable time to check out so you'd better do this in advance.

- You have generated a key to sign the jar files (for Java Web Start).

  Run the command **keytool -list -v** and give the keystore password *`secret`*. You should have a key named argouml that is valid several months in the future.

  This is to make sure that you have a valid key for the purpose of signing the jar files.

> Since the ArgoUML project and the Tigris organization are loose organizations we cannot buy a "real" key. The keys we use are the unsigned keys that can be generated by anyone using the keytool provided with Java.

  A key is generated with the command **keytool -genkey -alias argouml -storepass secret**.

  By default these keys have a validity of just three (3) months but by giving the *`-validity days`* the validity can be extended.

  Don't forget to upload your new key to the Downloads area. This is for those who want to see the key on the site separately.

Here are the steps to be done when one actually does a release:

1. Check for new projects.

   If there are any new projects to be included in the release, add them to the list of projects in `argoumlinstaller/build-release.sh`. You also need to create the `releases`-directory at the top of the SVN repository.

2. Create the release branch/tag and checkout that copy.

   This is done using the command **./build-release.sh -tc** in the argoumlinstaller project and giving the release name.

   You must have set JAVA_HOME for this to work.

   The script will check that the releases top directory is present in all the involved projects and that

the given release name is not already present in any of the involved projects.

3. Set the `argo.core.version` to not include the "PRE-" part.

   This is done in the `default.properties`-file in `build/VERSION_`*`GIVEN_VERSION`*`/argouml/src_new` and then, commit the file.

4. Build ArgoUML and the subprojects, and sign the jar files.

   This is done using the command **./build-release.sh -bs**

5. Build the pdf version of the documentation.

   This is done using the command **./build-release.sh -d**

6. Go through Issuezilla and check things.

   Things to check are:

   a. That there is a Version created in Issuezilla for the newly created release.

      The purpose of this is to make it possible for everyone to report bugs on the new release.

   b. Make sure that the upcoming releases have target milestones created for them. This needs to be done for all components that has the same release scheme. Also see that the numbering is the same in all components and that it is in the correct chronological order except for the not yet done releases that come before the already completed.

   c. Change the target milestones of all the not yet resolved issues for this release to ---.

   d. Change the target milestones of any fixed issue in component argouml with target milestone --- to that of the current release.

      This is probably some developer that has fixed an issue but forgotten to set the target milestone correctly.

   e. Move all issues reported on 'current' to this release (for the component argouml).

      These items were reported between the previous version and this version. Since 'current' will be reused for the next release, they need to be locked to the closest release to where they were found.

   f. Reopen RESOLVED/REMIND

      This can also be a good time to change all RESOLVED/REMIND. Search for them and Reopen them.

   g. Check RESOLVED/LATER

      It could also be good to check that all RESOLVED/LATER has a valid target milestone (must be an upcoming milestone). Search for them and Reopen the ones that haven't. Also, if the milestone denotes or is going to be resolved in the upcoming release, Reopen them with a comment that they are now active.

After this, the work continues with the installers.

This is what you do:

1. Create the zip files and the tgz files, copy the documentation, copy changed Java web start files and create new Java web start jnlp files.

   This is done by the command **./official.sh**.

2. For Java Web Start, update the "Latest development" or perhaps the "Latest stable" files essentially with the contents of the newly create JNLP file.

   These files are located in the `svn/argouml-downloads/www/jws`-directory.

3. Update the index file for the downloads project to point out the new release.

   It should point out the release at `/argouml-`*`RELEASENAME`*`/`, the Java web start file at `/jws/argouml-`*`RELEASENAME`*`.jnlp`.

4. Commit the release in the argouml-downloads project

   The following commands will do it for you:

```
cd ../argouml-downloads/www
svn commit -m'The release RELEASENAME.'
```

2.11.1. The release did not work

Warning

This description is not yet updated to fit the subversion set up for ArgoUML.

This shouldn't happen! This really shouldn't happen!

The reason that this has happened is that one of the developers has made a mistake. You now must decide a way forward.

2.11.1.1. Fix the problem yourself.

If the problem is obvious to you and you can fix it quickly, do so. This is done by doing the following:

- Make the release tag into a branch
- Checked out that branch
- Fix the problem in your checked out copy
- Commit the problem in the branch
- Continue the build process

  This is done by restarting the **build dist-release**-command and from that point on working in the branch instead of at the tag.

- Explain to the culprit what mistakes he has made and how to fix it.

  It is now his responsibility to make sure that the problem will not appear in the next version. He can do this either by merging in your fix or by fixing the problem in some other way.

At this point an in-detail description of how poor programming skills the culprit has and how ugly his mother is, is probably in place but please keep it constructive! Remember, you might be mistaken when you guess who the responsible is.

2.11.1.2. Delay the release waiting for someone to fix the problem.

Create the branch as described in Section 2.11.1.1, “Fix the problem yourself.”. Then tell the culprit and everyone on the developer list what the problem is and that it is to be fixed in the release branch a.s.a.p.

Monitor the changes made to the branch to verify that no one commits anything else but the solutions to the problems.

When you get notified that it is completed, update your checked out copy and continue the release work.

Chapter 3. ArgoUML requirements

Linus Tolke

This chapter contains a description on how ArgoUML should work and behave for the users.

These things might not be implemented yet and the solutions might not even be clear but it is a definition of the goal.

The fact that it is not implemented or doesn't work as stated here should be registered as a bug in the bug registering tool.

Every requirement has a number (REQ1, REQ2, REQ3, ...) that never changes, a revision (REVa, REVb, REVc, ...) that changes when the requirement change, a text that is the requirement text to implement, a rationale that is the description on why this is important, a stakeholder that is one of the stakeholders in the vision for who this is important.

3.1. Requirements for Look and feel

This describes how the ArgoUML look and feel shall behave.

3.1.1. When multiple visual components are showing the same model element they shall be updated in a consistent manner throughout the application.

REQ1 REVa

Rationale: There is no way of telling where the user is looking while working with ArgoUML. For this reason he might be terribly confused if some other view that happens to show the same element is not showing the same thing.

Stakeholder: User of ArgoUML

3.1.2. All views of a model element shall be update as soon as the model element is updated.

REQ2 REVb

Rationale: If a user makes an update of a part of the model, an immediate feedback in all other parts that are currently showing might help him to get it right.

Stakeholder: User of ArgoUML

3.1.3. Editable views of the model should update the model on each keystroke and mouse click.

REQ11 REVa

Rationale: If a user makes an update of a part of the model, an immediate feedback in all other parts that are currently showing might help him to get it right.

Stakeholder: User of ArgoUML

3.1.4. Any text fields that require validation should not be editable directly from a view.

REQ12 REVa

Rationale: If a text field requires validation there exists, by definition, a possibility that the text field is in an invalid state at any time during editing. Therefore the model cannot be updated until the field is completed in a valid state or rejected.

Stakeholder: User of ArgoUML. TODO: Is this the correct stakeholder?

3.1.5. With dialogs, the model is not updated until the dialog is accepted by the user with valid fields.

REQ13 REVa

Rationale: It is a common feature of GUIs that a dialog displays a snapshot of its model at the time of creation and only updates that model on the user acceptance of the entire dialog. This is a familiar look and feel for users.

Stakeholder: User of ArgoUML.

3.1.6. The user shall receive some visual feedback during the edit process of textual UML to indicate whether the text represents valid UML syntax.

REQ14 REVa

Rationale: Writing a correct syntax of anything is complicated. Good compilers are helpful in pinpointing where the problem is (what line and what token is in error). The text fields in ArgoUML are not developed in the same way as source code and we have no compiler step to verify it all. Instead this validation needs to be done while editing meaning that the user needs all the help he can get to as quickly as possible, get the syntax right. TODO: Is this the correct motivation for this?

Stakeholder: User of ArgoUML.

3.1.7. There shall be no indication of an exception on the screen or in the log if it has occured merely because of a user mistyping or not being aware of UML syntax.

REQ3 REVa

Rationale: An exception in the log or on the screen is always the sign of a serious error in the application that should be reported as a DEFECT. If a mistyping generates such a problem the user might loose interest in ArgoUML as a tool because he percieves it as not working correctly.

Stakeholder: User of ArgoUML

3.1.8. All text fields shall have context sensitive help.

As follows:

1. A tooltip that explains the data and format expected by the particular field.

   This can be omitted when there is a header stating the data of the field and the format is obvious.

2. Pressing F1 or choosing help from the menu shall display a popup window explaining for data and format required by the current input field. Input focus shall be left on the field during any user interaction with the popup (dragging, scrolling or closing).

REQ4 REVa

Rationale: Throughout a complex application like ArgoUML there are lots of text fields. Unless there is a possibility to always get this kind of help the user might not be able to make out what he is actually supposed to do in that field.

Stakeholder: User of ArgoUML

3.2. Requirements for UML

3.2.1. ArgoUML shall be a correct implementation of the UML 1.4 model.

REQ5 REVa

Rationale: The vision of ArgoUML is to provide a tool that helps people work with an UML model. The UML model might later on be used in some other tool. If the implementation is not correct then ArgoUML will not be compatible with that other tool or the user will be confused. There might be a lot of tough decisions when it comes to if it is ArgoUML or some other tool that deviates from the UML 1.4 but there shall never be any doubt that the intention of ArgoUML is to implement UML correctly.

Stakeholder: User of ArgoUML

3.2.2. ArgoUML shall implement everything in the UML 1.4 model.

REQ6 REVa

Rationale: The ambition is to implement all of UML. This means that no matter how you use UML ArgoUML will always be a working tool.

Stakeholder: User of ArgoUML

3.3. Requirements on java and jvm

3.3.1. Choice of JRE: ArgoUML will support any JRE compatible with a Sun specification of any JRE from Sun that has not begun the Sun End of Life (EOL) process.

REQ7 REVb

Rationale: The JREs and the adjoining libraries (especially swing) are always improving to include new features and new ideas. The developers of ArgoUML would like to use these new features.

Note: J2SE 1.3.1 begun its Sun End of Life (EOL) process on October 25, 2004.

Stakeholder: Developers of ArgoUML

3.3.2. Download and start

It shall be possible to install ArgoUML locally on the machine and use without Internet connection.

REQ8 REVa

Rationale: ArgoUML is an application that edits an UML model. There is no need to have any network defined while doing this.

Stakeholder: User of ArgoUML

3.3.3. Console output: Logging or tracing information shall not be written to the console or to any file unless explicitly turned on by the user.

REQ9 REVa

Rationale: ArgoUML is an application that edits an UML model. Any information written to anywhere but the files that the user specifies the user won't know what to do with and it will be perceived as garbage generated by the ArgoUML application.

Stakeholder: User of ArgoUML

3.4. Requirements set up for the benefit of the development of ArgoUML

3.4.1. Logging: The code shall contain entries logging important information for the purpose of helping Developers of ArgoUML in finding problems in ArgoUML itself.

REQ10 REVa

Rationale: When the developers are searching for some problem or when they ask any of the users to help them pinpoint some problem such logging messages are very helpful.

Stakeholder: Developers of ArgoUML

Chapter 4. ArgoUML Design, The Big Picture

Currently this is more of a base for discussion and ambition but hopefully this will mature and prove useful.

The code within ArgoUML is separated in subsystems that each have a few responsibilities.

In Issuezilla each subsystem has its issues sorted in a subcomponent with the same name as the subsystem. Furthermore the Diagrams subsystem has a set of subcomponents for issues connected to the specific diagrams.

This chapter gives an overall picture with a list of subsystems, their dependencies, and their main responsibility. Chapter 5, *Inside the subsystems* explains each subsystem in detail.

The subsystems are organized in layers. The purpose of the layers is to make it easy to see in what direction the dependencies are and thus allow us to know what dependencies are to be removed in the cases where we have circular dependencies. This will also allow us to know which other subsystems that are involved when testing a subsystem.

4.1. Definition of subsystem

All ArgoUML code is organized in subsystems.

Each subsystem has:

- A name
- A single directory/Java package where it resides

  Subparts of the subsystem can reside in subdirectories of this directory. Auxiliary parts, implemented in other products, of the subsystems can reside somewhere else. Notice that each other product used by ArgoUML is, in the design, located within one of the existing subsystems. This means that a change of version or indeed a change of choice of such a dependency is an internal matter for the subsystem and should ideally not affect any other subsystem.

  All public and protected methods of all public and protected classes in this directory constitute the API of that subsystem.
- A section in the chapter Chapter 5, *Inside the subsystems* .

  The section shall for each subsystem contain the responsibilities, the package name, the API, the Facade (if any), all the plug-in interfaces (if any). This shall be in the first part of the section i.e. not in a subsection. It should also document the design of the subsystem. This is in subsections.

Each subsystem can have:

- a Facade class

  The facade can be used by all other subsystems when using the subsystem.

  The Facade class is called *`SubsystemName`* `Facade` and is located in the subsystem package.

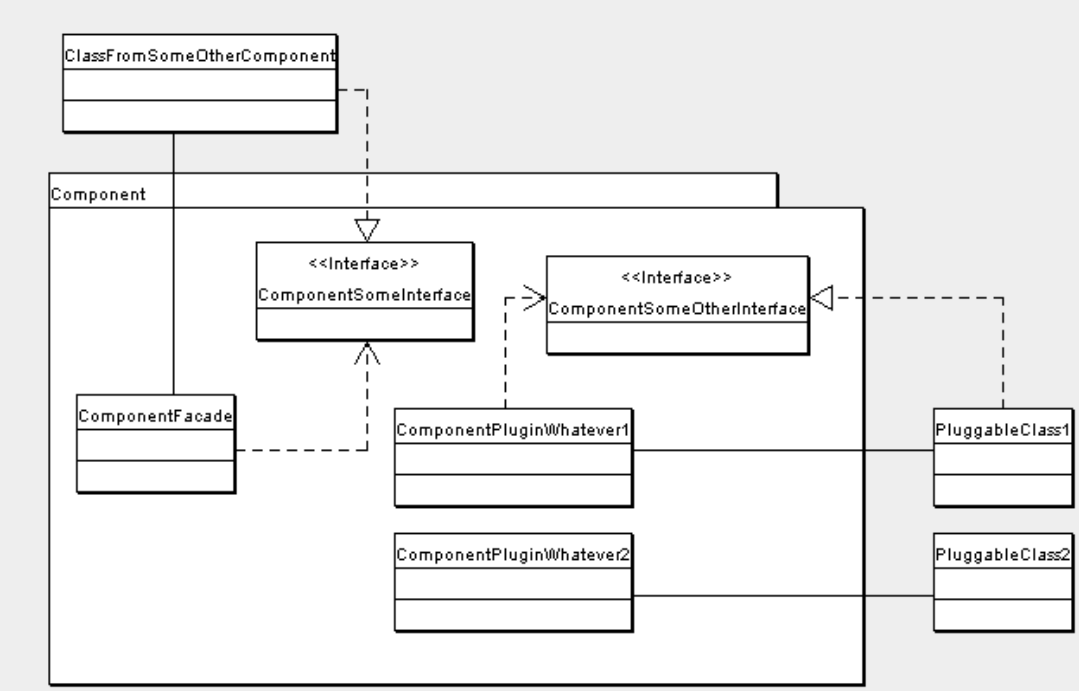
How it is used is primarily documented in the class file itself (as javadoc) but the more complex picture is documented in the Cookbook (in Chapter 5, *Inside the subsystems*).

- Plug-in interfaces.

  These are Facade objects where modules or plug-ins can connect themselves to modify or augment the behavior of that subsystem.

- The plug-in interfaces are also all located in the subsystem package and called *`SubsystemName`*`Plugin`*`PluginType`* . Example: `ModelPluginDiagram`, `ModelPluginType`.

- If the subsystem uses a callback-technique the callback is always made to an interface defined by the subsystem. The interface is also in the subsystem package and it is called *`SubsystemName PluginType`*`Interface` . Example: `ModelDiagramInterface`, `ModelTypeInterface`.



4.2. Relationship of the subsystems

Each subsystem that is used by other subsystems provide two ways for other subsystems to use them:

- The Facade class

  The use of Facade class is not wide spread in ArgoUML. This is because ArgoUML is traditionally built as a whole and no subsystems were clearly defined.

  A Facade class provides the most common functions other subsystems want to do when using that subsystems to reduce the need of having to use anything else but the Facade class. The Facade class should be very much more stable than the subsystem itself. Methods in the Facade should change really slowly and only be removed after several months (and one stable release) of deprecation.

  The Facade class is documented in the class file itself (as javadoc) and the more complex picture (if needed) is documented in the Cookbook (in Chapter 5, *Inside the subsystems*).

- An API with calls to public or protected methods.

  Traditionally, the subsystems in ArgoUML communicate through public methods and public variables and the subsystems, as defined by the responsibilities, are spread over several packages setting aside the Java visibility rules. For this reason it is not well-known or documented what public methods form part of a subsystem's API and what public methods are internal to a subsystem. For this reason, always exercise extreme caution when changing the signature of a public method. (See Section 7.2, “When Committing to the Repository”.)

  In order to improve things, make it very clear when encountering and understanding the purpose of a public method or class, if it is part of the subsystem's API or not (by improving the javadoc for that method or class).

  Try to help in moving the public API methods and classes from wherever, to the subsystem's directory/package using the proper deprecation procedure.

  In order not to worsen things, always add new API classes and methods in the subsystem's directory/package.

  This way of communicating is still to be used when it is not convenient to use the Facade for a specific use of that subsystem.

Notice that the Facade is normally a part of the API or a simplified version of the API.

For each subsystem X in ArgoUML that uses the subsystem Y, the designer of the subsystem X must decide if he wants to use the API of Y when using the subsystem Y (putting a set of import org.argouml.Y.internals.blabla.*; statements in each file of subsystem X that uses subsystem Y) or use the Facade class of subsystem Y (putting only one import org.argouml.Y.YFacade; in each file in the subsystem X that uses subsystem Y).

The API solution makes the subsystem X depending on the subsystem Y meaning that when we change the API of the subsystem Y we must also change subsystem X. The facade calls solution doesn't make the subsystem X depending on the API of subsystem Y but just the Facade of subsystem Y.

The choice between the usage of the API or the Facade shall be stated in the Cookbook's description of subsystem X in the list of used subsystems.

4.3. Low-level subsystems

These subsystems are infrastructure subsystems that just are there for every other subsystem to use.

They are all insignificant enough not to be mentioned when listing dependencies.

All these subsystems are all started and initiated (if needed) from the Application subsystem.

- Logging - see Section 5.15, “Logging”
- Internationalization - see Section 5.14, “Internationalization”.
- GUI - see Section 5.11, “The GUI”.
- Help system - see Section 5.13, “Help System”.
- JRE with utils - see Section 5.16, “JRE with utils”

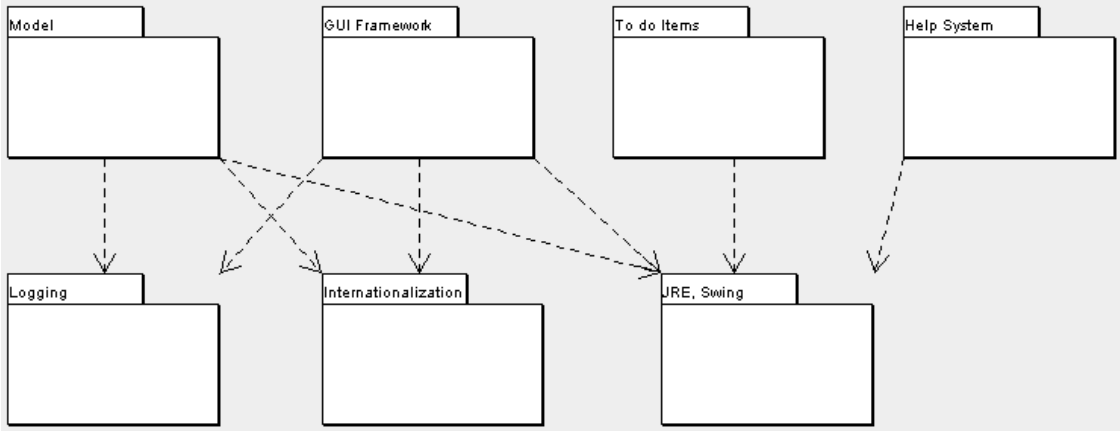
4.4. Model subsystems

These subsystems do not rely on any other part of ArgoUML to do their work (except the infrastructure). They can all be tested in full individually i.e. independent of any other subsystem.

They contain all the information used by all other subsystems so for that reason they represent the model in the MVC pattern.

All these subsystems are all started and initiated from the Application subsystem.

- The Model - See Section 5.1, “Model”.
- To do items - see Section 5.17, “To do items”.



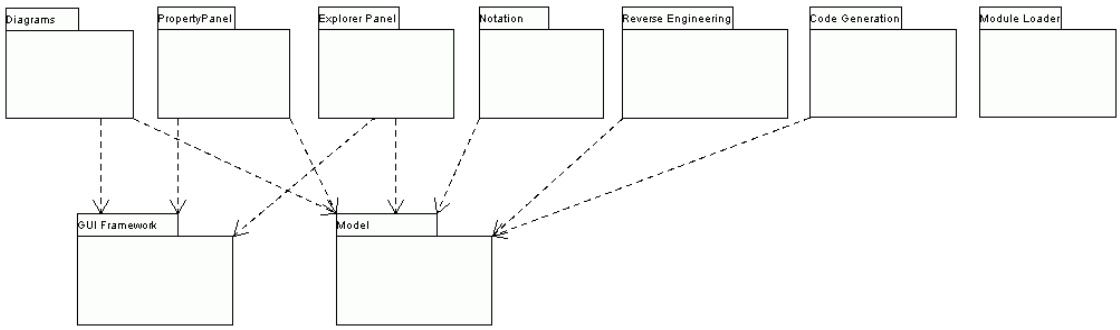
4.5. View and Control subsystems

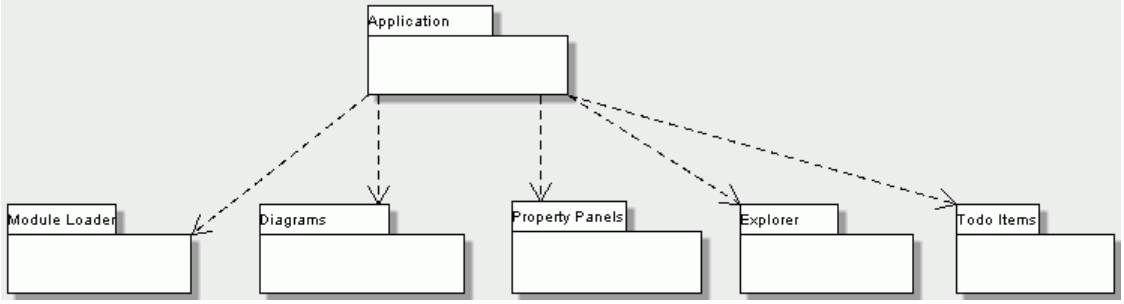
These subsystems rely on the information in the model subsystems in order to do their work.

All these subsystems are all started and initiated from the Application subsystem.

- Application - see Section 5.12, “Application”.
- Diagrams - see Section 5.3, “Diagrams”.
- Property panels - see Section 5.4, “Property panels”.
- Explorer - see Section 5.18, “Explorer”.
- Notation - see Section 5.6, “Notation”.

- Code Generation - see Section 5.8, “Code Generation Subsystem”.
- Reverse Engineering - see Section 5.7, “Reverse Engineering Subsystem”.
- Module loader - see Section 5.19, “Module loader”.

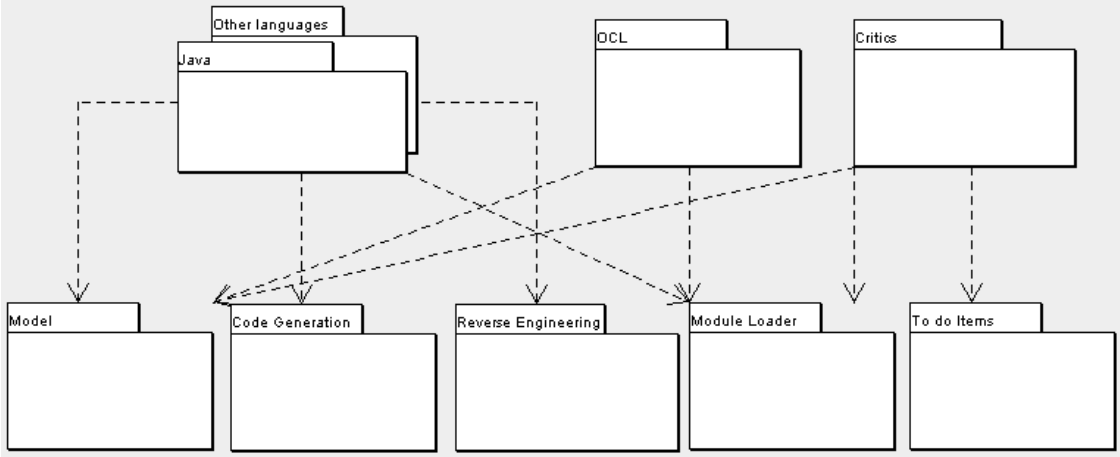




4.6. Loadable subsystems

These subsystems should be connected only through the interfaces provided by other subsystems. This means that they can be individually enabled and disabled using the module loader. Note: The old module loader does not have the possibility to disable modules. The new one has but is not widely used. (July 2005).

- Java Code generation, Reverse engineering - see Section 5.9, “Java - Code generations and Reverse Engineering”.
- Other languages - Code generation, Reverse engineering - see Section 5.10, “Other languages”.
- Critics and checklists - see Section 5.2, “Critics and other cognitive tools”.
- OCL - see Section 5.20, “OCL”.


Other languages
Java
OCL
Critics
Model
Code Generation
Reverse Engineering
Module Loader
To do Items

Chapter 5. Inside the subsystems

Warning

This chapter is currently under rework with new subsystem organization.

Things that are not actually in place are: TargetManager

...

5.1. Model

Purpose - To remove knowledge from the rest of ArgoUML of what model repository is in use (e.g. MDR, EMF, NSUML) and to give a consistent interface for manipulating data within those repositories.

The Model is located in `org.argouml.model`.

The Model is a Model subsystem. See Section 4.4, “Model subsystems”.

Currently there is a full implementation using NetBeans MDR to store the OMG UML 1.4 metamodel. The previous implementation used the NSUML library to implement a UML 1.3 metamodel.

The decision of which implementation to use is controlled by the Model class which contols the implementations as alternative strategies (as in the Strategy Pattern - GOF p315)

The Model class provides the rest of ArgoUML with various interfaces through which ArgoUML can manipulate the repository.

Currenty there are factory and helper interfaces for controlling the lifetime and properties of elements in the repository.

An interface is also made available to the Diagram Interchange Model should the repository implementation contain such.

A ModelEventPump interface is provided through which ArgoUML can listen for changes in the repository in a consistent way. Implementations of this pump convert from the repository specific events to ???

The factories contain all methods that deal with creating and building model elements. The helpers contain all utility methods needed to manipulate the model elements. Per section of the semantics chapter of the UML specification there is one factory and one helper.

Both helpers and factories (and the Facade and ModelEventPump) are interfaces that are fetched through static methods in the Model object.

Because the same interface is used internally each implementation must provide objects for each of these interfaces.

5.1.1. Factories

The factories contain in most cases a create method for each model element. Example: `createClass` resides in `CoreFactory`-interface.

Besides that, there are several build methods to build classes. The build methods have a signature like

```
public Object buildMODELELEMENTNAME
                (
                      params
                   );
```

.

Each build method is intended to follow the wellformedness rules as defined in the UML spec. The reason for having extar build methods, is that the model repository does not enforce the wellformedness rules even though, in som cases, non-well-formed UML can lead to non-well-formed XMI which leads to saving/loading issues and all kinds of illegal states of ArgoUML.

If you want to create an element you shall use the build or create methods in the factories. You are strongly advised to use a build method or, if there is none that suits your needs, to write a new one reusing the already existing build methods and utility methods in the helpers. The reason for this is that the event listeners for the newly created model element are setup correctly.

Question: Am I allowed to call the factories from any thread? Answer: The current checks are not written to allow for multiple threads so don't!

5.1.2. Helpers

The helpers contain all utility methods for manipulating model elements. For example, they contain methods to get all model elements of a certain class out of the model (see `getAllModelelementsOfKind` in `ModelManagementHelper`).

To find a utility method you need to know where it is. As a rule of thumb, a utility method for some model element is defined in the helper that corresponds with the section in the UML specification. For example, all utility methods for manipulating classes are defined in `CoreHelper`.

There are a few exceptions to this rule, mainly if the utility method deals with two model elements that correspond to different sections in the UML specification. Then you have to look in both corresponding helpers and you will probably find what you are searching for.

Question: Am I allowed to call the helpers from any thread? Answer: The current checks are not written to allow for multiple threads so don't!

5.1.3. The model event pump

5.1.3.1. Introduction

Late 2002, the ArgoUML community decided for the introduction of a clean interface between the NSUML model and the rest of ArgoUML. This interface consists of three parts:

1. The model factories, responsible for creation and deletion of model elements
2. The model helpers, responsible for utility functions to manipulate the model elements and
3. The model event pump, responsible for sending model events to the rest of ArgoUML.

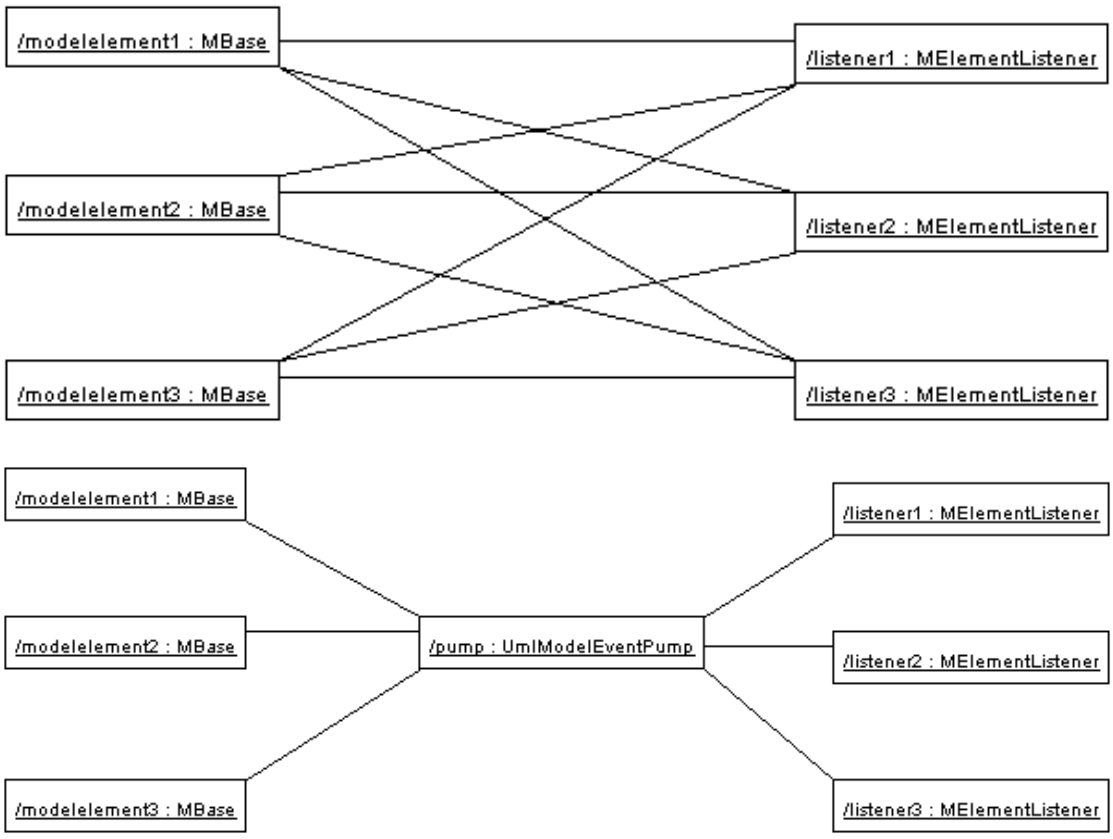
The model factories and the model helpers are described in Section 5.1.1, “Factories” and Section 5.1.2, “Helpers” respectively.

In the beginning of 2003, in the work to replace NSUML, the need was seen for this interface to not use any NSUML classes. The `ModelFacade` was introduced to wrap model factories, model helpers, and direct calls to NSUML but not the model event pump. In April 2004 a `ModelEventPump`-interface

was introduced to wrap the `UmlModelEventPump` using `PropertyChangeEvent`s.

The model event pump is the gateway between the model elements and the rest of ArgoUML. Events fired by the model elements are caught by the pump and then 'pumped' to those listeners interested in them. The main advantage of this model is that the registration of listeners is concentrated in one place (see picture *). This makes it easier to change the interface between the model and the rest of ArgoUML.

Besides this, there are some improvements to the performance of the pump made in comparison to the situation without the pump. The main improvement is that you can register for just one type of event and not for all events fired by some model element. In this respect the pump works as a filter.



The model event pump will replace all other event mechanisms for model events in the future. These mechanisms (like UMLChangeDispatch and ThirdPartyEventlisteners for those who are interested) are DEPRECATED. Do not use them therefore and do not use classes that use them.

5.1.3.2. Public API

You might wonder: how does this all work? Well, very simple in fact.

A model event (from now on a Event) has a name that uniquely identifies the type of the event. In most cases the name of the Event is equal to the name of the property that was changed in the model. In fact, there is even a 1-1 relationship between the type of Event and the property changed in the model. Therefore most listeners that need Events are only interested in one type of Event since they are only interested in the status of 1 property.

TODO: What thread will I receive my event in? What locks will be held by the Model while I receive my event i.e. is there something I cannot do from the event thread?

In the case described above (the most common one) you only have to subscribe with the pump for that type of event. This is explained in section Section 5.1.3.2.1, “ How do I register a listener for a certain type event ” and Section 5.1.3.2.2, “How do I remove a listener for a certain event”

Besides the case that you are interested in only one type of event (or a set of types), there are occasions that you are interested in all events fired by a certain model element or even for all events fired by a certain type of model element. For these cases, the pump has functionality too. This is described in section Section 5.1.3.2.3, “ Hey, I saw some other methods for adding and removing? ”.

5.1.3.2.1. How do I register a listener for a certain type event

This is really very simple. Use the model

```
addModelEventListener(PropertyChangeListener listener, Object modelelement, String
```

like this:

```
Model.getPump().addModelEventListener(this, modelelementIAmInterestedIn, "IamInter
```

Now your object this gets only the `Events` fired by modelElementIAmInterestedIn that have the name "IamInterestedInThisEventnameType".

5.1.3.2.2. How do I remove a listener for a certain event

This is the opposite of registering a listener. It all works with the method

```
removeModelEventListener(PropertyChangeListener listener, Object modelElement, Str
```

on the `ModelEventPump` like this:

```
Model.getPump().removeModelEventListener(this, modelelementIAmInterestedIn, "IamIn
```

Now your object is not registered any more for this event type.

5.1.3.2.3. Hey, I saw some other methods for adding and removing?

Yes there are some other method for adding and removing. You can add a listener that is interested in ALL events fired by a certain model elements. This works with the method:

```
addModelEventListener(PropertyChangeListener listener, Object modelelement)
```

As you can see no names of events you can register for here.

Furthermore, you can add a listener that is interested in several types of events but coming from 1 model element. This is a convenience method for not having to call the methods explained in section Section 5.1.3.2.1, “ How do I register a listener for a certain type event ” more than once. It works via:

```
addModelEventListener(PropertyChangeListener listener, Object modelelement, String
```

You can pass the method an array of strings with event names in which your listener is interested.

Thirdly there is a very powerful method to register your listener to ALL events fired by a ALL model elements of a certain class. You can understand that using this method can have severe performance im-

pacts. Therefore use it with care. The method is:

```
addClassModelEventListener(PropertyChangeListener listener, Object modelClass)
```

There are also methods that allow you to register only for one type of event fired by all model elements of a certain class and to register for a set of types of events fired by all mod elements of a certain class.

Of course you can remove your listeners from the event pump. This works with methods starting with remove instead of add.

5.1.3.3. Tips

1. Don't forget to remove your listener from the event pump if it's not interested in some event any more.

   If you do not remove it, that's gonna cost performance and it will give you a hard time to debug all the logical bugs you see in your listener.

2. When you implement your listener, it is wise to NOT DO the following:

   ```
   propertyChanged(MElementEvent event) {
         // do my thing for event type 1
         // do my thing for event type 2
         // etc.
   }
   ```

   This will cause the things that need to be done for event type 1 to be fired when event type 2 do arrive.

   This still happens at a lot of places in the code of ArgoUML, most notably in the modelChanged method of the children of FigEdgeModelElement.

5.1.3.4. Possible investigation points and improvements

Should we use our own event types?

Should we replace the `MElementListener` with `PropertyChangeListener` and `MElementEvent` with `PropertyChangeEvent`? One reason we have not done so yet is that it involves a lot of work and testing.

Should we change the implementation of the Event pump itself? Not the API but the implementation!

At the moment the event pump does not use the AWT Event Thread for dispatching events. This can make ArgoUML slow (in the perception of the user).

Use the standard data structure that Swing uses for event registration (i.e. `javax.swing.EventListenerList`). Would this be an improvement?

5.1.4. NSUML specifics

Up to version 0.18.1, ArgoUML used the NSUML model repository internally to implement the UML model. Since version 0.19.1, the NSUML implementation was replaced with the NetBeans Model Data Repository (MDR) which implements the JSR-040 Java Metadata Interface.

5.1.5. The use of IDs in MDR

All changes to the MDR repository are effectively serialized in the form of change events. The ArgoUML Model-MDR subsystem gets notified before each change with the contents of the change that is about to be made and then again after the change is made. The latter is what we propogate back to the ArgoUML application as model subsystem events.

UUIDs aren't used internally by MDR. We only maintain UUIDs because PGML requires them. MDR has two types of IDs: 1) MOF ID - managed by the repository and guaranteed unique within it for the life of the repository, and 2) xmi.id - used within a single XMI file to link various items together (type references, etc). What gets called a "UUID" is actually the MOF ID of the creating repository. We maintain an internal mapping that gets created every time a new XMI file gets read to map from this "UUID" to the current internal MOF ID.

5.1.6. How to work against the model

The Model subsystem is a set of classes that lay between the model implementation (e.g. MDR) and the rest of ArgoUML that hides the APIs of the implementation. It was originally implemented to provide the ability to switch between NSUML and MDR. This is the API classes of the Model subsystem i.e. Factories, Helpers, Event Pump (where to register for changes).

Here follows a list of how different things were done to make the transition easy. Everything within ArgoUML should access the Model subsystem through the interfaces in the `org.argouml.model` package. The NSUML or MDR and whatever other implementation we eventually come up with would provide the implementation of those interfaces.

Table 5.1. How to work against the model

What	NSUML (use only within Model subsystem)	MDR (use only within Model subsystem)	Model subsystem
Test that an Object o has a certain type	o instanceof M*modelelementtype* # boolean	???CLASSNAME??? .isInstanceOf(RefObject toTest, String className) # boolean	Model.getFacade() .isA*modelelementtype*(o) # boolean
Get a single valued model element from an Object o	((M*modelelementtype*)o) .get*property*() # model element	((RefFeatured)obj) .refGetValue(String propName) # ???Type???	Model.getFacade() .get*property*(o) # Object
Get a multi valued property from an Object o	((M*modelelementtype*)o) .get*property*() # Collection	((RefFeatured)obj) .refGetValue(String propName) # Collection	Model.getFacade() .get*property*(o) # Iterator or Collection (total confusion!)
Create a new model element of type Type:	MFactory .getDefaultFactory() .create*Type*()	???CLASSNAME??? .creatInstance(String "*Type*", List argument) # RefObject	Model .get*ModelElementDomain?*Factory .build*modelelementtype*(args) or Model .get*ModelElementDomain?*Factory .create*modelelementtype*() to create them completely empty.
Delete a model element			Model .getUmlFactory() .delete(*object*) ... but only call this function from Project.moveToTrash(Object).

What	NSUML (use only within Model subsystem)	MDR (use only within Model subsystem)	Model subsystem
Register for notification that a model element Object o has changed:	((MBase) o) .addMElementListener(MElementListener el)	((MDRChangeSource) obj) .addChangeListener(???)	Model.getPump() .addModelEventListener((PropertyChangeListener)li, Object o, String[] eventnames)
Register for notification on all model elements of a certain type Type:	Not possible!	((MDRChangeSource)obj .refClass()) .addChangeListener(???)	Model.getPump() .addModelEventListener((PropertyChangeListener)li, (Object)Model .getMetaTypes() .getMODELELEMENTTYPE(), String[] eventnames)
How do I get the model as XMI on the stream Stream:	(new XMIWriter(MModel m, Writer Stream)).gen()	new XMIWriter(???)	Handled by the Persistence subsystem.

5.1.7. How do I...?

- ...add a new model element?

  Make a parameterless build method for your model element in one of the UML Factories (for instance `CoreFactory`). Use the UML 1.4 spec to choose the correct Factory. The package structure under `org.argouml.model` follows the chapters in the UML spec so get it and read it! In the build method, create a new model element using the appropriate create method in the factory. The build method e.g. is a wrapper around the create method. For all elements there are already create methods (thanks Thierry). For some elements there are already build methods. If you need one of these elements, use the build method before you barge into building new ones. Initialize all things you need in the build method as far as they don't need other model elements. In the UML spec you can read which elements you need to initialize. See for example `buildAttribute()` for an example.

  If you need to attach other already existing model elements to your model element make a `build`*`XXXX`*`(MModelelement toattach1, ...)` method in the factory where you made the build method. Don't ever call the create methods directly. If we use the build methods we will always have initialized model elements which will make a difference concerning save/load issues for example.

  Now you probably also need to create a Property Panel and a Fig object (See Section 5.3.3.5, “Creating a new Fig (explanation 2)”).

- ...create a new create method?

  Create it in the correct factory.

- ...create a new utility method?

  Create it in the correct helper.

- ...delete a model element?

Project.moveToTrash(Object obj).

(See issue 2353.)

5.2. Critics and other cognitive tools

Purpose - to provide cognitive help for the User. This help is based on the current model that the User works with.

The Critics are located in `org.argouml.cognitive`.

The Critics is a Loadable subsystem. See Section 4.6, “Loadable subsystems”.

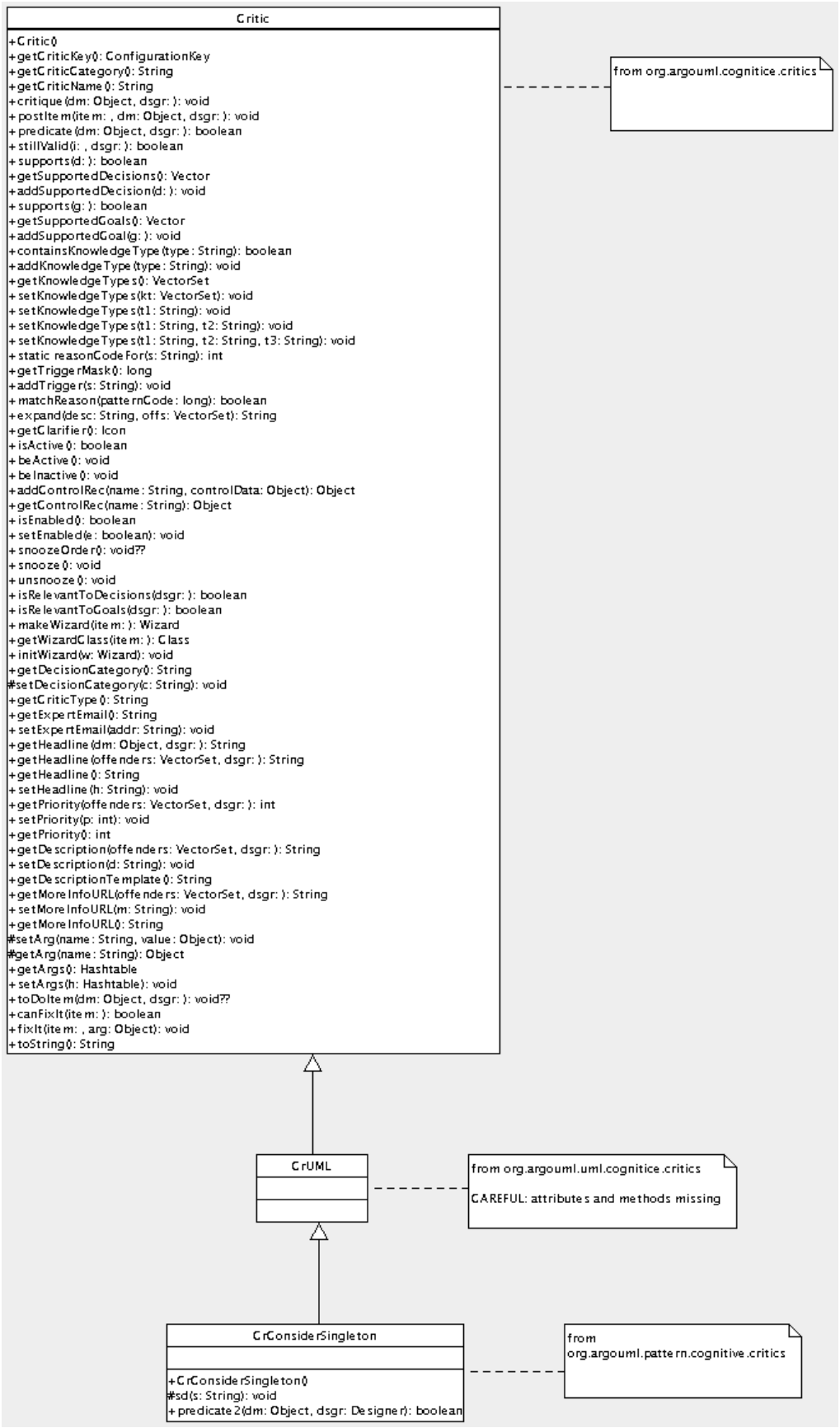
The Critics subsystem depends on the Model that it works against to take all decisions and the To Do Items used to present the information.

This subsystem contains the following main class types:

- Critics provide help to find artifacts in the model that do not obey simple design "rules" or "best practices".
- Checklists provide help for the user to suggest and keep track of considerations that the user should make for each design element. Checklists are currently (0.9.5 and 0.9.6) turned off.
- ToDoItems provide a way for the Critics to communicate their knowledge to the User and let the User start Wizards.
- Wizards are step by step instructions that fix problems found by the Critics.

5.2.1. Main classes

Here is an illustration of the main classes implementing critics


Critic
+Critic()
+getCriticKey(): ConfigurationKey
+getCriticCategory(): String
+getCriticName(): String
+critique(dm: Object, dsgr:): void
+postItem(item: , dm: Object, dsgr:): void
+predicate(dm: Object, dsgr:): boolean
+stillValid(i: , dsgr:): boolean
+supports(d:): boolean
+getSupportedDecisions(): Vector
+addSupportedDecision(d:): void
+supports(g:): boolean
+getSupportedGoals(): Vector
+addSupportedGoal(g:): void
+containsKnowledgeType(type: String): boolean
+addKnowledgeType(type: String): void
+getKnowledgeTypes(): VectorSet
+setKnowledgeTypes(kt: VectorSet): void
+setKnowledgeTypes(t1: String): void
+setKnowledgeTypes(t1: String, t2: String): void
+setKnowledgeTypes(t1: String, t2: String, t3: String): void
+static reasonCodeFor(s: String): int
+getTriggerMask(): long
+addTrigger(s: String): void
+matchReason(patternCode: long): boolean
+expand(desc: String, offs: VectorSet): String
+getClarifier(): Icon
+isActive(): boolean
+beActive(): void
+beInactive(): void
+addControlRec(name: String, controlData: Object): Object
+getControlRec(name: String): Object
+isEnabled(): boolean
+setEnabled(e: boolean): void
+snoozeOrder(): void??
+snooze(): void
+unsnooze(): void
+isRelevantToDecisions(dsgr:): boolean
+isRelevantToGoals(dsgr:): boolean
+makeWizard(item:): Wizard
+getWizardClass(item:): Class
+initWizard(w: Wizard): void
+getDecisionCategory(): String
#setDecisionCategory(c: String): void
+getCriticType(): String
+getExpertEmail(): String
+setExpertEmail(addr: String): void
+getHeadline(dm: Object, dsgr:): String
+getHeadline(offenders: VectorSet, dsgr:): String
+getHeadline(): String
+setHeadline(h: String): void
+getPriority(offenders: VectorSet, dsgr:): int
+setPriority(p: int): void
+getPriority(): int
+getDescription(offenders: VectorSet, dsgr:): String
+setDescription(d: String): void
+getDescriptionTemplate(): String
+getMoreInfoURL(offenders: VectorSet, dsgr:): String
+setMoreInfoURL(m: String): void
+getMoreInfoURL(): String
#setArg(name: String, value: Object): void
#getArg(name: String): Object
+getArgs(): Hashtable
+setArgs(h: Hashtable): void
+toDoItem(dm: Object, dsgr:): void??
+canFixIt(item:): boolean
+fixIt(item: , arg: Object): void
+toString(): String
from org.argouml.cognitice.critics
CrUML
from org.argouml.uml.cognitice.critics
CAREFUL: attributes and methods missing
CrConsiderSingleton
+CrConsiderSingleton()
#sd(s: String): void
+predicate2(dm: Object, dsgr: Designer): boolean
from
org.argouml.patte rn.cognitive.critics

Critics are currently located in:

- org.argouml.cognitive.critics

  These are basic critics, which are very general in nature. For example ArgoUML keeps nagging when Model elements overlap, which makes the Diagram hard to read.

  This package also contains the base classes for the handling.

- org.argouml.uml.cognitive.critics

  These are Critics which are directly related to UML issues (well, more or less). For example, it will nag when a class has too many operations, because that makes it hard to maintain the particular class.

  This package also contains Wizards used by these Critiques.

- org.argouml.pattern.cognitive.critics

  These are critics related to patterns. Currently they deal only with the Singleton pattern

- org.argouml.language.java.cognitive.critics

  These are critics which deal with java specific issues. Currently this is only a warning against modeling multiple inheritance.

The Base class for Wizards is `org.argouml.kernel.Wizard`.

Checklists are located in the package org.argouml.cognitive.checklist.

Helper classes for To Do Items, To Do Pane, Wizards and the Knowledge Types are located in the package org.argouml.cognitive.ui.

5.2.2. How do I ...?

- ...create a new critique?

  Currently the only way to add a new critique is to write a class that implements it so that is described here. There have however been ideas on a possibility to build critics in some other way in the future, as a set of rules instead of java code.

  Create a new critic class, of the form `Cr`*`XxxxYyyyZzzz`* , extending `CrUML`. Typically your new class will go in the package `org.argouml.uml.cognitive.critics`, but it could go in one of the other `cognitive.critics` packages.

  Write a constructor, which takes no arguments and calls the following methods of `CrUML`:

  - **setHeadline("");** to set up the locale specific text for the critic headline (the one liner that appears in the to-do pane) and the critic description (the detailed explanation that appears in the to-do tab of the details pane). The String parameter is ignored.
  - **addSupportedDecision(UMLDecision.dec*AAAA*);** where *AAAA* is the design issue category this critic falls into (examples include STORAGE, PATTERN METHODS).
  - **setPriority(ToDoItem.*BBB*_PRIORITY);** where *BBB* is one of `LOW`, `MEDIUM` or `HIGH`, to set

the priority for the critic in the to-do pane.

- **addTrigger("*UML Meta-Class*");** where *UML Meta-Class* is a UML Meta-Class, with initial lower capital, e.g. "associationEnd". The intention is that critics should only trigger for elements (or children) of particular UML meta-classes. I (Jeremy Bennett February 2002) believe this code is not yet working so you can probably leave it out. You can have multiple calls to this method for different meta-classes.

After this add a method **public boolean predicate2(Object dm, Designer dsgr);**. This is the decision routine for the critic. *dm* is the UML entity (a Model subsystem object) that is being checked. The second argument, *dsgr* is for future development and can be ignored. The `Critic` class conveniently defines two boolean constants `NO_PROBLEM` and `PROBLEM_FOUND` to be returned according to whether the object is OK, or triggers the critic.

dm may be any UML object, so the first action is to see if it is an artifact of interest and if not return `NO_PROBLEM`.

The remaining code should examine *dm* and return `NO_PROBLEM` or `PROBLEM_FOUND` as appropriate.

Having written the code you need to add the text for the headline and description to the cognitive resource bundles. These are in the package `org.argouml.i18n`, in the file `critics.properties`. You need to add two keys for the head and description, which will be named respectively `critics.`*`CrXxxxxYyyyZzzz`*`-head` and `critics`*`CrXxxxxYyyyZzzz`*`-desc`. There are plenty of examples to look at there. The other files for British English, Spanish, ... respectively) are the responsibility of the corresponding language team. Notify the language teams that there is work to be done.

In method `Init` of the class `org.argouml.uml.cognitive.critics.Init`, add two statements:

```
  private static Critic crXxxxxYyyyZzzz = new CrXxxxxYyyyZzzz();
...
  Agency.register(crXxxxxYyyyZzzz, DesignMaterialCls);
```

If you want to add a critic to a design material which is not already declared (for example the Extend class), you will have to add a third statement to the `Init` method as well, which is:

```
  java.lang.Class XxxYyyyZzCls = MXxxYyyyZzImpl.class;
```

where *MXxxYyyyZzImpl*.class should be part of the NovoSoft UML package.

Finally you should get a new section added to the user manual reference section on critics. The purpose of this is to collect all the details and rationale around this critic to complement the short text in the description. It should go in the ref_critics.xml file and have an id tag named critics.*CrXxxYyyyZzzz*.

- ...write the test in a critique?

The critiques tests are essentially a combination of conditions that are to be fulfilled. The conditions are often simple tests on simple model elements.

The class `org.argouml.cognitive.critics.CriticUtils` contains static routines that are commonly needed when writing `predicate2` (for example to test if a class has a constructor). If you find you are writing code that may be of wider use than just your critic, you should add it to `CriticUtils` rather than putting it in your critic.

For commented examples to copy, look at `org.argouml.pattern.cognitive.critics.CrConsiderSingleton`, `org.argouml.pattern.cognitive.critics.CrSingletonViolated` and `org.argouml.uml.cognitive.critics.CrConstructorNeeded`.

- ...fix a critique?

  Locate the critique and insert some logging code. You should make sure that you understand all the implications of changes, therefore it is a good idea to see what makes the critic nag in the first place. But rest assured: some of the critics haven't been updated to reflect the latest changes in ArgoUML, so this is a procedure which is well worth digging into, since it gives you also some exposure to related UML elements.

- ...change the text of a critique?

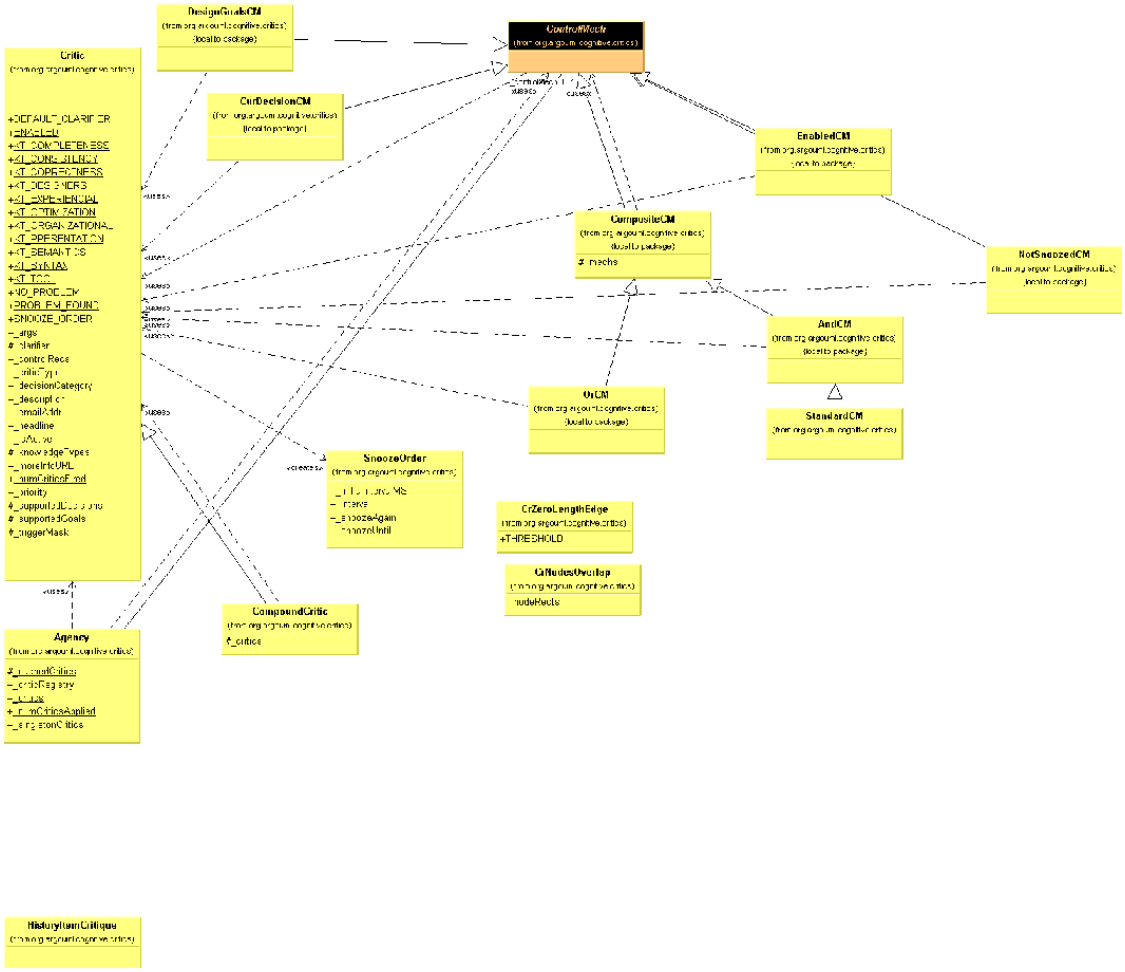
  The texts of the critics should be in the according localization files and resource bundles. Be careful: in some critics the text is still in the critic, but if you change that, you will notice that it doesn't have any effect.

- ...get my critic to trigger?

  This is a suggested way to troubleshoot if the critic doesn't trigger.

  1. Check that the settings for critics are enabled (Toggle Auto Critique)
  2. Check that your critic is registered in `org.argouml.uml.cognitive.critics.Init` with the right class (e.g. check inheritance structure against UML spec)
  3. Check that your particular critic is enabled in Browse Critics
  4. (IMPORTANT) Check that the design material is actually found in `org.argouml.uml.cognitive.critics.ChildGenUML`. This method is incomplete and might not find all model elements!

5.2.3. org.argouml.cognitive.critics.* class diagram



5.3. Diagrams

Purpose - To present the diagrams to the user and allow the user to manipulate the diagrams through the view.

The Diagrams are be located in `org.argouml.uml.diagram`.

The Diagrams is a View subsystem. Section 4.5, “View and Control subsystems”.

The Diagrams are depending on the Model subsystem and the GUI.

The classes in this subsystem are extensions of the GEF base classes (GraphModels, Figs, Selections etc) together with some supporting classes.

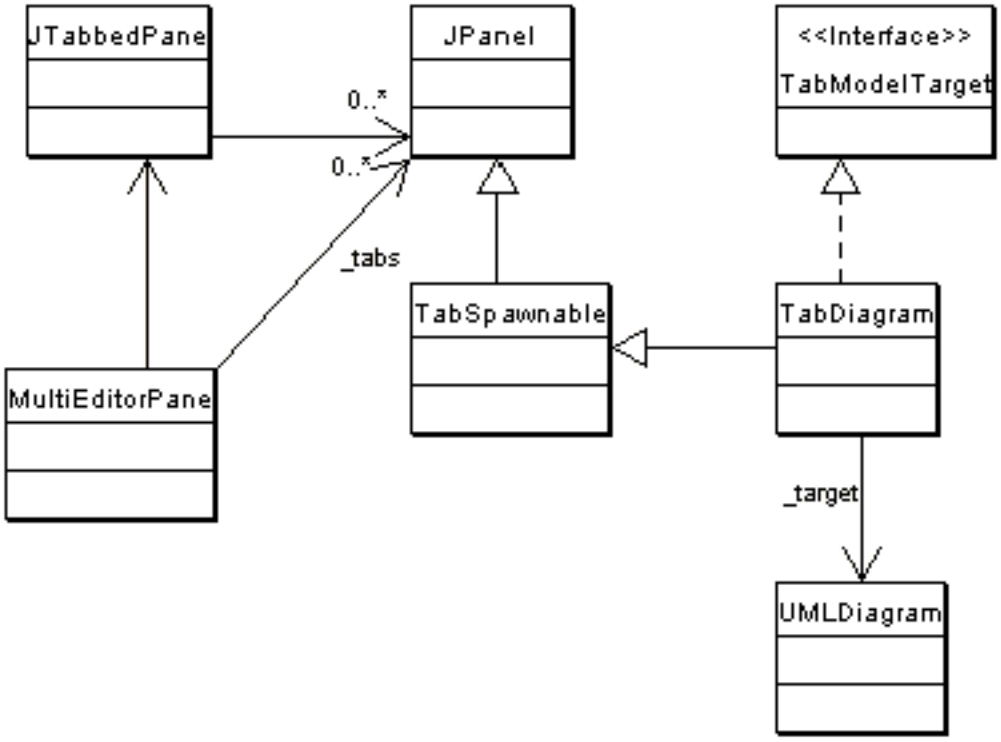
This subsystem has no direct access to a specific implementation of the OMG model repository. However it does update such a repository via the interface of the Model Subsystem.

There is an intention (Bob Tarling) to split this subsystem into several smaller subsystems, one for each diagram type. This is to allow for indiviual diagram reuse by other applications and to allow us to fast track developers onto a specific subproject containing that subsystem (Michael MacDonald and sequence diagrams in mind).

5.3.1. Multi editor pane

The multi editor pane is the pane with the diagram editor in it. Normally it is placed in the upper right corner of the application. One of the feature requests is to make the pane dockable so maybe it won't be there in the future.

The multi editor pane consists of tabs that hold editors as you can see in the class diagram.



At the moment there is only one editor tab in place. This is the `TabDiagram` that shows an UMLDiagram, the target.

The `TabDiagram` is spawn-able. This means that the user can double click the tab and the diagram will spawn as a separate window.

The target of the `MultiEditorPane` is set via the `setTarget` method of the pane. This method is called by the `setTarget` method of the `ProjectBrowser`. The pane's `setTarget` method will call each `setTarget` method of each tab that is an instance of `TabModelTarget`. Besides setting the target of the tabs, the `setTarget` method also calls `MultiEditorPane`.select(Object o). This selects the new target on a tab. This probably belongs in the `setTarget` method of the individual tabs and diagrams but that's how it's implemented at the moment.

5.3.1.1. How do I ...?

- ...add a new tab to the `MultiEditorPane`?

  Create a new class that's a child of `JPanel` and put the following line in argo.ini:

  ```
  multi:  fully classified name of new tab class
  ```

5.3.2. How do I add a new element to a diagram?

To add a new element to a diagram, two main things have to be done.

1. Create new Fig classes to represent the element on the diagram and add them to the graph model (org.argouml.uml.diagram.xxxx.XxxxDiagramGraphModel.java) and renderer (org.argouml.uml.diagram.xxxx.ui.XxxxDiagramRenderer.java).

2. Create a new property panel class that will be displayed in the property tab window on the details pane. This is described in Section 5.4, “Property panels”.

Throughout we shall use the example of adding the UML Extend relationship to a use case diagram. This allows two Use Cases to be joined by a dotted arrow labeled «`extend`» to show that one extends the behavior of the other.

The classes involved in this particular example have all been well commented and have full Javadoc descriptions, to help when examining the code. You will need to read the description here in conjunction with looking at the code.

5.3.3. How to add a new Fig

The new item must be added to the tool-bar. Both the graph model and diagram renderer for the diagram will need modifying for any new fig object.

5.3.3.1. Adding to the tool-bar of the diagram

Find the diagram object in `uml/diagram/`*`XXXX`*`/ui/UML`*`YYYY`*`Diagram.java`, where *`XXXX`* is the diagram type (lower case) and *`YYYY`* the diagram type (bumpy caps). For example `uml/diagram/use_case/ui/UMLUseCaseDiagram.java`. This will be a subclass of UMLDiagram (in `uml/diagram/ui/UMLDiagram.java`).

Each tool-bar action is declared as a private static field of class `Action`, initiated as a `new CmdCreateNode` (for nodal UML elements) or a `new CmdSetMode` (for behavior, or creation of line UML elements). These classes are part of the GEF library.

The common ones (select, broom, graphic annotations) are inherited from UMLDiagram, the diagram specific ones in the class itself. For example in UMLUseCaseDiagram.java we have the following for creating Use Case nodes.

```
private static Action actionUseCase;
```

which is then created as follows:

```
    protected Action getActionUseCase() {
        if (actionUseCase == null) {
            actionUseCase = new RadioAction(new CmdCreateNode(
                    Model.getMetaTypes().getUseCase(), "button.new-usecase"));
        }
        return actionUseCase;
    }
```

The first argument of the CmdCreateNode() function is the class of the node to create from the Model subsystem Metatypes interface, the second a textual tool tip.

The tool-bar is actually created by defining a method, `UMLDiagram.initToolBar()` which adds the tools in turn to the tool-bar (a private member named `toolBar`).

The default constructor for the diagram is declared public, despite that it must not be called directly - it is needed for loading projects from files. The desired constructor takes a name-space as an argument, and sets up a graph model (`UseCaseDiagramGraphModel`), layer perspective and renderer (`UseCaseDigramRenderer`) for nodes and edges.

5.3.3.2. Changing the graph model

The graph model is the bridge between the UML meta-model representation of the design and the graph model of GEF. They are found in the parent directory of the corresponding diagram class, and have the general name *YYYY*`DiagramGraphModel.java`, where *YYYY* is the diagram name in bumpy caps. For example the use case diagram graph model is in `uml/diagram/use_case/UseCaseDiagramGraphModel.java`

The graph model is defined as `UMLMutableGraphSupport`, a child of the GEF class `MutableGraphSupport`, and should implement `MutableGraphModel` (GEF).

5.3.3.3. Changing the renderer

The renderer is responsible for creating graphic figs as required on the diagram. It is found in the same directory of the corresponding diagram class, and has the general name *YYYY*`DiagramRenderer.java`, where *YYYY* is the diagram name in bumpy caps. For example the use case diagram graph model is in `uml/diagram/use_case/ui/UseCaseDiagramRenderer.java`

This provides two routines, `getFigNodeFor()`, which provides a fig object to represent a given UML node object and `getFigEdgeFor()`, which provides a fig object to represent a given UML edge object.

In our example, we must extend `getFigEdgeFor()` so it can handle UML Extend objects (producing a `FigExtend`).

5.3.3.4. Creating a new Fig (explanation 1)

New objects that are to appear on a diagram will require new Fig classes to represent them. In our example we have created FigExtend. They are placed in the same directory as the diagram that uses them.

The implementation of a Fig must provide constructors for both a generic fig, (i.e. without setting the owner - which is used for loading from files) and one representing a specific Model subsystem object. It should provide a setFig() method to set a particular figure as the representation. It should provide a method canEdit() to indicate whether the Fig can be edited. It should provide an event handler modelChanged() to cope with advice that the UML model has changed.

5.3.3.5. Creating a new Fig (explanation 2)

Assuming you have your model element already defined in the model and your PropPanel for that model element you should make the Fig class.

1. For nodes, that are Figs that are enclosed figures like `FigClass`, extend from `FigNodeModelElement`. For edges, that are lines like `FigAssociation`, extend from `FigEdgeModelElement`. The name of the Fig has to start with (yes indeed) Fig. The rest of the name should be equal to the model element name.

2. Create a default constructor in the Fig. In this default constructor the drawing of the actual figure is done. Here you draw the lines and text fields. See `FigClass` and `FigAssociation` for an example of this.

3. Create a constructor `FigMyModelelement(Object owner)`. Set the owner in this method by calling `setOwner`. Make a method `setOwner` that overrides it's super. Let the method call it's super. Set all attributes of the Fig with data from it's owner in this `setOwner` method. See `setOwner` of `FigAssociation` for an example.

4. Create an overridden method `protected void modelChanged(PropertyChangeEvent e)`. This method must be called (and is if you implement the fig correctly) if the owner changes. In this method you update the fig if the model is changed. See `FigAssociation` and `FigClass` for an example.

5. If you have text that can be edited, override the method `textEditStarted(FigText text)` and `textEdited(FigText text)`. This latter method is called AFTER the user edited the text. In this method the edited text is parsed. If the parsing is simple and not Notation specific, just do it in textEdited. But for most cases: use the Notation subsystem.

6. Make an Action that can be called from the GUI. If you are lucky, you just can use `CmdCreateNode`. See for examples `UMLClassDiagram` of using `CmdCreateNode`.

7. Adapt the method `canAddEdge(Object o)` on subclasses of `GraphModel` if you are building an edge so it will return true if the edge may be added to the subclass. Subclasses are for example `ClassDiagramGraphModel` and `UseCaseDiagramGraphModel`. If you are building a node, adapt `canAddNode(Object o)`.

8. Adapt the method `getFigEdgeFor` on implementors of `GraphEdgeRenderer` if you are implementing an edge so it will return the correct `FigEdge` for your object. If you are implementing a node, adapt the method `getFigNodeFor` on implementors of `GraphNodeRenderer`. In ArgoUML classes like `ClassDiagramRenderer` implement these interfaces.

9. Add an image file for the buttons to the resource directory `org/argouml/Images`. This image file must be of GIF format and have a drawing of the button image to be used in itself. This image is also used on the PropPanel. The name of the Image file should be *`model element`*`.gif`

10. Add buttons to the action you created on those places in the GUI that have a need for it. This should be at least the button bar in each diagram where you can draw your model element. Probably the parent of your model element (e.g. class in case of operation) will want a button too, so add it to the PropPanel of the parent. In case of the diagrams, add it in `UML`*`diagram`*`.java`, so in `UMLClassDiagram` if it belongs there. In case of the `PropPanel`s, use `UndoableAction`.

5.4. Property panels

Purpose - to provide a form view of the diagrams and objects in the model. The contents of the model is modifiable.

The Property panels will be located in `org.argouml.uml.?`.

The Property panels is a View subsystem. See Section 4.5, “View and Control subsystems”.

The PropPanels for the diagrams are in `org.argouml.uml.diagram.ui` and the property panels for UML objects are in `org.argouml.uml.ui.`*`UML path`* .

5.4.1. Adding the property panel

Property Panels for UML model elements are found as class `PropPanel`*`XXX`*`.java`, where *XXX* is the UML meta-class. They are in sub-packages of org.argouml.uml.ui corresponding to the *XXX* UML packages, which in turn correspond to their section in the chapter 2 of the UML 1.3 spec.

So for our example we create a new class `PropPanelExtend` in package `org.argouml.uml.ui.behavior.use_cases`.

Any associated classes that do not fall into the UML classification are provided in org.argouml.uml.ui.

Typically the constructor for the new proppanel class invokes the parent constructor, and then builds the fields required on the property tab. The parent constructor may need an icon. If you need a new icon, a call to `lookupIcon()` should be made (note that this is a utility method of the parent PropPanel class). For our example we had to add `Extend.gif`.

You will need to make an icon, in .gif format, 16 X 16 pixels, with the transparent background color set to white. Place this file in the org.argouml.Images directory (it must be named like Name.gif). This icon will automatically be used in the toolbar and in the Navigation pane.

Finally the property panel must be added to the list of property panels in the `run()` method of the `TabProps` class, with a new call of `panels.put()`. If you don't do this, navigation listeners won't know about it!

The content of the property panel is created as a grid with columns (1 column if there are only a few fields, 2 or 3 if there are more). Each row of each column contains a caption (i.e. label) and its corresponding field.

A caption and its field may be added with one of a small number of utility methods which shield you from the layout stuff: addField() and addSeperator().

A button may be added to the toolbar with the utility method addButton().

Every field is built from Java Swing components. However these are extended by ArgoUML to help in the provision of action methods for fields in the property tab. Several fields involve lists, and these require in addition list models to compute the members of the list.

The fields that you might add to a property panel include:

- Simple editable text. For example the Name field. Supported through the `UMLTextField2` class.
- A drop down box (aka combobox) of options that can be selected. Supported by the `UMLComboBox2` class. Used e.g. for the type of a parameter.
- A check box. This one does not use a seperate model class, thanks to the simplicity of the represented boolean value. Supported by the `UMLCheckBox2` class. Used e.g. for the concurrency checkbox on a composite state.
- A radio button. These always come in a group. Supported by the `UMLRadioButtonPanel` class. Used e.g. for selecting the visibility on the properties panel of a class.
- A list. Used e.g. for the Generalizations field on the proppanel of a class. The non-editable list is supported by the `UMLList2` class and its child `UMLLinkedList`. The latter also exists in the form of `UMLMutableLinkedList`, which allows adding, creation and deleting elements by popup menu. Used e.g. for the subvertex list for a composite state.

  The list model is usually provided by a sub-class of `UMLModelElementListModel2`. There is a variant `UMLModelElementOrderedListModel2` intended for ordered links, which adds a few items to the pop-up menu, allowing sorting. This latter model is used e.g. for attributes of a class.

- A drop down box of options that can be selected. This one exists in several versions, each having different possibilities. The most simple version is the `UMLComboBox2`.

  The `UMLEditableComboBox` allows editing the selected item.

  The `UMLSearchableComboBox` allows editing the selected item. See e.g. the Operation combobox on the callevent properties panel.

  Then there is a variant with a seperate button for navigation to the property panel for the currently selected item. This is supported by the`UMLComboBoxNavigator` class. Used e.g. for the stereotype field.

- An editable multiline text area. Supported by the `UMLTextArea2` class. Used e.g. for the text field of a UML Comment.

Examples of these fields in more detail follow below.

5.4.1.1. Adding a simple list field

For example we need to add a field to the use case property panel for the extends relationships that derive from this use case.

This field consists of a label and a scrollable pane (`JScrollPane`) containing the list (`JList`), which may be empty, or contain extend relationships from this use case.

Rather than a straight `JList`, we use its child, `UMLLinkedList`, which adds several features to the standard JList specifically for ArgoUML's properties panels.

The constructor for `UMLLinkedList` requires two arguments, a list model and a flag to indicate whether to show an icon.

The list model should be a subclass of `UMLModelElementListModel2`, a subclass of the Swing `DefaultListModel` which implements `AbstractListModel`. The `UMLModelElementListModel2` implements two interfaces: one that listens to target changes, and one that listens to UML model changes.

5.4.1.1.1. The list model

In our example we create `UMLUseCaseExtendListModel`. Its constructor takes no arguments. However, we need to provide the parent class with a Model subsystem event name by invoking the constructor of the parent class, with the event name as parameter.

A string naming an Model subsystem event that should force a refresh of the list model. A null value will cause all events to trigger a refresh. The name of the event is the same as the name of the associated attribute or association end from the UML 1.4 metamodel.

This list model should then be provided with a number of methods. The following are mandatory, since they are declared abstract in the parent.

`protected void buildModelList()`
: (Re)Builds the list of elements. Called from targetChanged every time the target of the proppanel is changed.

`protected boolean isValidElement(Object/*MBase*/ o)`
: Returns true if the given element is valid, i.e. it may be added to the list of elements. This function is called for many UML elements, to determine if it fits in the list. Remark: The indication /

MBase/ is a remainder from the time that ArgoUML included direct references to the NSUML model all over the code. Now it is a practical reminder of what we are dealing with.

Warning

The following description is old and the property panels have undergone some fundamental changes since it was written. It would be good if someone that knows how it works now could write a description on how it works now.

The following are sometimes provided as an override of the parent, although for many uses the default is fine.

`public void open(int index)`
Perform the action associated with the “open” pop-up menu on the element at the given index. The default provided in the parent just navigates to that element.

`public boolean buildPopup(JPopupMenu popup, int index)`
Build a pop-up menu for the list and return whether it should be displayed. Any actions will be associated with the item at the given index in the list. This is built using `UMLListMenuItem`, which can record the index, rather than plain `JListItem`. The default provides open, add, delete, move up and move down, with add disabled if there are already as many elements as the upper bound (if any) for the list, open and delete disabled if there are no elements and move up and move down disabled if they cannot be invoked on the given element. The default implementation always returns true.

The following should be declared as needed to support particular pop-up functions.

`public void add(int index)`
Perform the actions associated with the “add” pop-up menu on the element at the given index. There is no default provided, so this must be given if the “add” operation is supported. The `addAtUtil()` method (see below) may prove helpful.

In this routine you may create a new Model subsytem entity. The best way to do this is using a buildXXX method from the appropriate factory so that the appropriate initialization gets done, but you can also use a createXXX method and set it up (don't forget e.g namespace etc) yourself. Remember also to change anything that references the newly created entity.

Warning

NOTE: The following was written regarding NSUML. It may be generally true for the Model subsystem, but this hasn't been verified.

NSUML routines generally set up the “other” end of a relationship automatically if you set up one end. If you try to do both (on a NxM relationship) you will probably end up doing it twice. If you do encounter this, the rule of thumb is to explicitly set the ordered end (if you do it the other way round, NSUML will assume you mean the "other" end to be at the end of its ordered list).

`public void delete(int index)`
Perform the actions associated with the “delete” pop-up menu on the element at the given index. There is no default provided, so this must be given if the “delete” operation is supported.

`public void moveUp(int index)`

Perform the actions associated with the “move up” pop-up menu on the element at the given index. There is no default provided, so this must be given if the “move up” operation is supported.

`public void moveDown(int index)`
Perform the actions associated with the “move down” pop-up menu on the element at the given index. There is no default provided, so this must be given if the “move down” operation is supported.

The following normally use the default method, but may be declared to override methods in the parent

`public void resetSize()`
Called when an external event may have changed the size of the list. The default just sets a flag, which will ensure recalcModelElementSize (see above) is invoked as needed.

`public Object formatElement(MModelElement element)`
Return an object (invariably a String) that represents an element. The default provided in the parent defers this to the container, which in turn defers it to the current profile. This is usually perfectly satisfactory.

`public void targetChanged()`
Called when the number of elements in the displayed list (including “none”) may have changed. Default invokes the necessary Swing operations to advise of a change in list size.

`public void targetReasserted()`
Called when the navigation history has been changed (and navigation buttons may need changing). Not clear why anything is needed, but default recomputes the list size, and invokes the necessary Swing operations.

`public void roleAdded(final MElementEvent event)`

Warning
This describes the old event interface. It needs to be updated.

part of the NSUML EventListener interface. Called when an add event happens, i.e. some Model subsystem object has been added. The default provided looks to see if the event is the role name we declared, or we are listening to all events, and if so looks to see if it relates to an element in our list. If so Swing is notified that the element has been added.

`public void roleRemoved(final MElementEvent event)`

Warning
This describes the old event interface. It needs to be updated.

part of the NSUML EventListener interface. Called when a remove event happens, i.e. some Model subsystem object has been removed. The default provided looks to see if the event is the role name we declared, or we are listening to all events, and if so looks to see if it relates to an element in our list. If so Swing is notified that the element has been removed.

`public void recovered(final MElementEvent p1)` , `public void listRoleItemSet(final MElementEvent p1)` , `public void removed(final MElementEvent p1)` , `public void propertySet(final MElementEvent p1)`

Warning
This describes the old event interface. It needs to be updated.

these are all required as part of the NSUML EventListener interface, which is not well documented. In each case the default implementation recomputes the size, and advises Swing that the entire list has changed. Needs more investigation.

`public void navigateTo(MModelElement modelElement)`
a request to navigate to the specified object as part of the NavigationListener interface. The default in the parent just invokes navigateTo() on the container (ultimately PropPanel).

The following utility routines are also provided in the parent. They are not normally overridden.

`public int getUpperBound()`
get any upper bound (-1 is used if there is none).

`public void setUpperBound(int newBound)`
set the upper bound (-1 is used if there is none).

`public final String getProperty()`
returns the Model subsystem event name being monitored (null if all are being monitored).

`protected final int getModelElementSize()`
returns the number of elements in the list. Invokes `recalcModelElementSize()` (see above) if necessary.

`final Object getTarget()`
returns the Model subsystem object associated with the container (some child of `PropPanel` usually) that holds this list model.

`final UMLUserInterfaceContainer getContainer()`
returns the the container (some child of `PropPanel` usually) that holds this list model.

`public int getSize()`
returns the size of the list. Including if there are no elements in the model, but the list has a default text when empty.

`public Object getElementAt(int index)`
returns the element at the given index in the list.

`static protected Collection addAtUtil(Collection oldCollection, MModelElement newItem, int index)`
helps in writing the “add” function. newItem is added at the specified index in the given oldCollection.

`static protected java.util.List moveUpUtil(Collection oldCollection, int index)`
helps in writing the “move up” function. Swaps the elements at offsets index and index-1. Not clear why it doesn't return a Collection.

`static protected java.util.List moveDownUtil(Collection oldCollection, int index)`
helps in writing the “move down” function. Swaps the elements at offsets index and index-1. Not clear why it doesn't return a Collection.

`static protected MModelElement elementAtUtil(Collection collection, int index, Class requiredClass)`
helps in writing the `getElementAt()`. Finds the element at a specific index. The last argument is ignored!

5.4.1.2. Building the field

By convention the background of the list is set to the same as the background of the PropPanel and the foreground to Color.blue.

The list is then added to a `JScrollPane`. Although ArgoUML has historically not used scrollbars (`JScrollPane.VERTICAL_SCROLLBAR_NEVER` and `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`), it is more helpful to permit at least a vertical scrollbar where needed (`JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED` and `JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED`).

Finally the inherited method `addCaption()` is used to add the label for the field and `addField()` to add the associated scroll pane.

The second argument of each of these identifies the index of the caption/field pair in the vertical column of the grid for this property panel. The third argument identifies the column index. The final argument is a vertical weighting to expand the field if there is room in the property tab. This is usually set to the same non-zero value for all fields and corresponding captions that can have multiple entries, so they expand equally. If none of the fields should expand, the caption only of the last field in each column should be given a non-zero value.

5.4.1.3. Adding Property Tab Tool-bar Buttons

These are added by creating new instances of `PropPanelButton` (you don't need to assign them to anything - just creating will do). This has six arguments.

- The container, i.e this property panel (usually just use `this`).
- The panel for the buttons. Use `buttonPanel` which is inherited from `PropPanel`.
- The icon. Lots of these are already defined in `PropPanel`.
- The advisory text for the button. Use `localize(string)` to ensure international portability.
- The name of the method to invoke when this button is used. Some of the standard ones (e.g for navigation) are provided, but you will need to write any specials.
- The name of the method (if any) to invoke to see if this button should be enabled. Use `null` if the button should always be enabled.

In our example, the extend property panel has a “add extension point” button, with a method `newExtensionPoint` that we provide to create a new use case.

5.4.1.4. Support for stereotypes

The PropPanel should override the following (note the spelling of the method name).

`protected boolean isAcceptibleBaseMetaClass(String baseClass)`. Returns `true` if the given base class is a class of the target in the `PropPanel`.

This is used to determine what stereotypes may be shown for this property panel.

5.4.1.5. Other sorts of fields

Another sort of field that may be useful is the ComboBox. This is useful to allow users to select from a

pre-defined list of alongside a navigation arrow to go to the selected entry.

For example this is used to provide drop-down lists for the base and extension use cases of an Extend relationship in PropPanelExtend.

The model behind the drop down is created by using `UMLComboBoxModel`: `UMLComboBoxModel(container, predicate, event, getter, setter, allowVoid, baseClass, useModel)`.

The container is the `PropPanel` where we are setting up this ComboBox, the predicate is the name of a public method in that PropPanel that, given a model element, determines if it should be in the drop down, the event is the Model subsystem event name we are looking for (see earlier for the list), `getter` is the name of a public method in the PropPanel that yields the current entry in the combo Box (of type `baseClass`), `setter` (with a single argument of type `baseClass`) sets that entry, `allowVoid` if `true` will allow an empty entry for the box, `baseClass` is the UML metaclass from which all entries must descend, `useModel` is `true` to consider all the elements in the standard profile model for inclusion (so the Java types, standard stereotypes etc.).

For our `PropPanelExtend`, we provide a predicate routine the call for the “base” field is:

```
UMLComboBoxModel(this, "isAcceptableUseCase", "base", "getBase", "setBase", true, MUseCase.class, true);
```

and we define the methods `isAcceptableUseCase`, `getBase` and `setBase` in `PropPanelExtend`.

5.4.1.6. How UMLTextField works

This information is provided by Jaap Branderhorst (September 2002).

`UMLTextField` implements several kinds of event listeners:

- `MMelementListener`
- `DocumentListener`
- `FocusListener`

Furthermore it is a `UMLUserInterfaceComponent`.

Since it is an `UMLUserInterfaceComponent` it must implement `targetChanged` and `targetReasserted`. `TargetChanged` is called every time the `UMLTextField` is selected. `targetReasserted` is of no interest for `UMLTextField`. It plays a role in keeping history but since history is not really implemented at the moment in ArgoUML it is of no interest. `targetChanged` does two things:

- It calls the `targetChanged` method of the `UMLTextProperty` this `UMLTextfield` is showing.
- It calls the `update` method. The `update` method is described further on.

Besides `UMLUserInterfaceComponent` there are several other interfaces of interest. One of them is `MMElementListener`.

Every time a `MModelElement` is changed this will fire an `MEvent` to `UMLChangeDispatch`. `UMLChangeDispatch` will dispatch these events to all containers implementing `UMLUserInter-`

faceComponents interested in this event, including UMLTextField. It will also dispatch the event to all children of an interested container implementing UMLUserInterfaceComponent. By this it is only necessary to register a PropPanel which holds an UMLTextField at UMLChangeDispatch to dispatch the event to the UMLTextField too. MMelementListener knows several methods of which only one is of interest to UMLTextFields:

- propertySet

  Called every time a property in a MModelElement is set. This method calls update too if the UMLTextProperty really is affected.

Furthermore UMLTextField implements DocumentListener. This is very typical for UMLTextField. At the moment it is not possible to change the style of the text in the UMLTextField. Therefore the method changedUpdate does not have a body. This method is only called when a DocumentEvent occurs that changes the style/layout of the text. The methods insertUpdate and removeUpdate are respectively called when a character is added to the document UMLTextField contains or removed. Since both methods are called when there is true user input and when the contents of the document are changed programmatically, the methods distinguish between them. InsertUpdate and removeUpdate are both handled via the protected method handleEvent. HandleEvent updates the property in UMLTextProperty if it is really changed. If the update comes via user input, it is checked if it is valid input. If it is not, a JOptionPane is shown with ' a warning and the change is not committed into the model. If it is not via user input, the input is not checked and the property is set. If the property is set, the update method is called.

The implementation of FocusListener makes sure that the checking of user input only happens when focus is lost. Otherwise, it would not be possible to enter 'intermediate' values that are not legal. For instance, say the value class is not legal. Without the implementation of FocusListener, it would not be possible to enter class diagram since handleEvent would pop-up a warning message box.

The method update updates both the actual JTextfield as the diagram as soon as some property is set. The updating of the diagram is done by calling the damage method of the figs that represent the property on the diagram.

5.5. Persistence

Purpose - To package and unpackage the persistence data from different subsystems to and from some storage medium.

The Persistence subsystem is located in org.argouml.persistence.

Currently the storage medium is a flat file (.uml - xml format) or a zipped file (.zargo - zip format)

During save the persistence subsystem requests each subsystem for its persistence data and adds that data to output it is collating.

During load the persistence subsystem unwraps the persistence data and passes these to the relevant subsystems for those subsystems to build themselves.

Edges and nodes are now saved and loaded in z order. Previously, edges were connected as they were loaded, and if the node had not already been read then this was a problem. We now load all nodes and edges (unconnected) and then connect the edges post load.

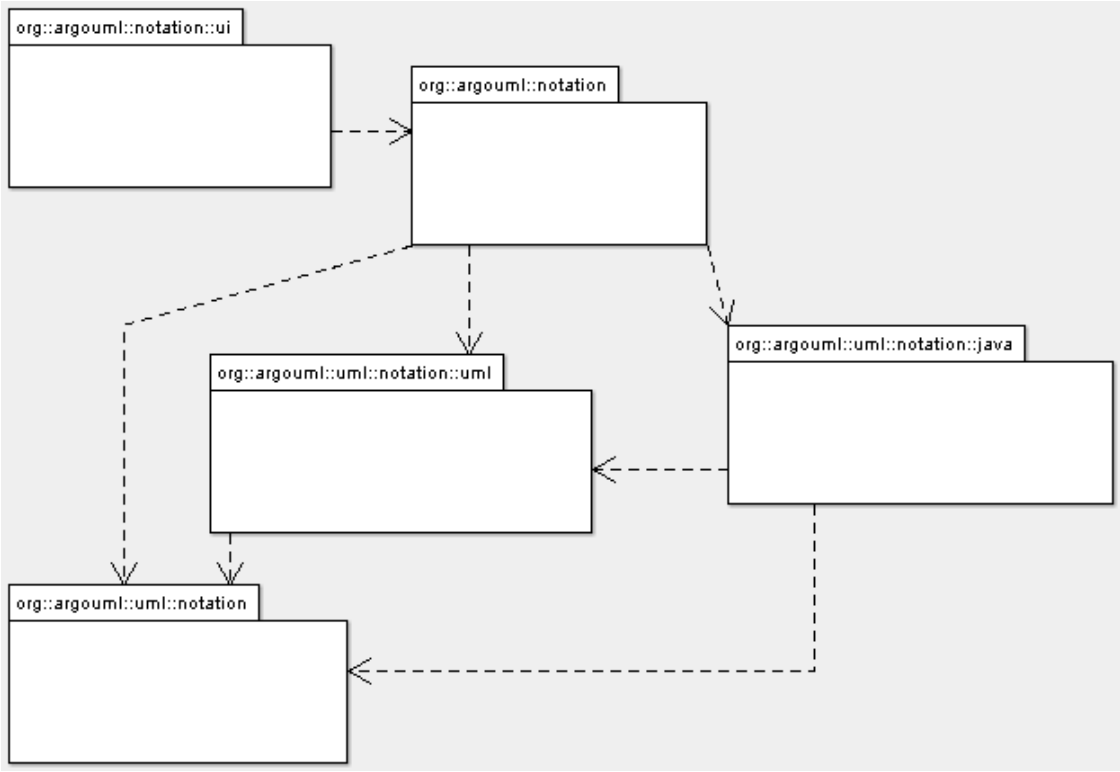
5.6. Notation

Purpose - To handle generating, updating and editing in different languages of a textual representation which represents one or more UML modelelements. Such a notation element gets e.g. attached to a dia-

gram figure or an explorer entry.

The notation subsystem is located in `org.argouml.notation` and `org.argouml.uml.notation` and their sub-folders.

Figure 5.1. Notation subsystem part 1.



In the scope of issue 3140, the following has been decided about the notation architecture:

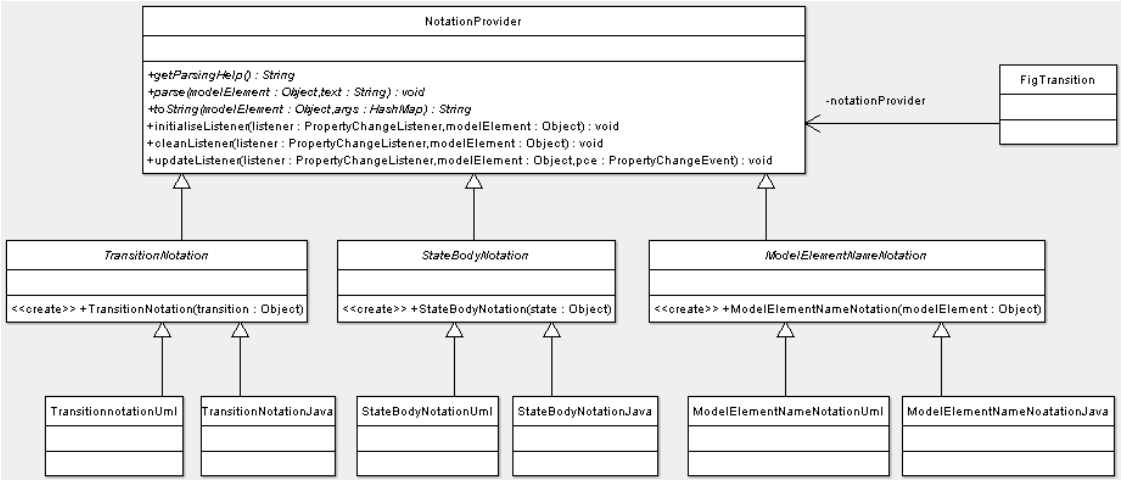
We decide that we support other languages than UML to show in diagrams, but this shall be a user-choice with project scope - and we will never refine the scope to something more detailed. Hence we need not store the notationlanguage per Fig. So, this means that you can set the notation lanuage in the project properties, and in the application-wide defaults.

The interface NotationProvider is at the center of the subsystem. There is an object implementing the NotationProvider interface per string (i.e. textual model representation) that is shown on the diagram: e.g. TransitionNotation, StateBodyNotation, ModelElementNameNotation. A state will show the latter 2.

This notation object would keep track of which UML objects it represents, generate and parse and provide a help text.

The Fig refreshes the text by calling the toString() method of the Notationprovider, at initialisation time and whenever the model changes. This latter works as follows: The notation object defines the set of model change events that the Fig listens at. Putting this task into this Notation class (instead of the Fig) has the advantage that the knowledge to which uml objects to listen is centralised where it is needed, and not any more in the Fig. The Fig then receives a PropertyChangeEvent when the model changes. This causes the Notationprovider to update the set of listeners, and the Fig refreshes the rendering of its string.

Figure 5.2. Notation subsystem part 2.

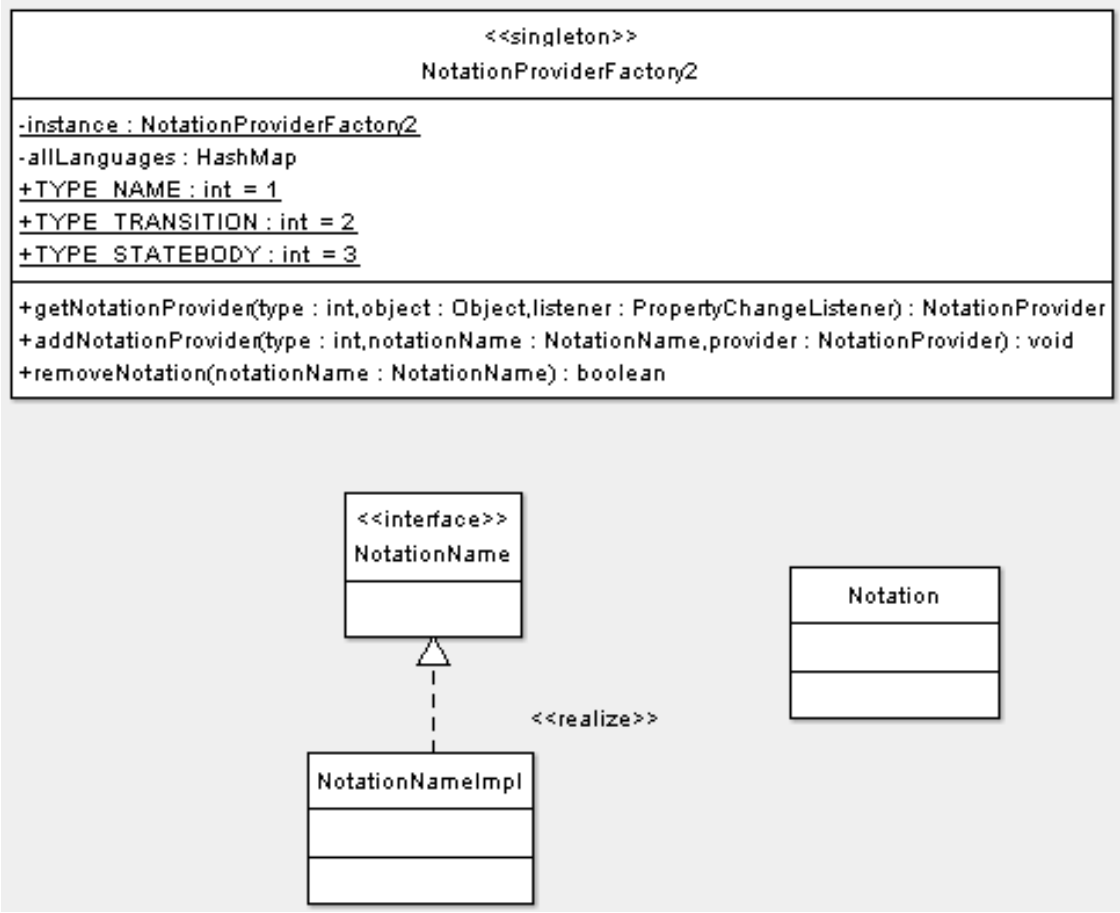


The NotationProvider objects TransitionNotation, StateBodyNotation, etc. are abstract, since they are specialised in classes that implement them for a certain language, i.e. one per language. So, we will have: TransitionNotationUML, TransitionNotationJava, ... etc.

The NotationProviderFactory2 is a singleton, since it is the accesspoint for all Figs to create the textual representation of modelobjects, and since plugin modules can add extra languages.

The NotationProviderFactory2 creates a notation object on request for any Fig, based on the current notation language selected by the user for the current project. Hence, when the user changes the language (in the menu), then it is the Fig's responsibility to listen to the NotationEvent (hence the Fig shall implement ArgoNotationEventListener), and ask the NotationProviderFactory2 to create a new NotationProvider child.

Figure 5.3. Notation subsystem part 3.



5.7. Reverse Engineering Subsystem

Purpose: Point where the different languages register that they know how to do reverse engineering and common reverse engineering functions for all languages.

The Reverse Engineering is located in `org.argouml.uml.reveng`.

The Reverse Engineering Subsystem is a Control subsystem. See Section 4.5, “View and Control subsystems”.

5.8. Code Generation Subsystem

Purpose: Point where the different languages register that they know how to do code generation and common functions for all languages.

The Code Generation is located in `org.argouml.language`.

The Code Generation subsystem is a Control subsystem. See Section 4.5, “View and Control subsystems”.

The following describes the plan, as of April 2004, as outlined by by Linus Tolke as to how to isolate the NSUML references (it may need to be updated to reflect what was actually done):

The different languages or notations supplied with ArgoUML are found in sub-packages of {@link

org.argouml.language}.

Any definition or foundation interfaces are found in the directory `org.argouml.language`. Any helper classes such as abstract implementation classes are also found in `org.argouml.language`.

At boot time, each language registers their interfaces in the `org.argouml.language.Language` register.

- Languages that generates a Notation implement the `NotationGenerator` interface.
- Languages that edits or parses the Notation implement the `NotationEditor` interface.
- Languages that generates Code templates implement the `CodeGenerator` interface.
- Languages that reverse engineer Code implement the `CodeReverseEngineer` interface.

Full MDA implementations of languages is not currently discussed. I (Linus April 2004) does not understand how it is supposed to work.

5.9. Java - Code generations and Reverse Engineering

Purpose - two purposes: to allow the model to be converted into java code and updated either in java or in the model; to allow some java code to be converted into a model.

The java things are located in `org.argouml.language.java`.

The Java subsystem is a Loadable subsystem. See Section 4.6, “Loadable subsystems”.

5.9.1. How do I ...?

...

5.9.2. Which sources are involved?

The package org.argouml.uml.reveng is supposed to hold those classes that are common to all reverse engineering (RE) packages. At the moment this is the Import class which is mainly responsible to recognize directories, get their content and parse every known source file in them. These are only Java files at the moment, but there might be other languages like C++ in the future. With this concept you could mix several languages within a project. The DiagramInterface is then used to visualize generated Model subsystem objects.

The package org.argouml.uml.reveng.java holds the Java specific parts of the current RE code. C++ RE might go to org.argouml.uml.reveng.cc, or so...

5.9.3. How is the grammar of the target language implemented?

It's an Antlr (http://www.antlr.org [http://www.antlr.org]) grammar, based on the Antlr Java parser example. The main difference is the missing AST (Abstract Syntax Tree) generation and tree-parser. So the original example generates an AST (a treelike data structure) and then traverses this tree, while the ArgoUML code parses the source file and generates Model subsystem objects directly from the sources.

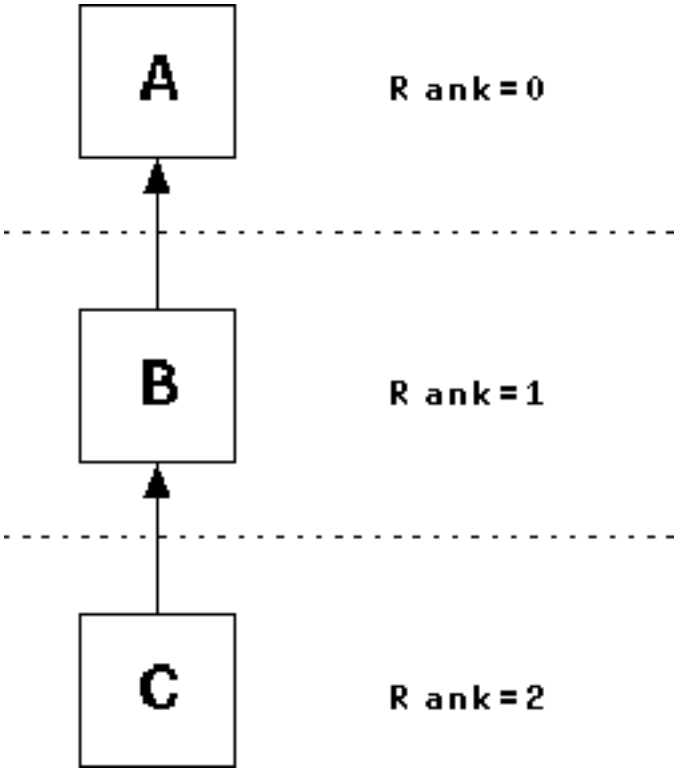
This was done to avoid the memory usage of an AST and the frequent GC while parsing many source files.

5.9.4. Which model/diagram elements are generated?

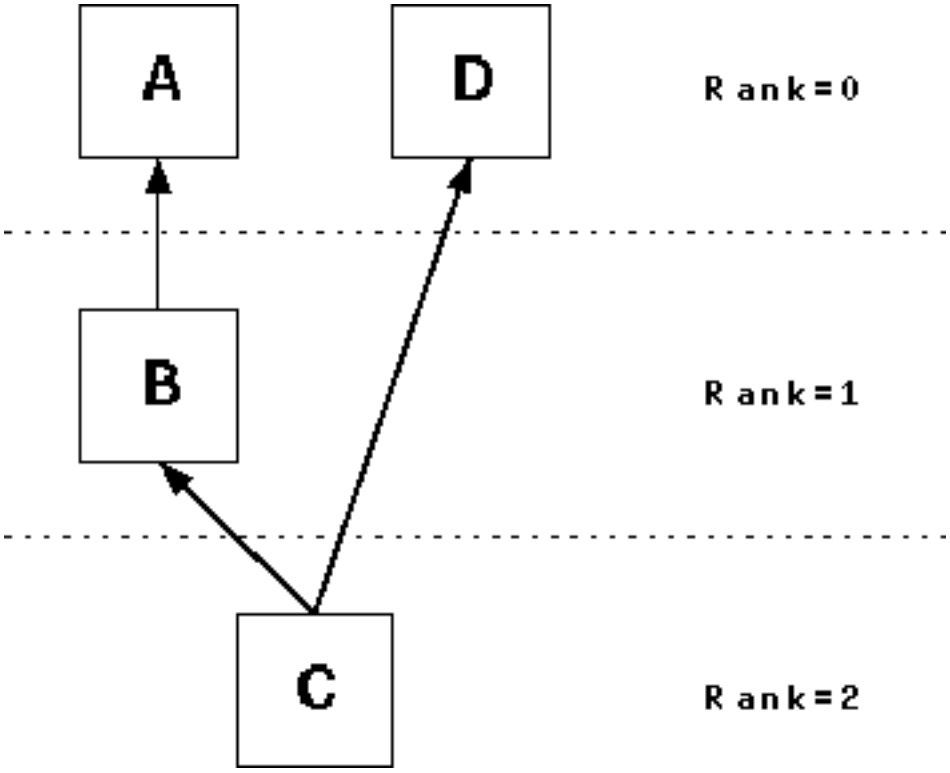
The *context classes hold the current context for a package, class etc. When the required information for an object is available, the corresponding Model subsystem object is created and passed to the DiagramInterface to visualize it.

5.9.5. Which layout algorithm is used?

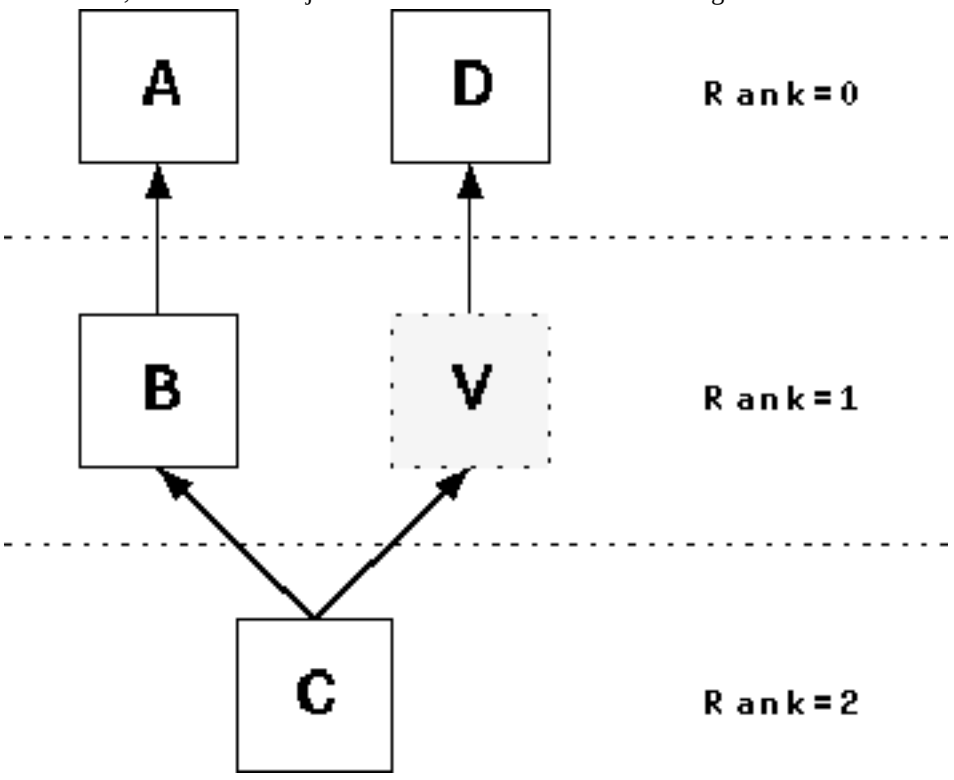
The classes in org.argouml.uml.diagram.static_structure.layout.* hold the Class diagram layout code. No layout for other diagram types yet. It's based on a ranking scheme for classes and interfaces. The rank of a class/interface depends on the total number of (direct or indirect) super-classes. So if class B extends A (with rank(A)=0), then rank(B)=1. If C extends B, then rank(C)=2 since it has 2 super-classes A,B. An implemented interface is treated similar to a extended class. The objects are placed in rows then, that depend on their rank. rank(0)=1st row. rank(1) =2nd row (below the 1st one) etc. Example:



In the next diagram, a link goes to an object that is not in the row above:



In this case, insert virtual objects which are linked to the actual target and link to them:



The objects are sorted within their row then to minimize crossing links between them. Compute the average value of the vertical positions of all linked objects in the row above. Example: we have 2 ranks, 0 and 1, with 3 classes each:

A B C : rank 0

D E F : rank 1

We give the super-classes an index in their rank (assuming that they are already sorted):

A:0, B:1, C:2

D, E, F have the following links (A, B, C could be interfaces, so I allow links to multiple super-classes here):

D -> C

E -> A and C

F -> A and B

Compute the average value of the indexes:

D = 2 (C has index 2 / 1 link)

E = 0 + 2 / 2 = 1 (A=0, C=2 divide by 2 links)

F = 0 + 1 / 2 = 0.5 (A=0, B=1, 2 links)

Then sort the subclasses by that value:

F(is 0.5), E(is 1), D(is 2)

So the placement is:

A B C

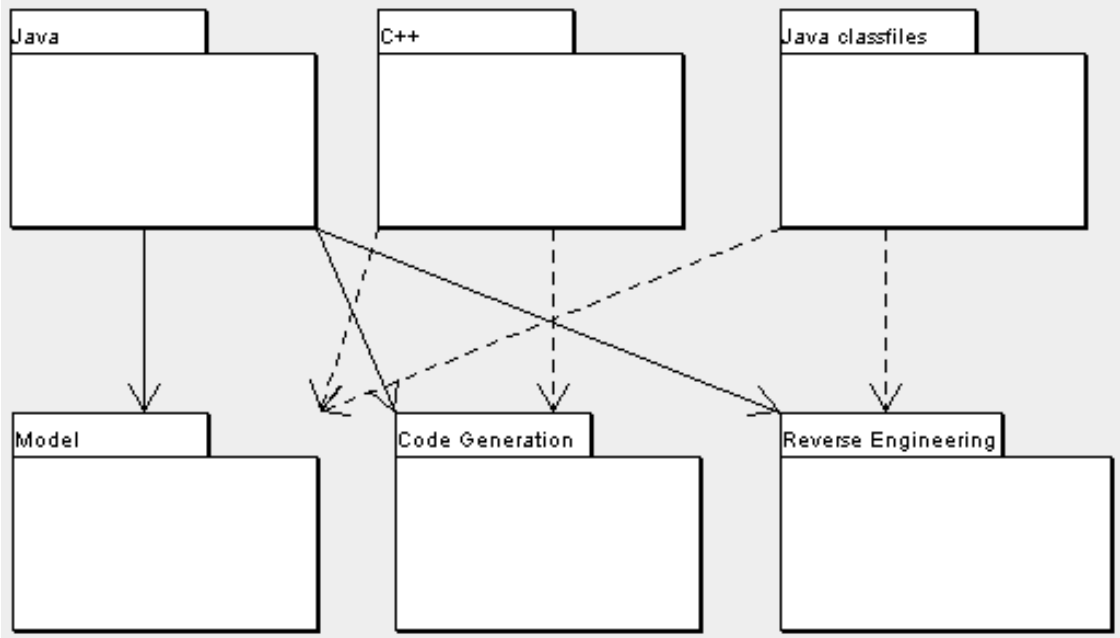
(here are the links, but I can hardly paint them as ASCIIs)

F E D

5.10. Other languages

Each other language supported by ArgoUML has its own subsystem. They are each different in level of support and implementation language.

Currently C++ has no reverse engineering but only code generation (and a very simple one at that). Java class files has only reverse engineering.



5.11. The GUI

Purpose - Provide an infrastructure with menus, tabs and panes available for the other subsystems to fill with actions and contents.

This subsystem has no knowledge of UML, Critics, Diagrams, or Model.

The GUI Framework is located in `org.argouml.ui`.

This is implemented directly on top of Swing and Java2.

The GUI framework provides the following options

- The menu with actions
- The tool-bar with actions
- The Explorer (formerly called Navigator)

  Located in `org.argouml.ui.explorer`. Contains the tree structure with configurable perspectives.
- Tabbed pane

  Could contain several different panes.
- The TargetManager

  Located in `org.argouml.ui.targetmanager`.

  Thanks to the architecture of ArgoUML of Modelelements and Figs, one rule has been decided upon (by mvw@tigris.org): *The list of targets shall not contain any Fig that has an owner*. Instead, the owner is enlisted.
- The Settings Tab

The GUI subsystem does not contain any knowledge of what is going to be put into the different panes but it has knowledge of parts of the semantics of those components.

The components that wants to be placed into any of these register with the GUI subsystem using the appropriate method in `org.argouml.ui.GUI`.

5.12. Application

Purpose - to provide the entry point when starting ArgoUML. Responsibility to start the ball rolling.

The Application is located in `org.argouml.application`.

The entry point is called `org.argouml.application.Main`.

5.12.1. What is loaded/initialized?

It all begins in org.argouml.application.Main: set up main application frame (org.argouml.ui.ProjectBrowser), the project (org.argouml.kernel.Project), numerous classes, and finally as a background thread: cognitive support (org.argouml.cognitive.Designer) and some more classes.

The ProjectBrowser initializes the menu, tool-bar, status bar and the four main areas: navigation pane (org.argouml.ui.NavigatorPane), editor pane (org.argouml.ui.MultiEditorPane), to do pane (org.argouml.cognitive.ui.ToDoPane), and details pane (org.argouml.ui.DetailsPane). Then, the actual project is set to either a read from file project (see ArgoParser.SINGLETON.readProject(URL) and ArgoParser.SINGLETON.getProject() in org.argouml.xml.argo.ArgoParser) or a newly generated project (see Project.makeEmptyProject()).

5.12.2. Details pane

Currently (May 2003) the Details pane contains several tabs: Property Panels (See Section 5.4, “Property panels”, Critics explanations and wizards (belonging to the Critics subsystem) (See Section 5.2, “Critics and other cognitive tools”), Documentation, Style, Source, Constraints (an OCL constraints of the current object) (See Section 5.20, “OCL”), and Tagged values.

Warning

It is not clear in what subsystem Documentation, Style, Source, and Tagged values belong.

5.12.2.1. How do I ...?

- ...add a tab in the Details Panel?

  Create your `TabXXX` class in `org.argouml.uml.ui` by copying from another `TabYYY.java` (e.g. `TabSrc`, `TabStyle`). Then register your `TabXXX` in `org/argouml/argo.ini` by adding a line giving the compass point to place the tab. Like -

  ```
  south:          TabXXX
  ```

- ...remove a tab from the Details Panel?

  Remove the line for the tab from `org/argouml/argo.ini`.

5.13. Help System

Purpose - to provide the menu actions that start the help and other documentation. To provide infrastructure that makes context sensitive help possible.

The Help System is not yet implemented.

The Help System will be located in `org.argouml.help`.

The Help System is a Model subsystem. See Section 4.4, “Model subsystems”.

Javahelp or some other help function will probably be used.

5.14. Internationalization

Purpose - to provide the infrastructure that the other subsystems can use to translate strings; to provide the infrastructure that makes it possible to plug in new languages; to administer the default language; to administer all supported languages.

When ArgoUML starts it starts with the language given by the first decisive language information of

1. Command line argument

   The prepared Java Web Start alternatives also uses this to override everything else.

2. The users' saved configuration

3. The default locale for the users' computer

4. ArgoUML default language (Enlish U.S.)

This is done independently of if ArgoUML has that language prepared or not.

Example 5.1. Starting with nonexisting language.

If ArgoUML is started with Swahili on the command line (that doesn't exist), while the user has French (that does exist) configured, ArgoUML will work in Swahili and all languages will show up in English U.S. i.e. the default language because Swahili doesn't exist.

The Internationalization is located in `org.argouml.i18n` in the class path. In the checked out copy of ArgoUML it is located partly in `argouml/src_new/org/argouml/i18n` - functionality, and non-localized default language (US English), and partly in `argouml-`*`language`*`/src/org/argouml/i18n` - all files for a specific language (one new tigris project for each language).

For the time being, some languages are also placed in `argouml/src/i18n/`*`language`*`/src/org/argouml`

The Internationalization is an Infrastructure subsystem. See Section 4.3, “Low-level subsystems”.

In ArgoUML internationalization (sometimes called i18n) is done using the property files that are loaded into PropertyResourceBundles.

5.14.1. Organizing translators

The problems with internationalization are not so much the technical problems as to how it works but more so the problems are with getting, keeping and coordinating the correct competences to do the job. This comes from the fact that by necessity the different persons working with internationalization have different native languages and that complicates the communications.

To handle this problem for GNU applications there is a community set up around “gettext” with one language team per language working with all “gettext” applications. There are also tools to help the translator do his job delivered with “gettext” that are the same for all the applications. In each of these language teams discussions are held that ensure a consistent use of words over all these applications.

It is for me (Linus Tolke, May 2002) unclear if and how such a community exists for Open Source Java tools and ArgoUML cannot simply benefit from the “gettext” communities since we don't use “gettext” and cannot use the same tools.

To get things done, we organize our own Language Teams for ArgoUML. Each language team is actually just one or several persons that know that language and are eager to work with translating ArgoUML.

The language team has the following responsibilities:

1. Constitute the foundation of the ArgoUML community for that language.
2. Translate all localized strings and resources.

   This is a constant work with keeping up with the changes that will be made to the ArgoUML code and ArgoUML modules since ArgoUML is under fast development.
3. The terminology used shall be correct.

   This requires work in keeping up with the current literature in the domain of ArgoUML.
4. Help with the improvements on ArgoUML by pin-pointing where ArgoUML needs to be modified to allow for localization.

   As ArgoUML is originally built without localization there may still have places in the GUI that is not localizable just by modifying the resource bundles. Each such place is a Defect and shall be corrected.
5. See that the used libraries also provide their part in that language.

   This is mostly GEF since GEF is central both when it comes to the fact that it has localized strings of its own but also because it handles parts of the localization.

   This means discussing with the teams developing the underlying package as to how best to provide the localization for those parts. Either by providing localization for that team to include in the package or by having ArgoUML overriding that package in that respect.

5.14.2. Ambitions for localization

An ArgoUML subproject is created for each Language Team.

The subproject has the following that is used in building a community:

- A web site

Ideally the site should match the main ArgoUML site, page by page, with a footer on every page with flags of all the languages that that specific page exist in so that it is easy to travel between languages. Missing pages should be linked to the main ArgoUML counterpart.

Alas there is no tool support on Tigris to help in this.

- Mailing lists

  The users@argouml-*language code*.tigris.org should be for user and dev@argouml-*language code*.tigris.org for persons discussing the building of the site or the translation. Discussions on both lists should be carried out in the actual language.

5.14.3. How do I ...?

- ...fix an incorrect or missing translation?

  This is the responsibility of the language team.

  If you are a member of the language team, commit the new translation.

  If you are not, send your corrections to the correct team by mail or enter an issue in issuezilla. Each language team might want to handle this differently. Whatever way, it should is stated on the web site for that team. The language team members are the ones with the Developer role in the language project.

  If the language team does not do its work quickly enough (or well enough in your opinion), please volunteer to help them out by joining the team.

  If the language team does not respond, contact the project leader of the ArgoUML project.

  For historic reasons there are currently (June 2005) teams without projects of their own. Projects will be created when something is to be done for these languages.

- ...verify that all translations are up to date?

  There is a simple tool you can use that is developed in the argouml-gen project. Currently (June 2005) this tool is run regularly and a web page with the result published in the argouml-stats project.

- ...start a new Language Team?

  Contact the project leader of the ArgoUML project to discuss this. He will create the project and make you the first member of the project once he is convinced that you have understood the responsibilities.

  The project are argouml subprojects so they are listed at the bottom of the argouml web page.

- ...find the languages internationalization code for the language your instance of ArgoUML is attempting to run with: en, es, en_GB,...

  The one you are currently using is shown in the Versions information in the about box. Help Menu => About ArgoUML Menuitem => Version tab just after the Operating System information. Search for the text looking like this:

  ```
  Language: sh
  ```

```
Country: KR
```

This example means that you have your computer set to Swahili as spoken in Korea (I think). Notice that the *Language:* and *Country:* are localized and could appear in your language.

- ...start the translation work?

  This is only applicable for members of the language team.

  Make sure you are a Developer in the appropriate project.

  Look at the files in `org/argouml/i18n`, under `argouml/src_new` (in the argouml project).

  Translate all the values in each of these files.

  This is a lot of extremely qualified work including searching well-known literature on UML and Software Engineering in order to get the correct terms for the domain. Discuss with other UML and Software Engineering professionals with the same native language to get it right.

  Create the files with the translations and store them in `argouml-`*`language code/`*`src/org/argouml/i18n`. They will have names like: `action_`*`language code`*`.properties`, `button_`*`language code`*`.properties`, `checkbox_`*`language code`*`.properties`, `combobox_`*`language code`*`.properties`, ...

  When this is done the first iteration of the Tool translation is completed. The work will probably be more maintenance-like from here on.

- ...join an existing Language Team

  Join the dev mailing list in the correct project and apply for an Observer role in the project.

  Discuss with the Language Team in question by mailing on the dev list. They will hopefully have work prepared for you and greet you with open arms.

- ...add or modify code with localized things?

  This is only applicable for developers working with the ArgoUML Java source or some argouml module.

  1. Everywhere the user would see a string in the GUI you should localize a key.

     This means that instead of writing a string you write a call to a localizer method with a "key" ("label" or "tag") as argument and the localizer method finds the resolution of the "key" is in one of the property files. You select one of the files for you key and name the key accordingly.

     The key is a String. The key has a special syntax like this:

     ```
     word1.word2.word3
     ```

     where word1 is the same as the first part of the filename that the key resides in. Example: The key "action.about-argouml" resides in the files `action.properties` and `action_`*`language code`*`.properties`.

     You will have to call the class `org.argouml.i18n.Translator` to convert them to wherever they are used.

     This is how a real example would look like:

```
import org.argouml.i18n.Translator;
...
    String localized = Translator.localize(key);
```

2. Add your "key" and resolution in English (U.S.) in the non-localized properties file in `argouml/src_new/org/argouml/i18n` and test that the GUI looks good for the default language.

   Which property file ArgoUML will eventually use depends on the localization settings of the running ArgoUML instance. While developing you should use en_US or some language that does not have a translation so that you can work with the default language.

3. Contact all language-teams so that they can update their files.

   Currently (November 2003) there is a great confusion as to where we stand on the different translations. For this reason we can't say if any language team is up to date with the changes and served by such a contact.

4. If you have strings that are sentences where you have dynamic values like a file name, a class name, or some property to enter at a certain place, remember that all languages would not write it exactly like that. Use `MessageFormat` to build every such sentence! There is a convenience function for this in `Translator` called `messageFormat`.

Notice that if you somewhere change the meaning of a specific localized string it would be a good idea to use a new "key" for the new meaning. This will make it easier for the translation team to spot the modification.

There allegedly are tools in the java world to spot this kind of changes. Until we have the tools and processes in place to handle them it is better to rely on this simpler mechanism to guarantee correctness.

Notice also that you shouldn't localize log entries, comments, exception names, names of environment variables, and tags and tokens used in save files. This is because the development project of ArgoUML is a one-language community (en_US) and the users of ArgoUML would want to be able to run an ArgoUML localized differently with otherwise the exact same settings, loading and saving the same files, ... Also a user, changing the language, should not have his files or configuration corrupted by this change.

5.15. Logging

Purpose - to provide an api for debug log and trace messages.

The purpose of debug log and trace messages is: To provide a mechanism that allows the developer to enable output of minor events focused on a specific problem area and to follow what is going on inside ArgoUML.

The Logging is located in `org.argouml.???`

The Logging is a Layer 0 subsystem.

Logging is currently implemented using log4j.

ArgoUML uses the standard log4j [http://jakarta.apache.org/log4j/] logging facility. The following sec-

tions deal with the current implementation in ArgoUML. By default, logging is turned off and only the version information of all used libraries are shown on the console.

5.15.1. What to Log in ArgoUML

Logging entries in log4j belong to exactly *one* level.

- The FATAL level designates very severe error events that will presumably lead the application to abort. Everything known about the reasons for the abortion of the application shall be logged.
- The ERROR level designates error events that might still allow the application to continue running. Everything known about the reasons for this error condition shall be logged.
- The WARN level designates potentially harmful situations. This is if CG can't find all the information required and has to make something up.
- The INFO level designates informational messages that highlight the progress of the application at coarse-grained level. This typically involves creating modules, subsystems, and singletons, loading and saving of files, imported files, opening and closing files.
- The DEBUG Level designates fine-grained informational events that are most useful to debug an application. This could be everything happening within the application.

This list is ordered according to the priority of these logging entries i.e. if logging on level WARN is enabled for a particular class/package, all logging entries that belong to the above levels ERROR and FATAL are logged as well.

For performance reasons, it is advised to do a check before frequently passed DEBUG and INFO log4j messages (see Example 5.3, “Improving on speed/performance”). The purpose of this test is to avoid the creation of the argument.

5.15.2. How to Create Log Entries...

You should *not* use `System.out.println` in ArgoUML Java Code. The only exception of this rule is for output in non-GUI mode like to print the usage message in `Main.java`.

To make log entries from within your own classes, you just need to follow the three steps below:

1. Import the org.apache.log4j.Logger class
2. Get a Logger
3. Start Logging...

Example 5.2. For log4j version 1.2.x

```
import org.apache.log4j.Logger;
...
public class theClass {
...
    private static final Logger LOG =
        Logger.getLogger(theClass.class);

...
```

```
public void anExample() {
    LOG.debug("This is a debug message.");
    LOG.info("This is a info message.");
    LOG.warn("This is a warning.");
    LOG.error("This is an error.");
    LOG.fatal("This is fatal. The program stops now working...");
}
```

Notice that we in the ArgoUML project have decided to have all loggers private static final with a static initializer. The reason for making them private is that this reduces the coupling between classes i.e. there is no risk that one class uses some other class' Logger to do logging. The reason for making them static is that our classes are more or less all either lightweight, like a representation of an object in the model, or a singleton. For the lightweight classes, having a reference to a logger object per object is a burden and for the singleton objects it doesn't care if the logger is static or not. The reason for making this final is that it shall never be modified by the class. The reason for having a static initializer is that then all classes can do this in the same way and we don't ever risk to forgot to create the Logger.

For performance reasons, a check before the actual logging statement saves the overhead of all the concatenations, data conversions and temporary objects that would be created otherwise. Even if logging is turned off for DEBUG and/or INFO level.

Example 5.3. Improving on speed/performance

```
if (LOG.isDebugEnabled()) {
    LOG.debug("Entry number: " + i + " is " + entry[i]);
}
if (LOG.isInfoEnabled()) {
    LOG.info("Entry number: " + i + " is " + entry[i]);
}
```

Warning

Since this has a big impact also on the readability, only use it where it is really needed (like places passed several times per second or hundreds of times for every key the user presses).

For more information go to the log4j homepage at http://jakarta.apache.org/log4j [http://jakarta.apache.org/log4j/].

5.15.2.1. Reasoning around the performance issues

Most of the log statements passed in ArgoUML are passed with logging turned off. This means that the only thing log4j should do is to determine that logging is off and return. Log4j has a really quick algorithm to determine if logging is on for a certain level so that is not a problem.

The problem is instead explained by noticing the following log statement:

```
        int i;
...
```

```
LOG.debug("Entry number: " + i + " is " + entry[i]);
```

It is quite innocent looking isn't it? Well that is because the java compiler is very helpful when it comes to handling strings and will convert it to the equivalent of:

```
StringBuffer sb = new StringBuffer();
sb.append("Entry number: ");
sb.append(i);
sb.append(" is ");
sb.append(entry[i].toString());
LOG.debug(sb.toString());
```

If the entry[i] is some object with a lot of calculations when toString() is called and the logging statement is passed often some action needs to be taken. If the toString() methods are simple you are still stuck with the overhead of creating a `StringBuffer` (and a `String` from the sb.toString()-statement.

5.15.3. How to Enable Logging...

log4j uses the command line parameter *`-Dlog4j.configuration = URL`* to configure itself where URL points to the location of your log4j configuration file.

Example 5.4. Various URLs

```
org/argouml/resource/filename.lcf
                     ❶

http://localhost/shared/argouml/filename.lcf
                     ❷

file://home/username/filename.lcf
                     ❸
```

❶ Reference to a configuration file *filename.lcf* within argouml.jar.
❷ Reference to a configuration file *filename.lcf* on a remote server/localhost.
❸ Reference to a configuration file *filename.lcf* on your localmachine.

5.15.3.1. ...when running ArgoUML from the command line

There are currently two possibilities of running ArgoUML from the command line:

1. Run ArgoUML using `argouml.jar`
2. Run ArgoUML using the ant script

In the first case, the configuration file is specified directly on the command line, whereas in the latter case this parameter is specified in the `build.xml` (which in that case needs to be modified). ArgoUML is then started as usual with **./build run**.

Example 5.5. Command Line for argouml.jar

```
[localhost:~] billy% java -Dlog4j.configuration=URL -jar argouml.jar
```

Example 5.6. Modification of build.xml

```
<!-- ================================================================== -->
<!-- Run ArgoUML from compiled sources                                  -->
<!-- ================================================================== -->
<target name="run" depends="compile">
    <echo message="--- Executing ${Name} ---"/>
    <!-- Uncomment the sysproperty and change the value if you want -->
    <java classname="org.argouml.application.Main"
          fork="yes"
          classpath="${build.dest};${classpath}">
          < sysproperty key="log4j.configuration"
                        value="org/argouml/resource/filename.lcf"></sysproperty>
    </java>
</target>
```

5.15.3.2. ...when running ArgoUML from WebStart

To view the console output, the WebStart user has to set *Enable Java Console* in the Java Web-Start preferences. In the same dialog, there is also an option to save the Console Output to a file.

As you cannot provide any userspecific parameters to a WebStart Application from within WebStart, it is currently not possible to choose log4j configuration when running ArgoUML from Java Web Start.

5.15.3.3. ...when running ArgoUML from NetBeans

At the time of writing this paragraph, it is not possible to set the logging configuration file on a per project basis in NetBeans. Instead, the Global Options of [Debbuging and Execution/Execution Types/ External Execution/External Process] need to be changed.

Example 5.7. External Execution Property (Arguments)

```
-cp {filesystems}{:}{classpath}{:}{library} -Dlog4j.configuration=URL
    {classname} {arguments}
```

5.15.4. How to Customize Logging...

There are some sample configuration files provided in *org.argouml.resource*. Modify these ac-

cording to your needs. Or alternatively, you can try configLog4j [http://www.japhy.de/configLog4j] to assist yourself in creating a log4j configuration file.

5.15.5. References

- The log4j project homepage at http://jakarta.apache.org/log4j [http://jakarta.apache.org/log4j/]
- The configlog4j homepage at http://www.japhy.de/configLog4j [http://www.japhy.de/configLog4j/]

5.16. JRE with utils

Purpose - to provide the infrastructure to run everything.

The JRE is an infrastructure subsystem. See Section 4.4, “Model subsystems”. It is not distributed with ArgoUML but considered to be a precondition in the same respect as the user's host.

This is a Java3 JRE so swing and awt can be used together with reflection.

5.17. To do items

Purpose - To keep track of the To do items. Items are generated and removed automatically by the critics. They could also be created by other means.

The To do items are located in `org.argouml.?`

The To do items is a Model subsystem. See Section 4.4, “Model subsystems”.

5.18. Explorer

Purpose - to provide tree views of the model elements, diagrams and other objects. Note: the Explorer used to be called the Navigator.

The Explorer is located in `org.argouml.ui.explorer` and sub-packages.

The Explorer is a Layer 2 subsystem. See Section 4.5, “View and Control subsystems”.

5.18.1. Requirements

The Explorer must react to user and application events.

User events include

- R1: selection of a node, which must notify the other views to make the same selection.
- R2: right click on a node, which brings up a pop-up menu.
- R3: selection of another perspective in the Combo box, which must change the explorer to that perspective. A perspective provides a different view of the model that will focus on one or other part of the model.
- R4: node expansion and collapse.
- R5: It is possible to drag name-space nodes on to other name-space nodes. Dropping a name-space node onto another, will, if the destination name-space is a valid one, update the explorer and model.
- R6: sorting of nodes with a particular Ordering. [an ordering is a comparator that orders child nodes in the explorer, e.g. by name and/or type].
- R7: copy diagram to clipboard functionality for windows/java 1.4 users.

- R8: tool-tip showing node name and type.
- R9: standard multiple discontinuous selection with mouse and keyboard.
- R10: the user can configure the perspectives using a dialog. Perspectives can be added, deleted, renamed, reordered and duplicated. Perspective rules can be added and removed from a perspective. The changes are saved to the user properties. If there are user perspectives when ArgoUML starts, it loads these, otherwise it loads a default set of perspectives.

Application events include

- R11: change in selection in another view, any relevant rows to be highlighted.
- R12: the UML model changes, the tree must update to reflect additions/deletions and name changes in the model.
- R13: change of project, the tree must update. the root node should be expanded with the default diagram selected.

5.18.2. Public APIs and SPIs

The Explorer Subsystem provides/will provide the following APIs:

- API1: Addition / Removal of a Perspective from the PerspectiveManager. Status: implemented
- API2: Addition / Removal of a Perspective Rule from a Perspective. Status: implemented
- API3: Selection of the Perspective to be displayed by the Explorer. Status: not implemented
- API4: Selection of Ordering for Explorer nodes. [an Ordering is a comparator that orders child nodes in the Explorer] Status: not implemented

The Explorer Subsystem provides/will provide the following SPIs:

- SPI1: Configurable Node pop-up menu. Status: not implemented
- SPI2: New PerspectiveRules can be defined and registered with the 'library' of available rules. Status: not implemented
- SPI3: New Orderings can be defined and registered with the available orderings. [an ordering is a comparator that orders child nodes in the explorer] Status: not implemented

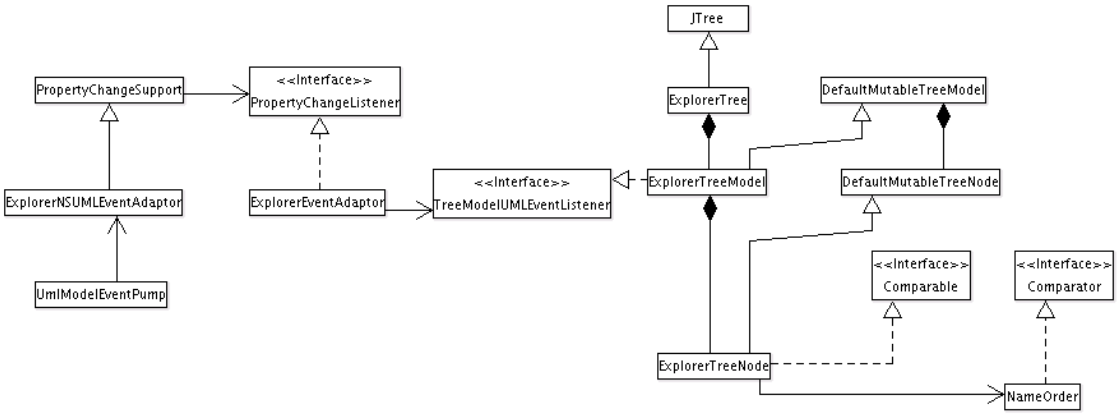
The APIs collectively represent the Explorer subsystem facade and the SPIs represent plug-ins.

5.18.3. Details of the Explorer Implementation

The Explorer is currently shown in the Explorer Pane (`org.argouml.ui.NavigatorPane`) - the upper left hand pane of ArgoUML.

Except for the Explorer Pane, The Explorer is located in org.argouml.ui.explorer.*. The explorer has been refactored since version 0.15.2 so that it has a slightly more standard Java Swing implementation.

The explorer perspectives provide the different views of the project. They are implemented by sets of PerspectiveRules that get the child nodes for any parent node in the tree.



The Explorer has 3 main subcomponents: a customized JTree, a customized TreeModel and an interface for generating child nodes in the tree which forms the tree Perspective.

1. The JTree (org.argouml.ui.explorer.ExplorerTree) has been customized to maintain consistent selection state with the other model views. It provides a pop up menu (ExplorerPopup) for performing actions on specific model elements. There is specific functionality in DnDExplorerTree for Drag and drop, and in ExportExplorer for copy diagram to clipboard.
2. The TreeModel is a customized DefaultTreeModel that listens to changes in the UML model. The JTree builds the tree model as the user expands nodes, this minimizes the size of the model to those part that the user is interested in. The TreeModel contains custom DefaultMutableTreeNodes, ExplorerTreeNodes, that maintain their own order on child nodes; this will typically be an alphabetical order on the model element names. However, it could be enhanced to include more powerful orders like total subtree size.
3. The model uses the third part of the Explorer design, PerspectiveRules, to add child nodes to the leaves of the tree. The structure of the tree is wholly dependent on the collection of PerspectiveRules that together provide a specialized view of the UML model. This is very flexible and extensible. The org.argouml.ui.explorer.rules package contains a default set of PerspectiveRules.

Each node is displayed with a name and an Icon, representing the type of node it is in the UML model. This is done using the `org.argouml.uml.ui.UMLTreeRenderer` (for the Icon), and the text is produced in the `convertValueToText(...)` method in `org.argouml.ui.explorer.ExplorerTree`.

5.18.4. How do I ...?

- ...add another perspective?

  - The perspectives can be configured using the `org.argouml.ui.explorer.PerspectiveConfigurator` by the User. The changes to the pre-defined built-in defaults are stored in the `argo.user.properties` file.

  - If you want to do this as part of an extension to ArgoUML then you should use (see above) APIs 1,2 and 3, and SPI 2. The functions needed are present in the `PerspectiveManager`.

- ...improve the PopUp menu?

  There is no way of doing this currently without modifying the core of ArgoUML. You could use SPI1 when it gets implemented.

- ...extend the Explorer in other ways?

  The best way is to use the above APIs/SPIs; if they are not implemented then it would be best to implement them and feedback your improvements to the ArgoUML project so that your code works on a recognized public API that will be maintained in the future.

- ...add new rules for new model elements?

  You should create a GoRule/PerspectiveRule in

  org/argouml/ui/explorer/rules

  . There are plenty of examples to look at. The important things to get right is of course that:

  - you return the right children
  - return the objects that the TreeModel must listen to to know when to update the node (and the list of immediate children) After that you must register your GoRule in org/argouml/ui/explorer/PerspectiveManager
  - add it to the list in loadRules()
  - perhaps add it to some of the default perspectives in oldLoadDefaultPerspectives(), I guess And then I think it should just be a matter of recompiling and possibly switching to the perspective you added your rule to.

- ...tell the explorer to refresh?

  You are not supposed to. The TreeModel is supposed to listen to events and refresh affected parts. And this is where the lack of events for adding diagrams creates a problem.

  Obviously it would be possible to add an operation somewhere to revalidate the expanded parts of the Explorer, but I'm not aware of the existence of such an operation today.

- ...navigate programmatically to a certain explorer element so that its path is exploded?

  In general you can't. The Explorer tree is lazy in that it only explores the parts of the tree that the user has opened. And since the GoRules are general navigating to them would require a complete tree search. Which is also complicated by the fact that the answer is not unique and there can be branches with infinite depth.

  In reality it would be possible to create an algorithm to search out one occurrence of an element (since the model only contains finitely many elements and I assume that no-one will add go rules that add branches of infinite length that does not infinitely often contain elements from the model), but I don't think anyone has don't it. Obviously finding all occurrences cannot be done.

5.19. Module loader

Purpose - to provide the mechanisms to load (and unload) the auxiliary modules.

The Module loader is located in `org.argouml.moduleloader`.

An old module loader is located in `org.argouml.application.modules.ModuleLoader` with interfaces (Pluggable) in `org.argouml.application.api`. Eventually this will be removed.

It is the modules' responsibility to connect and register to the subsystem or subsystems it is going to

work with using that subsystem's API, Facade, or Plug-in interface.

For details on how to build a module see Section 6.2, “Modules and PlugIns”.

5.19.1. What the ModuleLoader does

The ModuleLoader is looking for module jars. It actually scans through all jars available in the ext directory. See Edit Settings Environment tab. If you turn on logging on the debug level while running ArgoUML you should be able to see what jar files it finds and what it does with them.

A module jar contains the classes, resources and a manifest file. The manifest file points out the class to be loaded. Also notice that the Specification-Title and Vendor must be specified correctly for this to work.

5.19.2. Design of the new Module Loader

The plan is to implement this new Module Loader, then have them both working side by side for several releases (two stable releases), and if we all are happy with it, then remove the old module loader.

Design:

- We use a Loadable Proxy Pattern(?) for the modules.
- Each module can be enabled and disabled individually. Dependencies between modules is allowed although not yet handled gracefully.
- Each module is required to have one (1) class that implements `ModuleInterface`. That class (and all other classes that constitute the module) needs to be made available for some class loader, either by including it in the classpath or by letting the module loader hunt for it.
- The modules are allowed to use all the APIs available from all the subsystems within ArgoUML and from other modules.

  This is a big improvement over the old module loader in that:

  - We use the same APIs for the modules that we use within ArgoUML meaning that we implement at document it only once. This replaces the Pluggable class at every point where ArgoUML can be augmented.
  - We can have the module have different classes to register at different parts of ArgoUML.
  - We can have dynamic registrations that the module add and remove over time depending on some criteria that the module decides.
  - We don't need to search through all modules at every possible point where ArgoUML can be augmented.

  Just as in the old solution, whenever a module needs to do something to ArgoUML, there needs to be implemented an API, possibly with registration/deregistration and callbacks.
- All modules that can be found are examined at startup. They can be enabled and disabled individually from a special available modules window but have a default state that applies if the user hasn't taken action. Currently the default state is "enabled".
- Dependency between modules!

  If a module cannot be enabled because some other module needs to be enabled first or because some part of ArgoUML needs to be initialized first this is a problem. This is because the initial implement-

ation is such that we have no register of dependencies.

The solution suggested is that the module loader persists in it's attempts to enable a module so that the order among the modules is not important. For this to work the modules needs to signal when they fail. This is done by returning false or throwing a Exception from the module enabling method.

The module loader also provides an API that the well-behaving modules can use to test if the modules they depend on are enabled. The less well-behaving module can just throw an exception when they fail to enable themselves properly.

If a module cannot be disabled, because some other module depends on it then this is signaled by returning false from the disabling method.

- Where modules are loaded from?

  The modules are loaded from the same places as in the old module loader. They can be internally i.e. available in the core jar file of ArgoUML, from the `ext` directory, or if running from JavaWebStart, they can be downloaded from the site.

  To reduce the complexity of the downloads, let's use it in the simplest possible way: organize each module in a package and a jar file, have the jnlp-file list that jar file as a part and a package entry listing the classes, have a file listing optional classes and a GUI that allows the user to download them. Once a class is selected in the GUI it is loaded and, the JavaWebStart class loader will guarantee that it is available.

- The scope of the modules.

  Modules are always enabled and disabled on a per-application (per jvm) basis and not on a per-project or per-frame basis.

5.20. OCL

Purpose - To allow for editing of strings in the OCL language.

The OCL is located in `org.argouml.ocl`.

The OCL is a Layer 3 subsystem. See Section 4.6, “Loadable subsystems”.

The OCL editor GUI interface is `org.argouml.uml.ui.TabConstraints` (shown in the bottom right hand panel - details panel).

`org.argouml.ocl.ArgoFacade` adapts the `tudresden.ocl.gui.OCLEditor` for ArgoUML. There are some other helper classes in `org.argouml.ocl`, with names beginning with OCL but they are used for other purposes. Historically GEF uses OCL as a kind of template language to convert the UML diagrams to pgml(and back again), it doesn't have anything to do with OCL constraints in your UML model.

`ArgoFacade` is reused by `GeneratorJava` and `TabConstraints`.

Currently this subsystem is more or less only Dresden OCL Toolkit and adaptation.

Because of a problem with the interpretation of the UML specification and the OCL specification, the implementation of constraints in ArgoUML is only possible for Classes, Interfaces and Features (Attributes and Operations). See Issue 1805 [http://argouml.tigris.org/issues/show_bug.cgi?id=1805].

Chapter 6. Extending ArgoUML

This section explains some general concepts which come in handy, when developing additions to ArgoUML.

Warning

There are two module loading mechanisms, "the old one", and "the new one". This is so because we have made a change of the design used for this in order to simplify the writing of modules.

Eventually "the old module loader" will be removed so for all new additions, the new module loader shall be used.

6.1. How do I ...?

- ...get the according NS-UML element for a given `Fig`*`XXX`* class?

  Each `Fig`*`XXX`* implements the method `getOwner()` which returns the appropriate owner element which is responsible for this Fig element.

- ...get the according Fig element for a given `MModelElement`?

  for this one needs to iterate through all fig elements and invoke `getOwner`. Compare the result with the given `MModelElement`. Beware that there might be more than one Fig Element per `MModelElement`.

6.2. Modules and PlugIns

The ArgoUML tool provides a basis for UML design and potentially an executable architecture environment for more specialized applications. This is solved by a clear interfaces between the ArgoUML core and the extensions. Extensions are called modules.

6.2.1. Differences between modules and plugins

Note

This description is only relevant for the old moduleloader since the plugins concept is not used in the new one.

In the old moduleloader implementation the classes within the modules that attach to ArgoUML core are called plugins. In the new moduleloader implementation they don't have any special name.

- Modules

  A module is a collection of classes and resource files that can be enabled and disabled in ArgoUML. Currently this is decided by the modules' availability when ArgoUML starts but in the future it could

be made possible to enable modules from within a running ArgoUML.

This module system is the extension capability to the ArgoUML tool. It will give developers of ArgoUML and developers of applications running within the ArgoUML architecture the ability to add additional functionality to the ArgoUML environment without modifying the basic ArgoUML tool. This flexibility should encourage additional open source and/or commercial involvement with the open source UML tool.

The module extensions will load when ArgoUML starts. When the modules are loaded they have the capability of attaching to internal ArgoUML architectural elements. Once the plugins are attached, the plugins will receive calls at the right moment and can perform the correct action at that point.

Modules can be internal and external. The only difference is that the internal modules are part of the `argouml.jar` and the external are delivered as separate jar-files.

- Plugins

Note

This description is for the old moduleloader.

A plug-in in ArgoUML is a module that implements the `org.argouml.application.api.Pluggable` interface.

The `Pluggable` interface acts as a passive dynamic component, i.e. it provides methods to simplify the attaching of calls at the correct places. There are several `Pluggable` interfaces that each simplify the addition of one kind of object. Examples `PluggableMenu`, `PluggableNotation`.

One Module can implement several `Pluggable` interfaces.

This is essentially and implementation of the Dynamic Linkage pattern as described in Patterns in Java Volume 1 by Mark Grand ISBN 0-471-25839-3. The whole of ArgoUML Core is the Environment, the classes inheriting `Pluggable` are the AbstractLoadableClass.

6.2.2. Modules

6.2.2.1. Module Architecture for the old implementation

The controlling class of the module/plugin extension is `org.argouml.application.modules.ModuleLoader`. `ModuleLoader` is a singleton created in the ArgoUML main initialization routine.

`ModuleLoader` will:

- read in the property file
- for each of the classes found
  1. create the specified classes
  2. call `initializeModule` on this class
  3. place the class object into the internal list of modules

6.2.2.2. The ArgoModule interface - used in the old implementation

Each class must derive from the `ArgoModule` interface. This interface provides the following methods:

- ```
  String getModuleName (void);

  String getModuleDescription (void);

  String getModuleVersion (void);

  String getModuleAuthor (void);
  ```

  provides information about the ArgoUML module.

- ```
  boolean initializeModule (void);
  ```

  `initializeModule` is called when the class loader has created the module, and before it is added into the modules list. `initializeModule` should initialize any required data and/or attach itself as a listener to ArgoUML actions. `initializeModule` for all modules is invoked after the rest of ArgoUML has been initialized and loaded. Any menu modifications or system level resources should already be available when the module initialization process is called.

  `initializeModule` should return true if the initialization is successful (or if no initialization is necessary).

  The only available mechanism for handling dependencies between modules is the order in which they are read from the file.

- ```
  void shutdownModule (void);
  ```

  The `shutdownModule` method is called when the module is removed. It provides each module the capability to clean up or save any required information before being cleared from memory.

- ```
  void setModuleEnabled (boolean tf );

  boolean isModuleEnabled (void);
  ```

  Reserved for future implementation.

- ```
  Vector getModulePopUpActions (void);
  ```

  Reserved for future implementation.

  The plan is to have this called for each module when the module should add its entries in PopUpActions.
  ```

- ```
  String getModuleKey (void);
  ```

  Returns a string that identifies the module.

6.2.2.3. Module Architecture for the new implementation

The controlling class for the new implementation is `org.argouml.moduleloader.ModuleLoader2`. It is a singleton created when first used. It is first used in the main initialization routine.

When created it searches through all available modules and creates a list of their main objects (implementing `ModuleInterface`). Currently (September 2004) this also means that the found modules are by default selected i.e. they are marked to be enabled.

At the end of the main initialization routine the selected modules are enabled. (The original idea was to do this several times during the main routine to allow for modules to add command line arguments, add languages, and make functions available for batch command, but the example used for testing loaded the ProjectBrowser "too early" and the result wasn't so good. I (Linus) hopes this can be eventually fixed.)

6.2.2.4. The ModuleInterface interface - in the new implementation

Each class used by the `ModuleLoader2` must implement the `ModuleInterface` interface.

This interface has methods for enabling, disabling and identifying the module.

When a module is enabled it is expected to register some class wherever it affects ArgoUML using the interfaces provided there. Since the same interfaces and registration mechanism is used internally within ArgoUML there is a small likelyhood that there already is an interface and a possibility to register. If there isn't, ArgoUML cannot currently be extended at that point. If you still need ArgoUML to be extended at that point you will have to work in getting this interface or registration mechanism implemented within ArgoUML. (This could also be another module that has to be amended.)

Classes administered by the module that registers to whatever place of ArgoUML they are interested in, does not need to have any connection to the module loader. They are written exactly as if they would have been if they were part of the core ArgoUML.

6.2.2.5. Using Modules

When modules are used they can't be distinguished from the rest of the ArgoUML environment.

6.2.2.6. How do I ...?

- ...tell when a module is enabled?

  The method `isEnabled` in `ModuleLoader2` returns true if the module with that name is enabled and false otherwise.

  ### Note

  This only works for modules enabled in the new module loader. For modules loaded using the old module loader, it is not possible to determine if they are enabled.
  ```

6.2.3. Plugins

Note

This description is for the old moduleloader.

6.2.3.1. Plugin Architecture

Each class must derive from the `Pluggable` interface. In addition to the methods declared in `ArgoModule`, which `Pluggable` extends (see Section 6.2.2.2, “The ArgoModule interface - used in the old implementation”), the interface provides the following method:

- ```
  boolean inContext (Object[] context );
  ```

  `inContext` allows a plug-in to decide if it is available under a specific context.

One example of a plugin with multiple criteria is the PluggableMenu. PluggableMenu requires the first context to be a JMenuItem which wants the PluggableMenu attached to as the context, so that it can determine that it would attach to a menu. The second context is an internal (non-localized) description of the menu such as "File" or "View" so that the plugin can further decide.

6.2.3.2. How do I ...?

- ...create a pluggable settings tab?

  ...

- ...create a pluggable menu item?

  Look at the modules junit and menutest for examples of how to add to menus using the PluggableMenu interface.

  The implementation of inContext() that you provide should be similar to:

  ```
  public boolean inContext(Object[] o) {
      if (o.length < 2) return false;
      if ((o[0] instanceof JMenuItem) &&
              ("Create Diagrams".equals(o[1]))) {
          return true;
      }
      return false;
  }
  ```

  The string "Create Diagrams" is a non-localized key string passed in ProjectLoader at about line 440 in the statement

  ```
  appendPluggableMenus(_createDiagrams, "Create Diagrams");
  ```

  There is no restriction on a single class implementing multiple plugins - quite the contrary, that is
  ```

one of the reasons for providing the generic Pluggable interface that PluggableThings extend.

- ...create a pluggable notation?

  ...

- ...create a pluggable diagram?

  Let's say we want to enable a new diagram type as a plug-in. We use the interface PluggableDiagram that uses a method that returns an JMenuItem object:

```
    public JMenuItem getDiagramMenuItem();
```

  The returned menu item will be added to the diagrams menu to allow to open a new diagram of this type.

  In this example we do this by creating a helper class in the package org.argouml.application.helpers that implements the created plug-in interface PluggableDiagram, and call it DiagramHelper:

```
public abstract class DiagramHelper extends ArgoDiagram
implements PluggableDiagram {

    /** Default localization key for diagrams
     */
    public final static String DIAGRAM_BUNDLE = "DiagramType";

    /** String naming the resource bundle to use for localization.
     */
    protected String _bundle = "";

    public DiagramHelper() {
        _bundle = getDiagramResourceBundleKey();
    }

    public void setModuleEnabled(boolean v) { }

    public boolean initializeModule() { return true; }

    public boolean inContext(Object[] o) { return true; }

    public boolean isModuleEnabled() { return true; }

    public Vector getModulePopUpActions(Vector v, Object o) { return null; }

    public boolean shutdownModule() { return true; }

    public JMenuItem getDiagramMenuItem()
    {
        return new JMenuItem(Argo.localize(_bundle,"diagram_type"));
    }

    public String getDiagramResourceBundleKey() {
        return DIAGRAM_BUNDLE;
    }
}
```

  The extension of ArgoDiagram is specific to this example; the plug-in will provide a new ArgoUML diagram.

Important

Don't forget to do the localization stuff, because the plug-in might be used in all languages ArgoUML offers!

- ...do the localization stuff (not plug-in specific, but important)?

  ...

- ...create a pluggable resource bundle?

  ...

- ...create a new pluggable type?

1. Create the plug-ins interface

   In the package org.argouml.application.api, create an interface that extends `Pluggable` (in the same package). The class name must begin with 'Pluggable'.

   ## Note

   One of the main purposes of a plugin is to provide the capability to add an externally defined class that will be used by ArgoUML in the same way as a similar internal class. This means that modifications are needed all over ArgoUML in order to call the pluggable interface. Therefore this must be done in ArgoUML itself and cannot be done in any module.

   It now inherits from `ArgoModule` the methods

```
public boolean initializeModule();
public boolean shutdownModule();
public void setModuleEnabled(boolean tf);
public boolean isModuleEnabled();
public String getModuleName();
public String getModuleDescription();
public String getModuleVersion();
public String getModuleAuthor();
public Vector getModulePopUpActions(Vector popUpActions, Object context)
public String getModuleKey();
```

   and from `Pluggable` the methods

```
public boolean inContext(Object[] context);
```

and thus provides the basic mechanism that plug-ins need.

2. Decide in what context this is to be enabled and add calls there

   It is useful for those plugins which actually use context to provide a helper method

   ```
   Object[] buildContext (classtype1 parameter1 , classtype2 parameter2 );
   ```

   which will serve two purposes.

   First, it will provide a simple way of creating the Object[] parameter.

   Second, it helps to document the context parameters within the class itself.

   Again using `PluggableMenu` as an example, it contains the function

   ```
   public Object[] buildContext(JMenuItem parentMenuItem, String menuType);
   ```

   which is used as follows:

   ```
   if (module.inContext(module.buildContext(_help, "Help"))) {
       _help.add(module.getMenuItem(_help, "Help"));
   }
   ```

6.2.4. Tip for creating new modules (from Florent de Lamotte)

Note

This description is for the old moduleloader.

Florent wrote a small tutorial for creating modules. It can be found on the ArgoPNO website [http://argopno.tigris.org/documentation/argouml.html].

6.3. How are modules organized in the java code

The previous section describes how modules and plug-ins are connected on the java level totally independent of how they are actually linked into ArgoUML.

Within the ArgoUML project some parts of the code are for different reasons developed and kept separate from the main ArgoUML source code. These parts can be modules or plug-ins on the java level but on the source code level they are called modules. This section describes how they are organized and how you create such source-code modules.

6.3.1. Requirements on modules

An external module requires:

1. The module main class implements org.argouml.moduleloader.ModuleInterface
2. Archive the main class into a jar, with an entry "Name" in MANIFEST.MF specifying the name of the main class. For example, Name:

   ```
   Name: your/own/domain/your/package/your.class
   ```

   The class in question must implement `org.argouml.moduleloader.ModuleInterface`.
3. Put the jar file into a directory called "ext" under the home directory of ArgoUML
4. Run ArgoUML and check if the module appears in Edit -> Settings -> Modules

New modules that are added to ArgoUML shall reside in whole new packages. Either you put your module classes in *`your.own.domain .your.package.name`* or if you want to emphasize the connection to ArgoUML you can use `org.argouml.`*`your.package.name`* where *`your.package.name`* is the name of your addition.

6.3.2. How do I ...?

- ...create a new source-code module.

  Suggestion, copy from the `menutest` module as described here.

  Make a copy of `argouml/modules/menutest` into `argouml/modules/`*`your name`* .

  Add any jar you need to `argouml/modules/`*`your name`*` /lib` and add references to each of the jars in `argouml/modules/`*`your name`*`/build.xml`.

  Edit `argouml/modules/`*`your name`*` /module.properties`

  Edit `argouml/modules/`*`your name`*`/src/org/manifest.mf`.

  Reorganize the source files as necessary. Remove the directory `argouml/modules/`*`your name`*` /src/org/argouml/ui` and create your own classes like org.argouml.*`your package name`* in `argouml/modules/`*`your name`*` /src/org/argouml/`*`your package name`* .

- ...get Argo to use a plug-in?

  Note

  This description is for the old module loader.

  Once you've created a jar file with a plug-in in it, you need to make sure that Argo can find the jar to be able to execute it.

  If you are using a "standard" ArgoUML source structure, then you should be able to execute **build install** or **ant install** in the source directory of the plug-in. This will copy the jar file to the proper directory in the main ArgoUML build target. You can test your plug-in by running **build run** in the

`src_new` directory.

If you need to install the jar "the hard way", try the following steps.

- Start up ArgoUML.
- Go to the menu **Edit->Settings** and look at the **Environment** tab. Find the entry labeled `${argo.ext.dir}`. Create that directory if it does not already exist.
- Copy the plug-in jar and any other jars required by it into that directory.
- Start up ArgoUML again, and you should see the plug-in's startup banner (if it has one, of course).

Chapter 7. Standards for coding in ArgoUML

7.1. When Writing Java Code

The coding style for ArgoUML is based on the Code Conventions for the Java Programming Language [http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html]. We have the following exceptions and comments:

- Each file starts with some header info: file, version info, copyright notice. Like this:

```
// $Id$
// Copyright (c) 2007 The Regents of the University of California. All
// Rights Reserved. Permission to use, copy, modify, and distribute this
// software and its documentation without fee, and without a written
// agreement is hereby granted, provided that the above copyright notice
// and this paragraph appear in all copies.  This software program and
// documentation are copyrighted by The Regents of the University of
// California. The software program and documentation are supplied "AS
// IS", without any accompanying services from The Regents. The Regents
// does not warrant that the operation of the program will be
// uninterrupted or error-free. The end-user understands that the program
// was developed for research purposes and is advised not to rely
// exclusively on the program for any reason.  IN NO EVENT SHALL THE
// UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
// SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,
// ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
// THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
// SUCH DAMAGE. THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
// WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
// PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
// CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT,
// UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

package whatever;
...
```

  The file and version is maintained by subversion using keyword substitution. Remember to set the property "svn:keywords" to "Id" on all source files. The year in the copyright notice is maintained manually.

  This differs from the Sun Code Conventions that requires the initial comment to be a C-style comment.

  This is checked by Checkstyle.

- All instance variables are private.

  This is not required by the Sun Code Conventions but an additional requirement for ArgoUML.

  This is checked by Checkstyle.

- Use Javadoc for each class, instance variable, and methods that are not overriding or implementing a method from an interface or extended class. In general do not put comments in the body of a method. If you are doing something complex enough to need a comment, consider breaking it out into its own private commented method.

  If you are overriding or implementing a method and you want to describe specifics of the implementation, use the javadoc and reference the overridden or implemented method.

  This is not required by the Sun Code Conventions but an additional requirement for ArgoUML.

  This is partly checked by Checkstyle. Checkstyle does currently only warn if a Javadoc comment is omitted for a public, protected or default visibility variables.

- Indicate places of future modifications with

  ```
  // TODO: reason and explanation
  ```

  or if within a Javadoc or c-style comment

  ```
  * TODO: reason and explanation
  ```

  This differs from the Sun Code Conventions that uses either XXX or FIXME depending on if it works or not.

- Four spaces should be used as the unit of indentation. Tabs must be set exactly every 8 spaces (not 4) and represent 2 indents.

  This is exactly as it is stated in the Sun Code Conventions. It is here just for the emphasis.

  This is checked by Checkstyle.

- If possible use lines shorter than 80 characters wide.

  This is exactly as it is stated in the Sun Code Conventions. It is here just for the emphasis.

  This is checked by Checkstyle. Checkstyle ignores three kinds of lines in this check because of the historical use of long class names and package names. These are lines that contain "@see *`some method name`*", "// $Id:*`whatever`*$", and import statements.

- Open brace on same line (at end). Both for if/while/for and for class and functions definitions.

  This is exactly as it is stated in the Sun Code Conventions. It is here just for the emphasis.

- Use deprecation when removing public and protected classes, methods and attributes.

  Whenever you have a public or protected method or attribute in a class or a public class that you want to remove, change the signature in an incompatible way, or make change visibility for you shall always deprecate it first. After the next stable release you (or someone else) can remove it.

  In the future, when the subsystems are well defined and it is clear what public or protected methods, attributes or classes that are part of a certain subsystem's exported interface we can allow an exception to this rule for methods, attributes and classes that are not. (See Section 4.2, “Relationship of the subsystems”.)

  Write deprecation statements like this:

```
* @deprecated by your name in the upcoming release. Use {@link whatever}
*              a complete explanation on what to do instead
```

This is not checked by Checkstyle.

Rationale: This is part of the "Do Simple Things"-development approach that we use in ArgoUML. ArgoUML is a big project with lots of legacy code that we do not know exactly how it works. Deprecation shows the intent between decision to remove a method and the point where it is actually removed and this without breaking anything of the old code. There are also modules or plug ins that we might know nothing about that could be loaded by some user to run within ArgoUML to add functionality. It is for the modules and plug ins that we always save deprecated methods to the next stable release. It makes it possible for the module developers to do work during the unstable releases and release at the same time as ArgoUML releases its stable release.

- Don't use deprecated methods or classes.

  Rationale: Deprecation is an indication that a class is to be removed. We always want to build ArgoUML in a way that allows for future updates of everything. Using things that are on the way out already when doing the implementation is for this reason not allowed.

  Rationale 2: If you feel like you really want to use a method that is deprecated instead of the replacement you should first convince the person responsible for doing the deprecation that he has made a mistake and upgrade ArgoUML to a version of that library without that method or class deprecated. If it is within ArgoUML discuss it with the person who actually did the deprecation or in the development team.

  Comment: There is an ongoing work (probably perpetually) to change the calls to deprecated methods and classes that has been deprecated after used in ArgoUML. This is a normal part of improving ArgoUML. If this work is too slow it makes it impossible to upgrade to new versions of different sub-tools. This problem is seen by "the person responsible for sourcing of the sub-tool" when actually trying to upgrade the sub-tool. (See Section 9.8, “How to relate issues to problems in dependencies”.)

- Don't use very long package and class names.

  To make the code readable, keep class names shorter than 25 chars, and have at most four levels of packages.

  Historically in the ArgoUML design, a deep package structure has been used. There are several places in the code where the package structure is mimicking the UML hierarchy of objects resulting in impossibly long package names like `org.argouml.model.uml.behavioralelements.collaborations.`*`class name`* , and `org.argouml.uml.ui.behavior.common_behavior.`*`class name`* .

  While establishing the subsystems we use a two-level approach much like the rest of the Java world. For the subsystem API we always use: `org.argouml.`*`subsystem package name`* i.e. the classes are in the subsystem's directory and all subsystems have package names that is a single level below `org.argouml`. If a subsystem is really complex or will be complex w.r.t. the amount of classes (meaning more than 50 files with classes), we create new packages with internal classes on a single level below the subsystem package.

  This is the plan for the subsystems and new classes. Don't move old classes just yet! That would create more confusion that it would help.

- For everything else follow Code Conventions for the Java Programming Language

[http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html] (called Sun Code Conventions)!

Some of these rules are marked with a comment that they are checked by a Checkstyle. Checkstyle is a tool available with the ArgoUML development environment preconfigured for these rules. The current configuration can be found in `argouml/tools/checkstyle/checkstyle_argouml.xml`.

To run Checkstyle run the command **build checkstyle** from the `argouml/src_new` directory. This requires you to have checked out the directories `argouml/tools`, `argouml/tests`, and `argouml/src_new`.

The last couple of Checkstyle result are also available in the Xenofarm result.

Checkstyle will also check some of the rules from the Sun Code Conventions that are not stated here. Furthermore Checkstyle nags about when the order of modifiers does not conform to the suggestions in the Java Language Specification, Section 8.1.1, 8.3.1, 8.4.3.

7.2. When Committing to the Repository

The repository is a shared resource in the project. This means that once you commit your stuff it has the potential of getting in the way of everybody else's work in the project. For this reason special considerations are needed. This chapter describes the how you should do to limit the risk of causing someone else problems.

When you have done all the work, and all the testing and are about to commit something please do:

1. Compile ArgoUML (**build run** or **build package**).

   This goes for all changes, even changes in comments.

2. If your changes include removing files make a clean compile. (**build clean** followed by **build run** or **build package**).

3. If your changes include removing public or protected operations and attributes make a clean compile (**build clean** followed by **build run** or **build package**).

   The build mechanism does not yet have reliable dependency checker enabled so this is the best way to make sure.

4. If your changes include adding abstract operations make a clean compile (**build clean** followed by **build run** or **build package**).

   The build mechanism does not yet have reliable dependency checker enabled so this is the best way to make sure.

5. If you have changed anything that has the potential of affecting something in a totally different part of the code like internal data structure, handling of exceptions, run all JUnit test cases and start the tool and do some more testing.

   If in doubt, run all JUnit test cases.

6. Do a **svn status** in trunk to make sure that you do not forget to commit a file and a **svn update** to make sure that no one else has committed anything in the mean time.

   Remember that if you do not commit all the files from trunk that **svn status** found (marked A, R, and M) in the same commit then you would better remove those file from the checked out copy, update to get the original version from the repository and start over with the compilation.

If someone else have updated a file (**svn update** shown U, or R) please compile again.

7. Commit all files that are included in a change at the same time.

   This reduces the chance of anyone getting an inconsistent set of files by updating in the middle of your commit.

8. Commit often.

   Remember that the repository is also a backup copy of your work.

   If your change is so big and involves so many files that you would like to commit it for backup reasons but it doesn't compile or doesn't work or for some other reason should not confuse the trunk in the subversion repository, create a branch to work in. Then when your work is complete, you merge the branch into trunk.

Rationale: These ground rules is for the purpose of not stopping or hindering the work for anyone. Remember that there might be several developers working with different agendas and different efficiency (slower or faster) and the commits is the melting point of this.

Perspective: If this will take you an extra two minutes before every commit remember that if you commit something that will not work this will take everyone else (guess 10 persons) the extra time of looking at the compilation error or see the tool crash (1 minute), wonder why (1 minute), search for the error in his own changes (3 minutes), search for the error somewhere else (1 minute), glance at the mailing list to see if someone else has noticed this and send a mail (1 minute), wait for some response (1 hour wait), update (1 minute), compile (1 minute). This amounts to 10 hours wait and 1.5 hours extra work for all developers in the project.

7.3. When Using Branches

We use the following standards in ArgoUML:

- Developers working on code, with an unspecified due date are requested to put the code into a branch if it is deemed useful that the code can be shared. Developer branches follow the scheme: *work*_*explanation*_*owner* , where
  - *explanation* is something like javahelp, propertypanel, cppcodegeneration, issue12345
  - *owner* is the uid for the developer that started the branch, e.g. tlach (Thierry Lach) or mkl (Markus Klink).

Merging branches together is complex. Please use branches sparingly and announce your intentions on the mailing list.

7.4. When Working with the Build Process

For the `build.xml` files we use the following rules.

- Be careful when downloading stuff.

  ArgoUML is supposed to be a self-contained development environment. Some times it is better to have things downloaded from the ant script instead of from the repository. In that case separate the

download-targets from the target that does building so that it is easy for everyone to know when their development machine is working against the Internet and when it is not.

- Public targets shall have description. Non-public targets shall not have description (write xml comments or echos instead).

- Use ant-built-ins for everything.

  ArgoUML is supposed to be a self-contains development environment. If you feel tempted to use other tools (perl, sed, nsgmls), don't! They are probably not present in all environments where we want to run a development environment.

7.5. When Considering Dependencies

Linus Tolke

In the ArgoUML project we use several third-party libraries to solve parts of the problem for us. These libraries (referred to as dependencies below) are an important part of the ArgoUML tool and must be handled in a good way if ArgoUML is going to be successful.

Here is the list of things to check in the dependency and to discuss with yourself and maybe with the rest of the ArgoUML development team before considering to use it in the ArgoUML project.

- License

  We must be allowed to develop against, release with, distribute, and use the dependency indefinitely without monetary or other compensation.

  Rationale: We have no money in the ArgoUML project, we don't want to have money in the ArgoUML project. We have no organization that can enter agreements and live up to them. We don't want to require our users to enter agreements to use ArgoUML.

- Java version

  The dependency must have a policy that matches the ArgoUML project policy on Java version requirements.

  Rationale: The ambition for ArgoUML is to be a working tool for as many people as possible. Java is still under development and there are nice features available in future releases. In ArgoUML we have a plan for how to handle this. It is to always support two major releases of Java (currently JDK 1.4 and 1.5). We cannot have a dependency that restricts us in this aspect.

- Distribution

  We require the dependency to make it possible for us to take the distribution, enter it in our repository and write rules to automate the use of the dependency while developing, releasing and running ArgoUML. This automated use must be able to run without relying on access to some server and without user intervention.

  The API documentation of the dependency (assumed to be Javadoc) we can use from some web site belonging to that dependency.

  Rationale: In the ArgoUML project we want to make it as easy as possible for our users to install ArgoUML. We also want to make it as easy as possible for our developers to get their development environment working and for the release manager to prepare the releases.

- Road map

  The project developing the dependency must have a plan that fits the ArgoUML plan for the future.

  Rationale: If a dependency will soon go somewhere else i.e. stop doing what we require or stop supporting what we require, then we will soon have troubles with that dependency.

- Working project

  The project that develops the dependency should be a working project. Check that there is some person responsible for it, preferably with a team or organization backing him. Check that there is a plan for upcoming releases. Check that there is a way to report bugs and enhancement requests.

  Rationale: We don't want to rely on a dependency where there is no chance of ever getting a bug that we encounter fixed. We are also part of an ever-evolving world. Soon we want the tool to do more for us. We should then be able to wish that and eventually get that included.

Notice that we should not and don't need to do this in a passive way. We should explain to the dependency team what we want and why. Especially for dependency that we have already in ArgoUML but also for dependencies that we consider taking in. This is to increase the likelihood that they will have us in mind when planning and evolving.

Here are the steps to go through and the recommended order once the decision is taken to use the dependency in ArgoUML:

- Documentation

  Describe in the Cookbook in the appropriate subsystem section in Chapter 5, *Inside the subsystems* what part of the problem that the dependency solves and how it is used in ArgoUML.

- Javadoc

  Enter the package list file in a special directory under `argouml/lib/javadocs`. Update the list of links used when building the Javadoc. One place in `default.properties`, One or two places in `build.xml` (targets javadocs and javadocs-api).

  Test by referencing some class from the dependency, building the Javadoc, and check that the link is working.

- Repository

  Assuming that the dependency is distributed in a set of jar files, add the jar files to the `lib` directory in a versioned way together with the license file. Use filenames like: *`dependency-version`*`.jar`, and *`dependency`*`.LICENSE.txt`.

  The plan is to have each subsystem in their own directory. If the dependency in question belongs to a subsystem that is moved to a separate directory you should put it in the `lib` directory for that subsystem. See how the model-mdr dependencies are handled.

- Building

  Assuming that the dependency is distributed in a set of jar files, add the jar files to the list of files that are to be included when building ArgoUML. The files are to be copied to the argouml/build directory when performing the "package"-target. One place in `default.properties`, Four places in `build.xml` (targets init (tree places), prerequisites, package (two places), new target check.*`dependency`*), and One possibly place in `AboutBox.java` (Constructor). Notice especially that `build.xml` shall not contain any version information. Notice also that the text in `About-`

`Box.java` shall not contain anything that needs to be localized but just the dependency name, reference and possibly version.

This will take care of running tests, building releases, and building for the purpose of developing modules.

Check by having some class from the dependency loaded immediately when starting ArgoUML and start using **build run**.

See Section 9.8, “How to relate issues to problems in dependencies” for a discussion on how to handle bugs found in dependency and updates of the version of a dependency.

Chapter 8. Writing Documentation in the ArgoUML Project

8.1. Introduction

The documentation (currently manual, cookbook, and quickguide) is written using DocBook XML V4.1.2 [http://www.oasis-open.org/docbook]. This section covers some conventions for use of DocBook and for the documentation in general. It also includes some information for tooling configuration, e.g. for Emacs with the psgml package.

8.2. Style

- "We" in the documents means the persons reading the document. For the Quick-guide and User Manual this means the user using ArgoUML. For the Cookbook this means the developer working with improving ArgoUML.
- "I" in the document refers to the author and is only used to denote the authors personal opinion. Avoid using it!
- Use the active voice.
- Use plain rather than elegant language.
- Use specific and concrete terms rather than vague generalities.
- Break up your writing in short sections. Each section dealing with one topic.
- Use the present tense.
- Opt for an informal rather than a formal style.

8.3. Document Conventions

- All titles of chapters, sections etc. are capitalized throughout.
- All titles of figures, tables etc. have the first word only capitalized.
- Spelling is US English. (According to The Webster's Second Unabridged.)
- Use full URLs throughout all documents! Rationale: These documents may also be published in other formats than html on the ArgoUML web site.
- Do not include lists of what changes have been done to the files. This information is kept by the version control tool. This is changed since Jeremy Bennet did the work for the 0.9/0.10 User Manual and there might still exist such lists. Remove them while changing the files!

  The Cookbook has a Change Log (See Change Log) that is updated for every significant change but that is for the purpose of making it easier for the readers.
- When problems in the current implementation of ArgoUML are mentioned or perhaps even emphas-

ized using the `warning` tag, include the issue number in a sgml-comment in the source so that it is easy to know if this problem has been fixed when revising the document. The issue should be mentioned in the format “issue xxx”, i.e. there should only be a space between the word “issue” and the issue number. This allows the tigris web site to generate links when viewing the manual source.

- Do not write "currently". Better write either "in version 0.14" if you mean in the stable version 0.14 of ArgoUML or "in version &argoversion;" if you mean in the current version of the document as defined in `default.properties` when the document is deployed. There are some old references to "current" or "currently" also. If you encounter them, try to remove them!

- For documents that contain an “index”, Add indexterms while doing changes. Creating the index is a good idea and we eventually should have indexterms all over. Initially, the manual was written without useing indexterms at all. They have been added generously on certain parts but that makes the index strangely biased.

  Capitalize the part of the indexterms that are terms.

  Don't use the tertiary level of the index terms but use only two alternatives: Only primary, and primary/secondary. If you are unsure when to use primary or primary/secondary use the small word approach. I.e. if the indexterm contains a small word (typically to, of, for, in) and normally not capitalized, let the secondary start with that small word.

  When using primary/secondary, see that you get the same kind of word as used before in the index (especially when it comes to differences in singular/plural-form). Also create other indexterm by turning the phrase to as many permutations that you can think of.

8.4. DocBook Conventions

- The top level document of the document is in `documentname.xml`. Each chapter (or preface, glossary, appendix etc) of the cookbook and the quickguide is a separate file, defined as a system entity and included from this top level file. The manual is one big file: `manual.xml`

- There may be some useful entities defined for common terms in the beginning of this top level document.

  E.g. for the cookbook: The use of `&argouml;` will ensure consistent naming of the product (ArgoUML) and allow us to change it later (to Argo/UML, Argouml or whatever).

  In the build script there is some magic that translates @tagname@ to a real value. E.g. @VERSION@ in the `documentname.xml` file into *`0.16`*.

- XML comments are used throughout to explain what various sections are trying to achieve.

- Cross-referencing requires use of `id.` attributes. Many of these used in the manual are of the following format, but the use of this format is not obligatory any more.

  To avoid confusion, use a prefix of `ch.` for `chapter`, `app.` for `appendix`, `s.` for `sect1` through `sect5`, `fig.` for `figure`, `tab.` for `table` and `gl` for `glossentry`.

  A second prefix of `tut.` or `ref.` is allowed to distinguish tutorial and reference material. The remainder of the tag should be descriptive, but concise with words separate by underscore. Where a graphic is involved this remainder should correspond to the file name. For example `fig.ref.navigation_pane` for a figure showing the explorer, with the diagram in `navigation_pane.gif`

  There is one exception to this and that is the description of the critics in the manual. Each paragraph

about a critic is instead marked with `critics.` followed by the classname implementing that critic. The reason for this is that the intention is to have the manual accessable when pressing the Help button on that critic. Generating a link to the correct place in the manual is easier if the classname need not undergo some kind of textual transformation and the implementation doesn't care if a a specific critic is described in a `sect1`, `sect2`, `sect3`, or `sect4`. Reorganizing the manual would otherwise affect also the java code. The conversion to the correct tagname or really the correct URL is currently implemented in the `defaultMoreInfoURL()` method in the `org.argouml.cognitive.critics.Critic` class.

- Only use `glossterm` (for the term *or* its abbreviation/acronym), `glossdef` and `glossseealso` within `glossentry`. Other entries are not implemented in the style sheets and so do not appear in the glossary!

- Use spaces rather than tabs. Tabs are generally set so large the text moves over to the right of the page, and are not set the same everywhere (emacs uses 8 spaces, some MS editors use 6 spaces), making documents unreadable between users.

- The indentation size is 2.

- Make a new line after each sentence or before expressions. The Docbook source is source that is handled by subversion. When structuring the text the parts are paragraphs, sentences and words. By having each sentence on a line of its own it is easier to see which sentences have been changed and which have not in the `diff` reports from subversion. The contributions that Jeremy Bennet did for the 0.10 User Manual are not written like this. Change it while changing the paragraphs.

- All block graphics should be encapsulated within `figure`, allowing reference from around the text. Set attribute `float` to 1 to allow the figure to float (makes life easier for printed version).

- All block graphics should be provided through `mediaobject` and provided with both an `imageobject` and comprehensive description in a `textobject`. This gives the potential of meaningful content where a diagram cannot be displayed for any reason. Where appropriate the `mediaobject` should be wrapped by `screenshot`.

- Inline graphics can be done through `inlinegraphic`, rather `inlinemediaobject`. A textual alternative is of little value in these circumstances. Where appropriate the `mediaobject` should be wrapped by `guiicon`

8.5. For Eclipse Users

If you use the Eclipse, then the XML editor that comes with the Eclipse *Tools Platform (WTP)* project allows comfortable editing.

Regretfully, the docbook-validation of this editor (as any other suitable editor that has been evaluated) currently does not support the way the cookbook and quickguide are divided in seperate XML files. The manual consists of one single XML file, so it does not have this problem.

8.6. For Emacs Users

If you use the psgml library within emacs, then editing and verifying XML gets easier. Information on using this facility is inluded with psgml.

- Emacs' local variables appear in a few lines of comment at the bottom of each XML file. Please don't delete these!

- Adding `(setq sgml-set-face t)` to your `.emacs` file will cause all tags and entities to appear in boldface.
- Adding `(setq sgml-auto-activate-dtd t)` to your `.emacs` file will ensure the DocBook DTD is parsed as soon as the file is loaded.

8.7. User Manual Plans

The User Manual is a very separate part of the ArgoUML project. It is independent of the rest of the project w.r.t. updates, deliveries, ambition and plans. The development of the User Manual is more or less a project of its own. Since autumn 2003 we also have an appointed sub project leader for this. This Responsibility is called Editor for the User Manual and Quick Guide and is held by Michiel van der Wulp.

This section describes the ambition and plans for the User Manual.

8.7.1. Target Audiences for the User Manual

Target audiences are the following:

- Experienced users of UML in OOA&D (perhaps with other tools) who wish to transfer to ArgoUML.
- Designers who know OOA&D, and wish to adopt a UML based process.

In the longer term it would be desirable to also target the following.

- Those who are learning design and wish to start with a UML based OOA&D process.
- People interested in modularized code design with a GUI.

8.7.2. Goals for the User Manual

The goals are (in priority order):

1. A tutorial style explanation of ArgoUML in the context of an OOA&D process.
2. *Descriptive* reference material on all components of ArgoUML
3. Keep boundaries clearly defined, to avoid duplication with the Cookbook, FAQ, Quick Guide, online help etc.

I (probably Jeremy Bennet in 2002?) think the existing User Manual is a good start particularly towards the second of these goals.

8.7.2.1. What the User Manual is not (currently)

To keep the effort feasible the user manual should avoid the following (at least initially).

- Providing a quick overview—the Quick Guide already does this.
- Listing all the errors and what they mean. The help system does this—one day the manual will link to that.
- Explaining the internal workings of ArgoUML. The cookbook, combined with Jason Robbins dissertation is already a good start for this.

8.7.3. Suggested Manual Structure

Here are my (Jeremy Bennet, 2002?) thoughts. I think the user manual is really a set of two books, the tutorial manual (corresponding to Part I of the current manual), and the reference manual (Part II of the current manual)

I (Jeremy Bennet, 2002?) suggest that the tutorial book be based around an OOA&D process (any preferences), and that each UML concept is introduced with each step of the process, followed by an explanation of how to do it under ArgoUML. A *simple*case study will be needed throughout.

8.7.3.1. Tutorial Manual Structure

1. Introduction
    a. Origins and overview of ArgoUML
    b. Scope of the User Manual. Include cross-reference to other documentation (Cookbook, FAQ, Quick Guide, on-line help, ArgoUML website etc).
    c. Overview of the User Manual. Explains that ArgoUML will be explained in the context of an OOA&D process, and with an example running through.
    d. Assumptions. At this stage assume the user knows OOA&D, but not UML.
2. UML Based OOA&D
    a. Background to UML — what it is, history etc.
    b. UML based processes for OOA&D
    c. ArgoUML Basics — projects, drawing, exploring, details
    d. What ArgoUML has that other tools are missing (critics, to-do list, based in cognitive psychology theory).
    e. The Case Study
3. Requirements Capture
    a. Use Case Diagrams (this section will be relatively large, because its the first time we use ArgoUML to create something).
4. Analysis
    a. Concept Class Diagrams
    b. System Sequence Charts and Collaboration Diagrams

   c. System State-chart Diagrams

5. Design

   a. Class Diagrams for Realization

   b. Sequence Charts and Collaboration Diagrams for Realization

   c. State-chart Diagrams for realization

   d. Package Diagrams

6. Build

   a. Deployment Diagrams

   b. Code Generation in ArgoUML

8.7.3.2. Reference Manual Structure

1. Material on each of the diagram types, each of the artifacts that can appear on the diagrams and details of the features of each artifact type.

2. An Index

8.7.4. Actions, Priorities and Questions

This section has two serious problems. Firstly, I (Linus Tolke, 2004) think Jeremy Bennet wrote this and then started and has completed a lot of the items so they could be checked off. Secondly, keeping this list in a docbook document is not a good idea. It is better to make issues in Issuezilla of it that can be individually closed. I (Linus Tolke 2004) will make issues of the things I think are left to be done and remove this section (unless someone beats me to it). I (Linus Tolke 2006) am still hoping that someone will beat me to it.

8.7.4.1. Actions and priorities

Here's my first call for what needs to be done in priority order. From the comments made over the last few days I think the first 5 items won't take very long, meaning effort can concentrate on the main stuff.

1. Get buy-in for the approach. (Completed)

2. Agree document structure (broadly). (Completed)

3. Choose a suitable example to run throughout.

4. Break into several files (XML entities) to make the manual more manageable. (Completed and then joined again.)

5. Identify all existing sources of material to be reused

6. Get writing! I (Jeremy Bennet 2002?) suggest the priorities here are:

   a. User Manual sections relating to ArgoUML diagrams and artifacts (assume the reader knows

UML already, and allows a quick advance by pulling together a lot of existing material).

b. User Manual examples

c. User Manual sections relating to additional ArgoUML cognitive design features.

d. User Manual sections relating to UML (for readers who don't know UML).

e. Completion of Reference Manual material.

7. Create an index. (Completed)

8.7.4.2. Remaining Questions

1. The current manual shows copyright held by Phillipe, and no legal notice. What is the position of this material? (Solved)

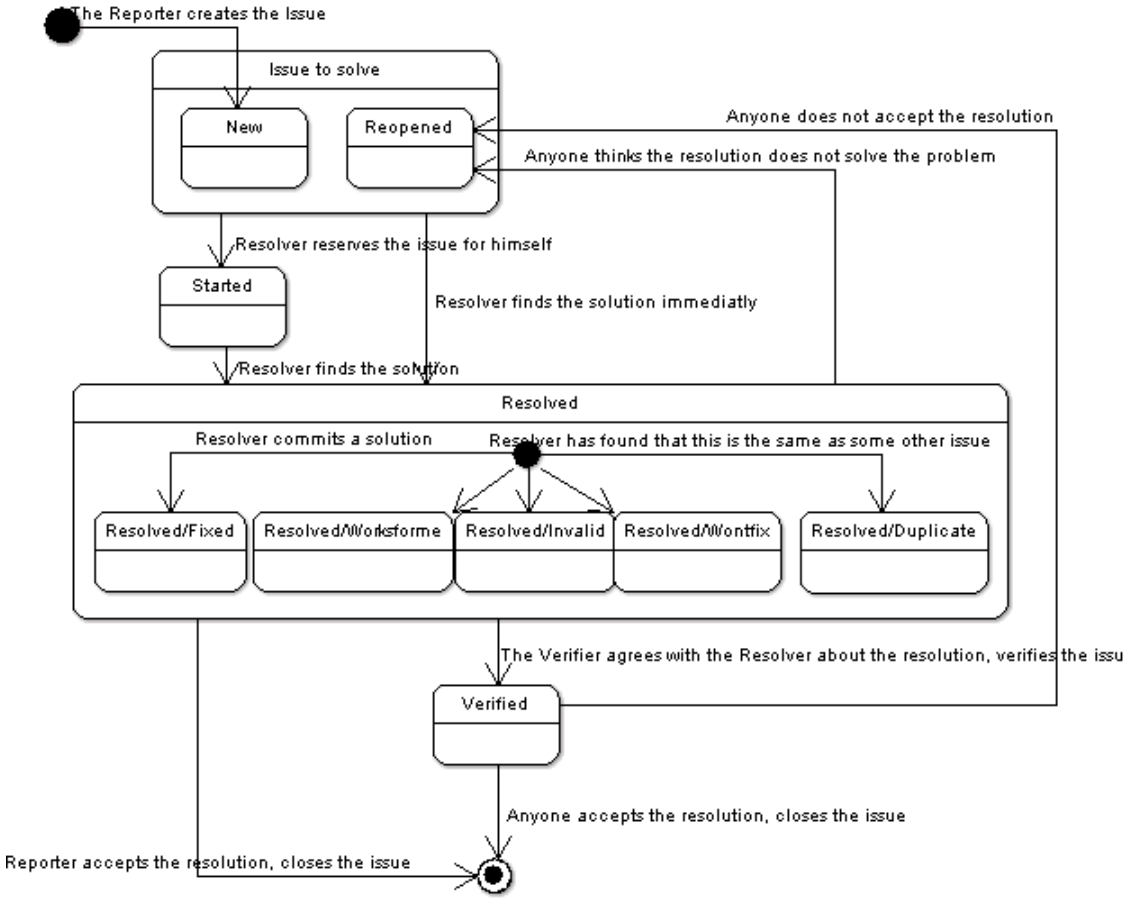
Chapter 9. Processes for the ArgoUML project

This chapter contains processes used when working with the ArgoUML project.

These processes are provided with the hope of being helpful for the members of the project and if they feel too complicated, ambitious or overworked, please raise the issue of simplifying them on the developers' mailing list [mailto:dev@argouml.tigris.org].

9.1. The big picture for Issues

Here is the big picture of the life of an Issue.



9.2. Attributes of an issue

This is what the different attributes mean and how they are used in the ArgoUML project. This is to be read as an addendum to the Tigris definition of the resolutions [http://argouml.tigris.org/nonav/scdocs/issue_lifecycle.html] and for that reason it is not a complete list.

9.2.1. Priorities

The priorities are used in the following manner in ArgoUML:

- P1 - Fatal error

  These issues are blockers for all releases.

  Examples: ArgoUML cannot start; Crashes program, JVM or computer; and Significant loss of user data.

- P2 - Serious error

  These issues are blockers for stable releases.

  Examples: Information lost.

- P3 - Not so serious error

  Examples: Functions not working; Strange behavior; and Exceptions logged.

- P4 - Confusing behavior

  Examples: Incorrect help texts and documentation; Inconsistent behavior; UI not updated; and Incorrect javadoc.

- P5 - Small problems

  Examples: Spelling errors. Ugly icons. Excessive logging. Missing javadoc.

9.2.2. Resolutions

- LATER

  Used to denote that a certain issue cannot be resolved until some special upcoming and planned-for event has happened. The event in question is noted in the target milestone.

  Events can be things like, dropping support for a JDK version, changing the version of UML that we support, or replacing some central mechanism within ArgoUML. Once they have a target milestone registered, they are considered events.

- REMIND

  Not used.

Rationale: Each issue have basically four states:

1. NEW/STARTED/REOPENED - To be resolved

2. RESOLVED - To be verified

3. VERIFIED - To be closed

4. CLOSED - Finished.

The statistics is based on this and persons looking for issues to resolve look among the "To be resolved"-group (the web pages to help in this are set up in this way). This is also in sync with our release process.

Looking at it from a single persons perspective an issue is either a "I could work with this issue but I currently don't", "I work with this one", or "I am now done with my work on this issue". For a resolver this is corresponds to NEW/REOPEN for the first group, STARTED for the second and RESOLVED for the third. For a verifier this corresponds to RESOLVED for the first group and VERIFIED for the third group.

The RESOLVED/REMIND does not fit this. They risk to be verified because the rest of our process urges people to resolve issues that are RESOLVED in which case they are probably lost. They risk to be hanging in the RESOLVED state because nobody understands where they should go from there. It is not clear who is responsible to move them forward. The person that "resolved" them or someone else. Someone risk to think that there is nothing left to do since it is resolved and if so his options of doing work are reduced which could lead to that he actually does less with ArgoUML than he else would.

To amend this we have made two things:

1. Decided that we don't use the RESOLVED/REMIND states.

2. At every release, as part of the release process, clean up issues that for some mysterious reason ended up in these states (See Section 2.11, “Making a release”, 6, f)

If you plan to solve an issue now, assign it to you, start it, and set the target milestone to the release you plan to have it solved. This will signal to everyone that you have the responsibility, will pursue it, and your time plan.

If you don't plan to solve this now, leave it in the "up for grabs"-pile (as not resolved). Somebody else might want to work with it.

If you know that an issue cannot be resolved now because it requires that another issue is solved before, register the other issue as "depends on" and leave the issue in the "up for grabs"-pile (as not resolved).

If you know that an issue cannot be resolved now because it requires some big event to take place, put the milestone for that event in the target milestone and resolve the issue as RESOLVED/LATER.

- WORKSFORME

  This means that it works in a released version of ArgoUML. State the version in the comment.

  If the version stated by the reporter in the issue is not the same as the version in the comment then this probably means that problem was fixed in some release without anyone noticing that this problem was fixed.

9.3. Roles Of The Workers

The roles described below are per issue, i.e. for every issue, there is at least a reporter and a resolver. Hence, each person involved in issues for the ArgoUML project can - at the same time - have different roles, and consequently, has issues to report, issues to close, issues to resolve, and issues to verify.

9.3.1. The Reporter

The Reporter is the person who enters the issue in Issuezilla.

Skills: The reporter is an ArgoUML user, should not need any knowledge of what the ArgoUML project is actually doing.

Responsibilities:

- Report an issue

  The address to enter new issues is: http://argouml.tigris.org/issues/enter_bug.cgi [http://argouml.tigris.org/issues/enter_bug.cgi]. To enter new issues, you will need to sign up for a Tigris account. For some operations in the issue database you may also need to apply for Observer status to the ArgoUML project.

- Answer clarification requests

  Occasionally, the developers of ArgoUML need to request the Reporter more information, to be able to solve the issue correctly. Another way of putting it is to say that if the issue was reported without some vital information the Reporter has some more work to do.

- Close the issue

  This applies to an issue that is in the resolved or verified state. At the end of processing the issue, the reporter has the final word: he can check the result, and if he agrees with the solution, close the issue himself. Closing an issue requires at least "observer" role in the ArgoUML project.

- Reopen the issue

  This applies to an issue that is in the resolved, verified, or closed state. The reporter has the final word: he can check the result, and when he does not agree that the solution is correct, he can reopen the issue himself. Reopening an issue requires at least "observer" role in the ArgoUML project.

9.3.2. The Resolver

The Resolver is the software developer who attempts to resolve the issue. Doing so requires at least "observer" role. The "developer" role is only needed to commit things into the repository (e.g. submit changed Java code, scripts or documentation).

Remark: Someone who does not have the developer role, but solves the issue and convinces someone else to commit the solution, is still the Resolver even though he cannot commit things into the repository.

The goal of the Resolver is to progress the issue to the status of "Resolved". The resolver may be the same person as the reporter.

Responsibilities:

- Decide usefulness (if this issue is really a bug or enhancement and if it is worth solving)

  The Resolver has to decide if solving the issue is really a useful improvement for ArgoUML. The Reporter of the issue may very well be mistaken in entering a bug-issue for what is in fact a feature, or entering an enhancement-issue which is not really an enhancement. Another thing that could be is a bug that appears in very exceptional circumstances and that may have large impact on ArgoUML architecture. If the Resolver decides after the investigation that this bug is really not that important or that he is not the right person to solve it he enters his findings as a comment and assigns the issue back to anyone (issues@argouml) and moves along to work on another issue instead.

- If applicable, program and test a solution

  As this might take considerable time it might be a good idea of the Resolver to assign the issue to himself to reserve the issue. He can also signal progress by setting the issue to the state Started.

- If applicable, write test cases

- Set the issue in the end on "RESOLVED".

  When the resolver is finished with the issue, he puts it in "RESOLVED" status, and indicates the "resolution" is FIXED, WORKSFORME, INVALID, WONTFIX, or DUPLICATE.

Skills: The resolver needs to know a lot of the insides of the ArgoUML code, Java, coding standards, and also the current status of the project with goals, requirements and release plans.

9.3.3. The Verifier

The Verifier may be neither the Reporter, nor the Resolver of the issue. The task of the Verifier is to check the quality of the solution by confirming that the solution is complete, to the point, bug-free, etc. This is an important part of the quality assurance work we do in the ArgoUML project and the object is to make sure that a resolved issue is in fact resolved.

The test must be done on the "Target Milestone" version of the issue, or any later version released to the public.

Responsibilities:

- Check that the issue is solved in the stated version of ArgoUML

- Verify the issue.

  If the Verifier can conclude that the problem does not exist or the feature/enhancement is now present.

- Close the issue.

  If someone else has already verified the issue then the issue can be closed.

- Reopen the issue if the solution is not fully correct

  If the solution is not correct or the feature/enhancement does not work, it is the duty of the Verifier to reopen the issue.

Skills: The verifier needs only to focus on that issue, how the problem in it is formulated. He doesn't need to know how it is actually solved.

9.4. How to resolve an Issue

This can be performed by any member of the project (any role). Persons without the Developer role need a person with the Developer role to actually commit the work if the resolution involves changing some artifact. There might be special skills involved but it differs widely depending on the nature of the Issue.

Do the following:

1. Pick any Issue that is NEW or REOPENED that you from the description think that you are able to solve. Best result if you also find some Issue that you really feel needs to be solved. The list of all of them [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=NEW&issue_status=REOPENED].

2. Look at your personal schedule and how much time you have during the next couple of weeks and compare that to the amount of time you think you will need to spend for solving the issue. Compare this to the release plan to see what release your contribution will fit in.

3. Accept the Issue and reserve it by assigning it to yourself. Set the Target Milestone to the release you have chosen.

4. Make sure you have a checked out copy of ArgoUML or else check out a new one.

   How this is done is described in Chapter 2, *Building from source*.

5. Mark the issue as Started (this could be done while assigning also).

6. Change the code to solve the problem.

7. Compile and test your new code.

   This should include developing a JUnit test case to verify that the problem is solved. You could also develop the JUnit test case before actually solving the problem.

   If your solution did not work as intended, continue changing it until it does.

   If you feel that your estimation of the complexity of the problem and your own abilities and time available was incorrect, then change the Target Milestone of the Issue to another one that fits your new estimation. This is just a change of plan.

   If you, at this point, feel that your personal plans have changed so that you won't have time to pursue the work, change the Issue back to "NEW" with your experiences so far stated in the comment. This means that you are giving up and giving the Issue back to anyone. You should also assign it back to issues@argouml or if you know someone else in the ArgoUML team that will continue the work, assign it to him. Remember not to commit your changes in the main branch but please commit your changes (if any) into a work branch and state the name of the branch in the issue. That will make it possible for someone to make use of your work so far.

8. Commit your changes and the JUnit test cases stating the number of the Issue in the comment.

   If you don't have a developer role in the project, this involves sending your changes to someone who has and then convincing him to commit them for you.

9. "Resolve" the Issue with the resolution "FIXED".

   Also set the target milestone of the upcoming release that will include the fix.

10. Sit back and feel the personal satisfaction of having completed a something that will be part of the

ArgoUML product.

11. If you during this, have discovered other problems, create new Issues stating those new problems according to the rule for creating Issues.

9.5. How to verify an Issue that is FIXED

This can be performed by any member of the project (any role). There might be special skills involved but it differs widely depending on the nature of the Issue.

If you are the Reporter of the issue, you Close the issue instead.

Do the following:

1. Pick any Issue that is RESOLVED/FIXED or WORKSFORME and that you have not raised, nor solved and that is included in a release (Target milestone set to a release available on the site). The list of all RESOLVED/FIXED and RESOLVED/WORKSFORME issues [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=RESOLVED&resolution=FIXED&resolution=WORKSFORME].

2. Run the specified release of ArgoUML. You can also use any later release. Use ArgoUML provided for downloads or through Java Web Start.

3. Test the problem in the issue and verify that the problem is no longer there or the feature is provided.

4. Do one of the following:

   - If the problem is gone, the feature is present put the Issue in Status VERIFIED and add the version of the ArgoUML used for the test in in the comment.

     Remark: As an additional activity, the verifier may check if the manual needs to be adapted, and if so, may REOPEN the issue with an explanation text, and setting the correct subcomponent (Documentation & Help).

   - If the problem is still there, the feature does not work, put the Issue in Status REOPENED with a description of what is still there, is still missing. Also state what version of ArgoUML used for the test in the comment.

5. If you during this, have discovered other problems than the one stated in the Issue, create new Issues for those new problems according to the rule for creating Issues.

6. Do this as many times as you like until there are no Issues left.

9.6. How to verify an Issue that is rejected

This can be performed by any member of the project (any role). There might be special skills involved but it differs widely depending on the nature of the Issue.

If you are the Reporter of the issue, you Close the issue instead.

Do the following:

1. Pick any issue that is RESOLVED/(INVALID, WONTFIX, or DUPLICATE) that you have not raised nor solved. The chosen issue need not be connected to an available release. The list of all RESOLVED/INVALID, RESOLVED/WONTFIX and RESOLVED/DUPLICATED issues [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=RESOLVED&resolution=INVALID&resolution=WONTFIX&resolution=DUPLICATE].

2. Read through the description provided.

3. Do one of the following:

   - If you agree with the statement and feel that the rejection is done for correct reasons, put the Issue in Status VERIFIED.

   - If you don't agree, put the Issue in status REOPENED and give a description as to why you don't agree.

4. Do this as many times as you like until there are no Issues left.

9.7. How to Close an Issue

This is performed by the person that originally raised the Issue, by the QA responsible for that area, or by anyone for issues that are verified. You need to be a member of the project (any role). This can also be done by someone who would raise the issue but did not because it was already present in Issuezilla.

1. Pick any Issue that is Verified (all VERIFIED issues [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=VERIFIED]) or that you have raised or refrained from raising because somebody else already had written it.

2. See that you are satisfied with the solution. This could involve reading through the resolution and starting the tool to verify it.

3. Do one of the following:

   - If you are satisfied, Close the issue.

   - If you are not satisfied but the problem is solved as it is written in the Issue, Close the issue and open a new Issue with the rest of the problem.

   - If you are not satisfied and the problem is not solved, put the Issue in status REOPENED with a description on what you are not satisfied with.

9.8. How to relate issues to problems in dependencies

ArgoUML uses products internally and is very dependent on that these products are functioning well. This are products like GEF, MDR, OCL, log4j, ...

Occasionally a problem found in ArgoUML is found to be a problem in one of the dependencies and cannot be or is extremely complicated to fix within ArgoUML.

If this happens this is the way to handle this problem.

This can be performed by any member of the project (any role). There might be special skills involved

depending on the nature of the problem. In this description "issue" means a issue in Issuezilla, "bug report" means a bug report in some other project, and "problem" denotes the conceptual problem.

Do the following:

1. During your examination of an issue you find that the problem is in one of the ArgoUML dependencies (GEF, MDR, OCL, ...).
2. Make sure that the issue is assigned to you.
3. Write a comment in the issue stating which one of the dependency that has the problem (and what the problem is within that dependency).
4. Post a bug report in that dependency bug reporting tool (or find that a bug report already registered).

   I am assuming that there is such a tool for the dependency in question. If there isn't, then make the bug report to the person responsible for this product so that we are sure that the problem is communicated.
5. Accept the issue (set it to STARTED) and enter the reference from the dependency bug reporting tool and if possible the URL to the bug reporting tool or to the bug report in question.

   I am assuming that there is a bug reporting tool for the dependency. If there isn't for the product in question, then include all communications (both ways) in the issue.

You are now responsible to follow up on the upcoming releases of the dependency. If you don't think that you are the best person for this (you should be since it was you that found that this problem is in the dependency), assign the issue to "the right person". To follow up you should do the following.

1. Look at each new release of that dependency to see if the bug report is in fact stated as fixed in that release.
2. If the bug report is fixed, then you weight together the importance of the problem, other bug reports that are also problems in ArgoUML that are solved in that release, the amount of work needed to fit the new version of the dependency instead of the old one, the planned releases of the dependency with promises to solve other bug reports, and the current release plan of ArgoUML. From this you decide whether it is time to do the update of the dependency within ArgoUML or to wait.
3. If you decide that it is time to update, you assign all issues against that dependency to you (if not already), then you do the work. The work is to add the new version of the dependency to ArgoUML, do all the needed work within ArgoUML to fit the new version, test and commit everything, put the issues indeed fixed in RESOLVED/FIXED, and close the bugs registered in the dependency bug reporting tool.

For dependencies that are not delivered with ArgoUML (JRE, Xerces, OS, drivers, HW, ...), the same process is taken except that the issue solved when it is entered in the ArgoUML FAQ or documentation or in some cases as tests in the code testing that we are not using that version. At that point is resolved (as RESOLVED/FIXED).

The rationale for this is that we, the development team, help the user to the right version of these by the FAQ and documentation and by code testing the versions.

9.9. How to Create Stable Release

We have two kinds of releases of ArgoUML:

- Development releases.
- Stable releases.

Stable releases are supposed to be better quality-wise and are always advertised to the users community on the main ArgoUML home page and as a news announcement.

Development releases are not supposed to be used by users and are only advertised to users for the purpose of recruiting developers or soliciting help with implementation and test of new features.

To increase the quality of a stable release, it is preceded by period during which a sequence of releases with increased quality standards.

The whole release schedule leading up to a stable release and patched stable release looks like this:

- Development Period.

  A period of one to several months where no special restrictions apply.

  During this period we attempt to make one development release per month. The releases are named x.y.z where y is an odd number and z is counting upwards from 1.

  The releases are checkpoints where:

  - Everything compiles (including the sub-projects).
  - The release script works.
  - No JUnit tests are failing.
  - There are no P1 issues.

  The releases are used:

  - As reference points when reporting bugs.
  - As reference points when verifying issues.
  - As reference points and convenient downloads for persons working with modules.

- First Alpha.

  This is the enhancement freeze point. All enhancements that are not completed and committed in the main trunk before this point will not be included in the stable release.

  The First Alpha release is named x.y.alpha1 or x.y.ALPHA_1 depending on the context. It marks the end of the Development Period and the start of the Alpha Period.

  Otherwise it works just like a development release.

- Alpha Period.

  A period of a couple of weeks where special restrictions apply when committing into the main trunk:

  - Only bug fixes are allowed in the code.

Put the number of the DEFECT you are addressing by the commit in the message.

Test case code can be added. Documentation, web site and other things can be added.

Exceptions to this are requested to and approved by the Release Responsible before commit.

During this period we attempt to make at least one Alpha Release each week. The releases are named *x*.*y*.alpha*z* where *y* is an even number and *z* is counting upwards from 1 that is the first alpha.

The purpose of the releases and their use are the same as during the development period.

- First Beta.

  This is the bug fixes freeze point. All enhancements and bug fixes that are not completed and committed in the main trunk before this point will not be included in the stable release. A known problems list could be compiled at this point.

  The first beta release is named *x*.*y*.beta1 or *x*.*y*.BETA_1 depending on the context. It marks the end of the Alpha Period and the start of the Beta Period.

  It is the first release candidate for the stable release. Because of this it is required that:

  - Everything compiles (including the sub-projects).
  - The release script works.
  - No JUnit tests are failing.
  - There are no P1 or P2 issues.

  Otherwise it works just like a development release.

- Beta Period.

  A period of a couple of weeks where the focus is quality assurance. Every developer should strive to:

  - Test ArgoUML as thoroughly as possible. Especially the areas that are new or changed since the last release.
  - Verify issues that are resolved.
  - Scrutinize the commits in the main trunk to see that no new bugs are introduced.

  Extreme caution applies when committing into the main trunk. Only under the following conditions are commits allowed:

  - It is a fix to some DEFECT that was previously fixed but it was found during the verification that the solution was not correct or complete.

    Reopen the DEFECT when the problem is found with a statement of what is still the problem. Put the number of the DEFECT you are addressing by the commit in the message together with the statement of the part of the problem. Resolve the DEFECT as FIXED and update the target milestone with the release name of the next beta.

    Test case code can still be added. Final documentation and web site updates for the release are done.

- All JUnit test cases are run from a cleaned checked out copy at the commit and no problems are found.

Exceptions to this are requested and approved by the Release Responsible before commit.

If a new problem is found the following needs to be done before committing the solution:

- The problem is registered as a DEFECT.
- The solution is implemented.

  Here the requirement is a high on the quality and low impact of the solution.
- A request is made to the Release Responsible to allow this change.
- This is granted by the Release Responsible.

During the period we attempt to release at least one beta release each week. The releases are named *x*.*y*.beta*z* where *y* is an even number and *z* is counting upwards from 1 that is the first beta.

Each release is a release candidate. When we have reached the point where no more issues are verified and we are confident that there are no more problems in this release we make the stable release without code changes compared to the last beta.

- Stable Release.

  This marks the end of the Alpha and Beta Period and the start of the next Development Period.

  The release is named *x*.*y* where *y* is an even number.

  The release is used:

  - By all users.

  It can also be used as a development release.

- A Stable Patch Release.

  If we find a serious problem in the stable release we can decide to make a Stable Patch Release.

  The following needs to be done before committing the solution:

  - The problem is registered as a DEFECT stating that it is a problem in the Stable Release or a previous Stable Patch Release.
  - A working branch is created against the release tag of the Stable Release or Stable Patch Release and the solution is implemented in that branch.

    Here the requirement is a high on the quality and low impact of the solution.
  - We decide that it is a serious problem and that we are going to do a Stable Patch Release.
  - Several developers scrutinize the solution, testing and verifying in the branch of the issue.
  - The Release Responsible creates a branch. If this is not the first Stable Patch Release the branch is reused.
  - The Release Responsible merges the solution into the branch.

A Stable Patch Release could contain several issues resolved. In that case they are all merged.

- Several developers scrutinize the merge, testing and verifying in the release branch.

The release is named *x.y.z* where *y* is an even number, and *z* is counting upwards from 1.

The release is used:

- By all users.

After the release is completed the person working with the solution commits his solution also in the main trunk if still applicable there.

Glossary

Terminology in the ArgoUML project

This is the Terminology for the ArgoUML project and also for the ArgoUML product. The purpose is to establish a common use of some words to make it easier to understand what we are talking about.

The terminology here is used for:

- All code (class names, variable names, method names)
- All comments throughout the code
- All strings in the user interface (in the default language en_US)
- All documents (in the default language en_US)
- All postings to the mailing lists and all other discussions within the ArgoUML project.

Add	Add existing object to something.
Code Generation / Reverse Engineering	Code Generation and Reverse Engineering are the processes of converting between the ArgoUML UML model and program code in a certain programming language. ArgoUML supports different languages such as Java, C++ and PHP for Code Generation and Reverse Engineering, but both are not supported for all languages equally.
Delete	Usually used as "Delete from Model". The object will be destructed, and will not be present any more on any diagram, nor in the model. Additionally, all objects that can not exist without the deleted object are deleted from the model, too. E.g. deleting a class also deletes all its associations.
Dependency	A piece of software that is developed by someone outside of the ArgoUML project and that the ArgoUML project is not responsible for. The ArgoUML product, like almost any other software project, use dependencies so that we don't have to do everything ourselves. Examples: JDK, NSUML, GEF, log4j, ... Dependencies were previously called subproducts.
Export	Export everything or some part from an existing structure to some format. This implies some kind of conversion involved.
Import	Import some other format into an existing structure. This implies that there is some kind of conversion involved and that the already existing things are not removed. They are either changed or left unchanged by the import. An Export, New, Import cycle can lead to information loss because of the two conversions involved.
Library	A part of a *dependency* that is installed and handled as a unit in the

	ArgoUML building and installation set-up.
Module	A piece of software that is to be loaded into ArgoUML by the Module Loader. Modules traditionally use the Plug-in interfaces and are for that reason also known as Plug-ins.
New	Create a new project or object. How does this relate to the Java GUI standards? (Don't use Create or Add)
Notation	Notation is the textual representation of model fragments on a diagram. ArgoUML supports Notation in different languages such as UML and Java. Notation may be not only pure text: e.g. UML attributes/operations have graphical text properties (underlining, italics) with a semantic relevance. To be shown on a diagram, the text has to be created by a generator, and when the user edits the text on the diagram, a parser processes the text, and adapts the model accordingly.
Open	Open an existing project or saved file. How does this relate to the Java GUI standards? (Don't use Load, Read, or Import for this).
Remove	Usually used as "Remove from Diagram". The object that will be removed is not deleted from the model, i.e. it still exists, but is simply not present anymore on the current diagram. It might still be present on other diagrams, or not, but it is surely present in the explorer. Additionally, all objects that can not be drawn without the removed object, are removed from the diagram, too. E.g. removing a class also removes all its associations. Once removed, an object can be "added" to a diagram (again).
Save	Save the existing project to a file. The Save operation save everything (it is a no-loss operation). A Save, Open cycle does not loose any information. (Don't use Export, or Write for this).

Index

M

N

O

P

R

S

T

U

V

W

X

Appendix A. Further Reading

A.1. Jason Robbins Dissertation

Cognitive Support Features for Software Development Tools

The dissertation of Jason Robbins is a *MUST READ* for everyone concerned about ArgoUML. Be careful though, since it is based on an old version of ArgoUML, but many of the concepts remain intact.

A.1.1. Abstract

Software design is a cognitively challenging task. Most software design tools provide support for editing, viewing, storing, sharing, and transforming designs, but lack support for the essential and difficult cognitive tasks facing designers. These cognitive tasks include decision making, decision ordering, and task-specific design understanding. To date, software design tools have not included features that specifically address key cognitive needs of designers, in part, because there has been no practical method for developing and evaluating these features.

This dissertation contributes a practical description of several cognitive theories relevant to software design, a method for devising cognitive support features based on these theories, a basket of cognitive support features that are demonstrated in the context of a usable software design tool called ArgoUML, and a reusable infrastructure for building similar features into other design tools. ArgoUML is an object-oriented design tool that includes several novel features that address the identified cognitive needs of software designers. Each feature is explained with respect to the cognitive theories that inspired it and the set of features is evaluated with a combination of heuristic and empirical techniques.

A.1.2. Where to find it

LINK: Robbins Dissertation [http://argouml.tigris.org/docs/robbins_dissertation/]

A.2. Martin Skinners Dissertation

Enhancing an UML Modeling Tool with Context-Based Constraints for Components

A.2.1. Abstract

Noch vor der Erstellung eines detaillierten Entwurfs hilft ein Spezifikationsmodell eines komponentenbasierten Systems dabei, Probleme so früh im Entwicklungsprozess wie möglich zu entdecken. Die Sprache CCL ('Component Constraint Language') wurde bei CIS entwickelt und erlaubt den Entwickler 'Contextbased Constraints' dem Spezifikationsmodell hinzuzufügen. Dadurch entsteht ein Modell, das über die Beschreibung der statische Struktur des Systems hinausgeht. Zur Zeit existiert allerdings kein Werkzeug, dass das Komponentenspezifikationsmodell in den Entwicklungsprozess integriert. Ziel dieser Diplomarbeit war der Entwurf eines solchen Werkzeugs, um die Philosophie des Continuous Software Engineering (CSE) zu unterstützten.

Before starting a detailed design, a specification model of the component-based system assists the software developer in early problem detection as soon as possible in the development process. The Component Constraint Language (CCL) developed at CIS enables the developer to add context-based constraints (CoCons) to a component specification model. This produces a model which goes beyond the simple description of the system's static structure. At this time, there is no tool to integrate the component specification model into the development process. The goal of this master's thesis was to design such a tool, thereby supporting the Continuous Software Engineering (CSE) philosophy.

A.2.2. Where to find it

LINK: Martin Skinners dissertation [http://www.cocons.org/publications/CCL_plugin_for_ArgoUML.pdf]

Appendix B. Repository Contents

This appendix describes what parts of the repository is used for what purpose. This is a rather terse collection. Further details on specific parts can sometimes be found elsewhere in this document.

Note

This explanation only describes the single Tigris project layout. Lately (since beginning of 2006) we have been working with splitting the project over several Tigris projects. This is not yet described here.

This chapter is organized as the repository itself and everything is in alphabetical order.

This is the normal structure for any argouml jar file and project.

- `build.xml`

  The file controlling the build of that subsystem.

  When built, the result end up in a newly created build directory.

- `build`

  Directory where the built things end up when building with ant.

  This is not kept in the repository. It is created by the build. For historic reasons this actually exists in the CVS repository of some of the projects. This will be fixed after the move to SVN.

- `build-eclipse`

  Directory where the built things end up when building with Eclipse.

  This is not kept in the repository. It is created by the build.

- `lib`

  jar files used by the project.

  This directory contains the jar files of products that are shipped with the project i.e. needed to run the project. For ArgoUML this is things like log4j, gef, ...

  These are distributed with ArgoUML and have licenses that allow this. For clarity the README files and licenses and other distribution details of each used jar will also be stored in this directory. (Quick summary: BSD License, Apache License, LGPL are OK, GPL is not.) Don't forget to arrange for the modules version and license information to appear when starting ArgoUML and in the About box.

  Take care also to make the versions of these libraries explicit, so as to allow people building from sources to figure out exact dependencies. Easiest way is to rename the files to include version informations, the same way as shared libraries in Unix world: foo-x.y.z.jar, bar-x.y.z.jar, etc...

- `src`

  Source code.

  This directory contains exactly one java class tree that is build into exactly one jar file.

- `tests`

  Source code for JUnit tests of everything that is in the src directory. See Section 2.6, “The JUnit test cases”.

  This directory contains exactly one java class tree.

- `tools`

  Tools used during the build process.

  Tools also have the readme files, licenses and other distribution files stored in this directory in much the same way as the libraries in `lib`. However the requirement on the license is different. The tools are never distributed with ArgoUML but merely used in the development of ArgoUML so it is enough to have a license that does not allow distribution. (Quick summary: BSD License, Apache license, LGPL, GPL, Freeware are OK.)

Because of they way the Tigris site works there is also the following:

- `www`

  This is all the static contents of the web site.

For historic reasons the argouml project i.e. the main project looks a little bit different.

- `documentation`

  Directory where the source of the documentation is.

  - `cookbook`

    XML-source code for this cookbook.

  - `docbook-setup`

    XML Tools and configuration files used for the formatting of the documentation from the XML-source to HTML and PDF.

  - `images`

    Pictures for all documents are collected here.

  - `javahelp`

    Not used. Empty.

  - `manual`

    XML-source code for the User Manual.

  - `quick-guide`

    XML-source code for the Quick Guide.

  It is not yet decided how to handle the User Manual in other languages. It is probably best if they reside in the documentation directory in the subproject for that language. That would make the cor-

rect group of developers get access to the files without special features.

Documentation for features and functions provided by other projects are, for the time being, best kept in the one big User Manual.

- `modules`

  Old module structure. To be removed.

- `src`

  In conflict with the description above this directory contains one directory for each subsystem within ArgoUML. Each directory is a complete tree as described above.

- `src_new`

  All source code for ArgoUML including pictures of icons.

  This is replacing the `src` directory above with the exception that it also contains the build.xml file.

The ArgoUML subprojects are projects on Tigris that belong to the ArgoUML project. To simplify the administration, they are all set up in the same way, i.e. as describe above.

Appendix C. Organization of ArgoUML documentation

Linus Tolke

Abstract

This chapter describes what goes into which part of the documentation. These ideas are formulated by Linus Tolke.

There are seven significantly different bits of documentation in the ArgoUML project. By documentation I mean some information of the product that is developed alongside the product and that has a persistent value.

1. The code, variable names, class names
2. The javadoc
3. The cookbook
4. The web site in SVN
5. The manual and quick-guide
6. Help texts within the running ArgoUML
7. The FAQ

These different bits have all different purpose and audience and the purpose of this chapter is to try to define that.

Table C.1. Bits of documentation

Bit	Audience	Main purpose	Contains
Source Code	1. Other developers that will maintain and improve on the code. 2. The compiler.	Implement ArgoUML in a maintainable and understandable way.	See Chapter 7, *Standards for coding in ArgoUML* for details on how to write the code.
Javadoc	Developers writing code that communicates or in other ways interact with this class.	Make it easy to see what the functions of every class are and how to use them.	Description of the functions of all classes, all public and protected methods, variables, and constants.
Cookbook	Developers writing code, maintaining the documentation or the web site.	Make it easy to learn how ArgoUML works and how to extend it. Be a collection of knowledge around how everything is	Instructions on how to add new functions and behavior. Instructions on how to do the chores around maintenance (build a re-

Bit	Audience	Main purpose	Contains
		set up. Be a store of the agreed solution around fundamental design decisions i.e. design decisions that are so big that it is meaningless to store them in the javadoc. Be a collection of knowledge around how and why the project makes certain decisions.	lease, publish a release, build the documentation part of the release, test ArgoUML, test the documentation, ...). Agreed project rules like what level of quality is aimed for and description of processes that achieves that level.
Web site	Everyone, i.e. developers in the project, users of the product, people searching for UML tools for the purpose of trying, testing, evaluating, and using the tools.	Be an entry point for the other parts of the documentation. Be the main download area for the ArgoUML product. Be the central point of the ArgoUML user community. Be the central point of the ArgoUML development project.	References to all the other parts of the documentation. Current project information like the contents of the upcoming releases and the plan for the nearest future. Easy access illustration for users to be. Some illustrations that do not work well in the other parts of the documentation. This is done as a complement to the other parts. Examples, tours.
Manual and quick-guide	Users of ArgoUML. Persons that want to evaluate ArgoUML for the purpose of starting to use it. Persons that are training to use UML and ArgoUML.	Describe how ArgoUML is installed and used. Describe how UML is used with ArgoUML.	Complete installation instructions for all supported installation schemes. Complete description on how to use ArgoUML in your project. Complete reference on how to use ArgoUML.
Help text in ArgoUML	Users of ArgoUML.	Give a quick help with a specific feature or button. Give short explanations of all commands and actions.	A complete set of quick help and explanations.
FAQ	Users of ArgoUML. Members of the users mailing list.	Cope for shortcomings in ArgoUML, the help text, the Manual and quick-guide and the web site.	A list of issues that are not addressed in the other part of the documentation. It is written in questions-answers-format and the contents is governed by the issues discussed recently in the user community.

The Cookbook, the User Manual, and the Quick Guide, are all written in docbook and generated into HTML and PDF during deployment. See Chapter 8, *Writing Documentation in the ArgoUML Project* for details on how to write these.