Cookbook for Developers of ArgoUML

An introduction to ArgoUML Programming

by Markus Klink and Linus Tolke

Cookbook for Developers of ArgoUML: An introduction to ArgoUML Programming

by Markus Klink and Linus Tolke

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 0.14 of ArgoUML.

Table of Contents

List of Tables

List of Examples

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 send more questions and/or answers to the dev mailing list [mailto:dev@argouml.tigris.org]!

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 price for climbing higher, you just get more responsibility in the project and more work.

1. Use ArgoUML.
2. Subscribe to the dev list.

   Monitor the discussions and as soon as you see something discussed where you have an opinion, jump right in!
3. Apply for an Observer role.

   This shows that you are commited to the project and also allows you to enter and comment on issues etc.
4. 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 just unsubscibe again.)

   c. Subscribe to the cvs list.

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

   d. Read the process part of the Developers Cookbook at Chapter 11, *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 Issuezilla works.

e. Get the Observer role granted.

f. From this on you can report bugs yourself directly in Issuezilla.

   You can also verify issues according to the verification process (see Section 11.5, “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).

g. 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.

h. Check out the source from cvs and build.

5. 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.

   c. Convince someone to commit your changes.

   d. Repeat.

      This can go on until you find that your main problem is the to get someone to actually commit your changes, not because they are hard to convince but because they don't have time to do commits to keep up with your pace.

6. Apply for a Developer role.

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

   There are a lot of special requirements on you to get this granted.

   Noted here for Linus to keep track on what to verify.

   - Understanding and accepting the goals.
   - Understanding where we are in the development process.

- Understanding the terminology used in the project.
- Good knowledge of CVS.
- Understanding the set of tools (ant, junit) and how to use them.

7. 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 challanges to your taste.

8. Accept responsibility for a specific area.

   With this you are part of the core team developing ArgoUML.

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 analyse the ArgoUML project for whatever purpose that may be.

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

Information on how you can compile ArgoUML.

Information on how different features of ArgoUML are implemented.

Information on how you should add modules and Plug-ins to ArgoUML.

Information that you, as a developer of ArgoUML, need to know about how the project is organized and how to contribute.

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 modelling tool.

1.5. Mailing Lists

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

It is also recommended to join the CVS 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

If you are in a hurry:

```
C:\Work>set CVSROOT=:pserver:guest@cvs.tigris.org:/cvs
C:\Work>cvs login   (use guest as password)
C:\Work>cvs checkout argouml_src
C:\Work>set JAVA_HOME=C:\Programs\jdkwhatever
C:\Work>cd argouml\src_new
C:\Work\argouml\src_new>build run
```

A window from the newly compiled ArgoUML opens after a while!

That was the compact version for Windows + JDK. (Note: jdk cannot be installed in a directory that contains space in its name.)

If you don't understand this or it doesn't work read the rest of the chapter that describes all the nitty details about why and how.

2.1. Getting started

In order to develop with ArgoUML it is absolutely mandatory to get the CVS version of ArgoUML. How this is done is described in Download from the CVS repository.

Notice that the CVS contents is not only a set of source files but instead it is the complete development environment for all work within the ArgoUML project.

2.1.1. Which tools do I need to build ArgoUML?

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

- A computer with a free disk space for your work.

  100MB is enough to download everything from the repository. (Currently March 2003 it is 68MB). 150MB is enough to download all and build the tool and the documentation. (Currently March 2003 it is 114MB). 250MB is enough to build it all (javadocs, documentation, classes, packages, ...).

- CVS for getting the files and committing source code updates.

- JDK, at least version 1.3 (includes the java compiler)

2.1.2. 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 CVS.

- ANT, the tool to manage compiling and packaging.

- 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 CVS.

- saxon for building documentation from docbook format.
- Docbook XSL stylesheets.
- fop for generating pdf versions of the docbook format.

  To build a pdf file with the pictures included you need Jimi.

2.1.3. What libraries are needed and used by ArgoUML?

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

- nsuml, the Novosoft UML library.

  ArgoUML project doesn't include the developing of Java classes for the purpose of storing, saving and loading an UML Model. That work is done by NSUML and is used by ArgoUML.
- GEF graph editing framework, available from gef.tigris.org [http://gef.tigris.org].

  It is also recommended that you check out GEF at the same time as you check out ArgoUML because many things in Argo relate to GEF and it is quite handy to have the source code available. GEF is also residing at tigris so you can do a simple **cvs *-d :user@cvs.tigris.org:/cvs* co gef** (with the same checkout arguments you had when you checked out ArgoUML) to get it.
- 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.

2.2. Download from the CVS repository

The CVS repository at Tigris is accessable using the pserver protocol. The CVS root is `/cvs` at cvs.tigris.org. You use your Tigris login and Tigris password.

This means that you will set the `CVSROOT`-variable to *`:pserver:login@cvs.tigris.org:/cvs`* where *`login`* is your Tigris login. This needs to be done for the first checkout. After that the root will be remembered by the checked out copy.

The next thing to do is to login. It is done using the command: **cvs login**. This only needs to be done once and then the account on your machine remembers this.

Then you do the actual checking out. **cvs checkout *`modulename`***.

The CVS module you need to check out to build ArgoUML is *`argouml_src`*. This will check out the directories `argouml/lib`, `argouml/tools`, `argouml/src`, `argouml/src_new`, and `argouml/tests`.

If you want to build the documentation you check out the module *`argouml_doc`*. This will check out the directories `argouml/lib` `argouml/tools` and `argouml/documentation`.

If you want to work with the web site you check out the directory *`argouml/www`*.

If you give the argument *`argouml`* all of ArgoUML is checked out. That is no problem except for the extra use of bandwidth and disk space but if you have plenty of both, get it all, and eventually you will see how everything is used for a purpose in the project.

If you don't want to acquire a tigris login to do this you can use the "guest" account with the password "guest". Since the checked out copy remembers the login you used to do the check out, if you do this, you will have to remember to delete this copy and start over if you get a developer role in the project and want to do commits directly.

2.3. Build Process

The ArgoUML build process is driven by ANT, and it is highly recommend that you stick to that. There are people known to build from JBuilder or Netbeans, but always make sure that your work compile with the plain vanilla build process.

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 (`src_new`, `documentation`, and `modules/`*`whatever`*).

2.3.1. How ANT is run from the ArgoUML development environment

For your convenience the ant tool of the correct version is present in the CVS repository of ArgoUML in the file `argouml/tools/ant-1.4.1/lib/ant.jar`.

Normally ant is started with the command **../tools/ant-1.4.1/bin/ant *`arg`*** and in the modules **../../tools/ant-1.4.1/bin/ant *`arg`*** . On windows the command **..\tools\ant-1.4.1\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`***.

2.3.1.1. 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 using **./build.sh**

   This gives you a list of targets with descriptions

4. Compile and run Argouml using **./build.sh run**

   You can do this over and over again when you have modified something or want to compile and run again.

2.3.1.2. Compiling for Windows

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

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

   chdir *\your\checked\out\copy\of\argouml\src_new*

3. Start ant using **build**

   This gives you a list of targets with descriptions

4. Compile and run Argouml using **build run**

   You can do this over and over again when you have modified something or want to compile and run again.

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

2.3.1.3. 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.

2.3.1.4. Building javadoc

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

2.3.1.5. 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.

2.3.2. How documentation is presented

This describes how the documentation arrives on the web site.

2.3.2.1. How the ArgoUML web site works

Tigris provides the ArgoUML site to be edited through CVS. Everything that is checked in under `argouml/www` becomes immediatly available at the url http://argouml.tigris.org/ with some added decorations.

Example: The file `argouml/www/project.html` is available at http://argouml.tigris.org/project.html.

This is the way the site is maintained and updated.

2.3.2.2. The ArgoUML documentation

For the ArgoUML project the same documenation shall be available in both html, pdf and javahelp. To this end the documentation is written in docbook xml and generated into two versions of html (one page per chapter and one

page for the whole book), pdf and javahelp.

We have tools that does the conversion from docbook xml to html and pdf. The conversion is done whenever you need to look at the result or when you want to present the final result on the web site.

There are currently three different books generated in this way, each into its own directory. They are cookbook (this document), manual and quick-guide. They are all generated and stored in the exact same way except for the name of the directory that is one of `cookbook`, `manual` or `quick-guide`. Below I will reference these directories using *`book`*.

When a new version of the documentation is to be made available on the web site the responsible document release person does the following:

1. He checks out everything needed and a copy of the `argouml/www`.

   The module *`argouml_docs`* is there for this purpose.

   If wanted, the CVS repository could be tagged and then the tag can be checked out. This makes it possible to know exactly how a certain version of the documentation was generated.

2. The documentation is generated using **build docs**.

   This generates all three books and the result appears in `argouml/build/documentation/defaulthtml/`*`book`*, `argouml/build/documentation/printablehtml/`*`book`*, and `argouml/build/documentation/pdf/`*`book`*.

   This has been done several times before while preparing the release so no problems are expected. If there are problems then the preparations were not good enough and the process is best stopped right here.

3. All the old files are removed from the checked out copy of `argouml/www/documentation/defaulthtml/`*`book`*, `argouml/www/documentation/printablehtml/`*`book`*.

4. New files are copied into the checked out copy of www on top of the previous files there replacing them.

   All the files are copied from `argouml/build/documentation/defaulthtml/`*`book`* to `argouml/www/documentation/defaulthtml/`*`book`*. The same for `printablehtml` and `pdf`.

5. No longer used files in `argouml/www/documentation` are removed from CVS and new files are added.

   cvs -n update

   Watch for "Missing" and "Unknown" files.

   The missing files are scheduled to be removed by: **cvs remove *`each of the missing files`***

   The "Unknown" files are scheduled to be added by: **cvs add *`each of the added files`***

   This removing of missing files and adding of unknown files may seem backward but it is from the perspective of CVS. The missing files are the ones that were present in the previous version of the documentation and do not have a replacement, either because that chapter does not exist anymore or that the tool generates filenames differently. The Unknown files are files with filenames that for the same reason appear from one version of the documentation to the next.

6. Commit the changes thus publishing it on the web site.

   cvs commit -m'New version of the documentation published'

7. The pdf book is uploaded to the download page.

2.3.2.3. How developers work with documentation

Developers that work with the documentation or with the tools to generate the documentation (or anyone else interested in this) can generate the documentation like described above and examine the result in `argouml/build`. It is only the last part about checking in and upploading the result under `argouml/www/documentation` that requires write access in the CVS and synchronisation with the rest of the project.

In order to do this you need to check out the whole of the `argouml/documentation` directory. You also need the directory `argouml/lib` and `argouml/tools` that contain the tools used: ANT, Fop, saxon, ...

The subdirectories of `argouml/documentation`, `cookbook`, `manual`, and `quick-guide` each contain one of the four 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.

2.3.3. Troubleshooting the development build

2.3.3.1. Compiling failed. Any suggestions?

It might be that some other developer has made a misstake 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 **cvs update -d** before trying to build again. If that doesn't work remove your checked out copy completely and get it all again through CVS.

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]!

2.3.3.2. Can't commit my changes?

You need to have a developer role in the ArgoUML project. If you don't then you cannot do commit yourself. Discuss what you have done and how best to test it on the developers mailing list [mailto:dev@argouml.tigris.org]! Eventually someone will 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, changed the CVSROOT variable you still cannot commit changes done in the repository since the checked out copy contains information on who checked out. For this reason, it is best to apply for an Observer role in the project if you are going to work with the source at all. The Observer role is probably granted within a couple of days (we welcome everybody!) and then you can check out with your tigris id. This means that when you eventually are granted a Developer role you can continue working with the same checked out copy.

2.4. 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 testcases java source code is located under `argouml/tests/org/argouml`.

2.4.1. How to write a test case

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

The name of the test case 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` searches for test cases with these names. If you write a test case that does not comply to this rule you still can run the test case manually after having started 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.

Testcases that doesn'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 case with **build tests** using jdk1.4 or later.

Testcases that do require GUI components in place have filenames like `GUITest*.java`.

We should try to get as many tests from the 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 called `org.argouml.`*`x.y`*`.Test`*`z`* stored in the file `tests/org/argouml/`*`x/y`*`/Test`*`z`*`.java` containing all the Unit Test Cases for that class that don't need the GUI components to run. Classes that have things that needs to be tested that do need GUI components to run should also have a class named `org.argouml.`*`x.y`*`.GUITest`*`z`* stored in the file `tests/org/argouml/`*`x/y`*`/GUITest`*`z`*`.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`*`.Test`*`z`* or `org.argouml.`*`x.y`*`.GUITest`*`z`*. 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.

The test case imports the JUnit framework:

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

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

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

```
// Exampel 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("hejsan"));
        assert("Disabled", to.shouldBeEnabled());
    }
    public void testCreate2() {
        UMLAction to = new UMLAction(new String("hejsan"), true);
        assert("Disabled", to.shouldBeEnabled());
    }
    public void testCreate3() {
        UMLAction to = new UMLAction(new String("hejsan"), true, UMLAction.NO_ICON);
        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();
    }
}
```

2.5. Manual Test Cases

The manual test cases are here to help us test ArgoUML in order to cover things that are not testable with the JUnit test cases. Since it is a little bit more cumbersome to run them, a tester must read the test cases, understand what he is supposed to do, do it, and document the result, we try to go as far as possible with the JUnit test cases and have as few manual test cases as possible. I.e. If one of these tests can be converted into a JUnit test case we shall try to do so because it can save us a lot of time. On the other hand, there are several things that cannot possibly be tested with JUnit tests, so there probably are a lot of Manual Test Cases to be written.

2.5.1. Running the manual tests

Anyone can run the manual tests on any version of ArgoUML. If it doesn't work, i.e. the expected result is not seen, then this is a defect in that version of ArgoUML and should be reported using Issuezilla.

At every release, the ambition is to run through all manual tests. Initially, when the amount of manual tests is small, this is done by the release responsible while testing the newly compiled release. Later on, when the amount of manual tests makes it unpractical to this during the release work, the work can be done by anyone, or any group of people within the project, after a development release is made and before a stable release is made. A signed statement with list of run tests including version number, a list (hopefully empty) of failed tests together with their Issuezilla DEFECT number, the host type, OS, jdk version, ArgoUML version, ... shall be mailed to the dev list when these tests are completed.

2.5.2. Writing the manual tests

Adding a new manual test to the group of already existing manual tests or improving one of the existing tests helps the project forward. Remember that the first priority is to test things with the JUnit tests because they can be, to some extent, run automatically and have their result reported automatically but then manual tests are the next big improvement.

Every test has several attributes to make sure that we can identify the test and help the developers and testers.

- A name

  This name is the title of the subsection where the test is described.

- A number

  These start with TEST1 and are allocated in sequence and maintained manually in this document (TEST2, TEST3, TEST4, ...). They are never reused when made available by removing a test case.

- A revision

  Every test case has a revision. These start with REV1 and are increased with one every time the test case is changed.

- A list of requirements tested

  This list is references to the requirements as stated in Chapter 3, *ArgoUML requirements*.

- Preparations i.e. what to do before the test

  This is Optional. The default is that you have just started ArgoUML.

- A description on what to do an what to expect

  This is a description in plain English telling the tester exactly what to do and what to expect. If this description doesn't work or is ambigous in any way the tester should consider the test to be DEFECT and report it in Issuezilla.

  This is probably best written like this:

  Do: whatever

  Expected output: whatever

  Do: whatever

  Expected output: whatever

2.5.3. The list of tests

This section contains all the tests each in a subsection of its own.

2.5.3.1. Modules are enabled

TEST1 REV1 (Does not test any current requirements.)

Preparations: Download and install ArgoUML together with the modules.

Do: Start in a window that allows you to see the output on Stdout.

Expected output:

```
Loaded Module: Java from classes
Loaded Module: GeneratorCpp
Loaded Module: GeneratorCSharp
Loaded Module: GeneratorPHP
```

Do: Press F7 (or select menu Generation => Generate All Classes...)

Expected output: A window pops up with Class Name, Java, Cpp, CSharp, and PHP.

Do: Select menu File => Import sources, then open the drop-down box Select language for import: to the far right.

Expected output: The drop-down box contains Java and Java from classes.

2.5.3.2. Class diagram

TEST2 REV1 (Requirements tested: 1 and 2)

Do: Select the Class Diagram. Click the Package symbol on the Edit pane toolbar. Click on the diagram. Click the Class symbol on the Edit pane toolbar. Click on the diagram. Click the Interface symbol on the Edit pane toolbar. Click on the diagram.

Expected output: The Class diagram and the explorer now contains one package, one class, and one interface.

Do: Select the class. Drag from the four quick-buttons located along the sides of the class and release somewhere on the diagram. Click on the fifth quick-button (buttom-left of the class). Select the interface. Drag from the quick-button located along the buttom of the interface symbol and release somewhere on the diagram.

Expected output: When releases on the diagram a new class is created both on the diagram, where released and in the explorer. The type of the association corresponds with the quick-button type. The association created when clicking the fifth quick-button goes back to the class itself.

2.6. Making a release

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, this list of what to do when releasing is maintained.

It is provided with the hopes of being helpful.

To understand this you need knowledge of how cvs works and how you normally build and test ArgoUML.

This instruction is supposed to work on a windows system (running build.bat). The author (Linus Tolke) has for some time been running it on a cygwin system (running build.sh) assuming that this will be the same as on any unix system. How it is actually run on a cygwin/unix system is also noted.

What needs to be done when one actually does a release:

1. Tag the whole CVS repository with the freeze tag!

   Normally this tag is "VERSION_*X*_*Y*_*Z*_F", e.g. VERSION_0_9_7_F. The according command line CVS command is **cvs rtag VERSION_*X*_*Y*_*Z*_F argouml**. (Because of a problem on the Tigris site, this doesn't work. Instead make sure you have a complete checked out copy of ArgoUML, go to the root directory `argouml` and run the command **cvs tag VERSION_*X*_*Y*_*Z*_F**.)

2. Check out a new copy of the source!

   This is done by checking out from the tag using the command **cvs co -r VERSION_*X*_*Y*_*Z*_F argouml_modules** and **cvs co -r VERSION_*X*_*Y*_*Z*_F argouml/tests** in a newly created directory.

   These commands assume that you have set the CVSROOT correctly. If not you will have to use commands like **cvs -d :pserver:*user*@cvs.tigris.org:/cvs co ...** instead.

3. Build the release!

   This is done in the `argouml/src_new` directory of the newly created copy by issueing the command **build dist-release**! (Linus: It takes around 10 minutes on my machine JDK1.3.1_01/700MHz/256MB (May 2003), It takes around 30 minutes on a Lysator machine simultaneously doing a lot of other things. JDK1.3.1_06/sun4d/256MB (July 2003).)

   On a Cygwin/Unix system you need to first make the **ant** executable with the command **chmod +x ../tools/ant-1.4.1/bin/ant** and then issue the command with **./build.sh** instead of **build**.

The output should be the files `ArgoUML-`*`VERSION`*`-libs.tar.gz`, `ArgoUML-`*`VERSION`*`-libs.zip`, `ArgoUML-`*`VERSION`*`-modules.tar.gz`, `ArgoUML-`*`VERSION`*`-modules.zip`, `ArgoUML-`*`VERSION`*`-src.tar.gz`, `ArgoUML-`*`VERSION`*`-src.zip`, `ArgoUML-`*`VERSION`*`-app.tgz`, `ArgoUML-`*`VERSION`*`.tar.gz`, and `ArgoUML-`*`VERSION`*`.zip` in the `argouml` directory in your new copy.

4. Test the release!

   Either the `ArgoUML-`*`VERSION`*`.tar.gz` or `ArgoUML-`*`VERSION`*`.zip` file is tested by unpacking, starting and then running through the test cases. Currently there isn't any defined test cases for manual testing.

   There are two sets of automatic test cases.

   - Run the JUnit test cases in `argouml/tests` by issueing the command **build alltests** in the `argouml/src_new` directory. (Linus It takes around 12 minutes on my machine JDK1.3.1_01/700MHz/256MB (May 2003), It takes around three hours on a Lysator machine simultaneously doing a lot of other things and the X session over a 50kb/s modem. JDK1.3.1_06/sun4d/256MB (July 2003).)

     There should not be any failed tests. (See details on where to find the result in Section 2.4, “The JUnit test cases”).

   - Run the JUnit test cases in `modules/junit` by cd:ing to `modules/junit` and running **build run**, invoking JUnit tests from the Tools menu, specifying the Test Case TestAll, and running without "Reload classes every run" checked. (See details in Section 2.4, “The JUnit test cases”).

     The corresponding `build.sh` is not available for a Cygwin/Unix system so you must run the ant command directly. First make the **ant** executable with the command **chmod +x ../../tools/ant-1.4.1/bin/ant** if you havn't made it above and then issue the command **../../tools/ant-1.4.1/bin/ant run** instead of the **build run** command.

     No problems shall be found.

   If the tests did not pass See Section 2.6.1, “The release did not work”.

5. Tag the whole repository with the release tag!

   This tag is "VERSION_*X*_*Y*_*Z*", e.g. VERSION_0_9_7. The according command line CVS command is **cvs tag VERSION_*X*_*Y*_*Z*** when your are standing in the `argouml`-directory.

6. Open the repository for commits towards the next version.

   This is done by setting the `argo.core.version` in `default.properties` to *`Number of next release`*, commiting and telling everyone on the developers mailing list. Notice that this cannot be done in the tagged copy but you either need to go back to your other working tree or need to check out the file `argouml/src_new/default.properties` specifically to do this.

7. Upload the files onto the tigris website!

   Only a project owner can do that. Please write the descriptions of the files like this:

   - libraries

     libraries needed to compile

   - sources

source code without libraries. If you want to build ArgoUML from source, the cvs version is recommended.

For the gnu version add also: Unpack with GNU-tar.

- complete set

  binary distribution, including all libraries

- application bundle

  binary distribution, runnable as application bundle

8. Contact Jason Robbins to make the new Webstart version available!

   Jason Robbins takes care of the signing and publishing of the Webstart version.

9. 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.

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

   d. Move all issues reported on 'current' to this release.

      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.

   e. Other stuff.

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

10. Update the web page

    The Lists of Issues [http://argouml.tigris.org/documentation/issuezilla/frequentlyusedlists.html] page contains a link to each version and needs to be updated.

11. Make announcements!

    Write a News announcements and a short note on the dev, users and announce lists. Announcer should make sure that he/she is already subscribed to all lists with a reference to the news item.

    The announcement shall include a statement on what kind of release this is, information on what has changed (for stable releases this is a list of what has changed since the last stable release), the list of resolved issues, a list of serious known problems with this release (stable releases shouldn't have any), technical details on how the release was built, and the plan for the following release.

    Freshmeat: currently Thierry Lach does the freshmeat announcements which require a login so just inform him.

2.6.1. The release did not work

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.

2.6.1.1. 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

  cvs rtag -b -r VERSION_*X_Y_Z*_F BRANCH_*X_Y_Z*

- Update your checked out copy to be on that branch

  cvs update -r BRANCH_*X_Y_Z*

- Fix the problem in your checked out copy

- Commit the problem in the branch

  cvs commit -m'Fix of problem blabla'

- 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.

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

Create the branch as described in Section 2.6.1.1, “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 (REV1, REV2, REV3, ...) 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.

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

   REQ1 REV1

   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

2. As soon as the model element changes then all views shall update. For text fields, this can be at every key stroke.

   REQ2 REV1

   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. 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 REV1

   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

4. All text fields shall have context sensitive help.

As follows:

a. 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.

b. 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 REV1

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

1. ArgoUML shall be a correct implementation of the UML 1.3 model.

   REQ5 REV1

   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 1.3 but there shall never be any doubt that the intention of ArgoUML is to implement UML correctly.

   Stakeholder: User of ArgoUML

2. ArgoUML shall implement everything in the UML 1.3 model.

   REQ6 REV1

   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

1. Choice of JRE

   We will support any JRE compatible with the Sun specification one version behind the most recent stable JRE from Sun. A stable JRE is considered to be one that has had a second non-beta release.

This is to allow ArgoUML to gradually take on board new stable features of the Java language while still offering users some choice of JRE.

REQ7 REV1

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.

Interpretation: This means that we currently want to support JREs 1.3.0, 1.3.1, 1.4.0, and 1.4.1. When a JRE compatible to Sun JRE 1.5.1 has come out for all major platforms: Solaris, Linux, Windows, Mac, support for 1.3.0 and 1.3.1 will be discontinued.

Stakeholder: Developers of ArgoUML

2. Download and start

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

   REQ8 REV1

   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. 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 REV1

   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

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 REV1

   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 components that each have a responsibility.

Chapter 5, *Inside the components* explains each component in details. This chapter just gives an overall picture.

The components are organized in layers. The purpose of the layers is to keep a clear view of what components provide services to others and to allow us to know how much is involved when testing each component.

TODO: Insert UML diagram describing the relation between components and layers.

This chapter contains a list of all components and what layer they are in and the definition of the responsibility of each component.

4.1. Definition of component

All ArgoUML code is organized in components.

Each component has:

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

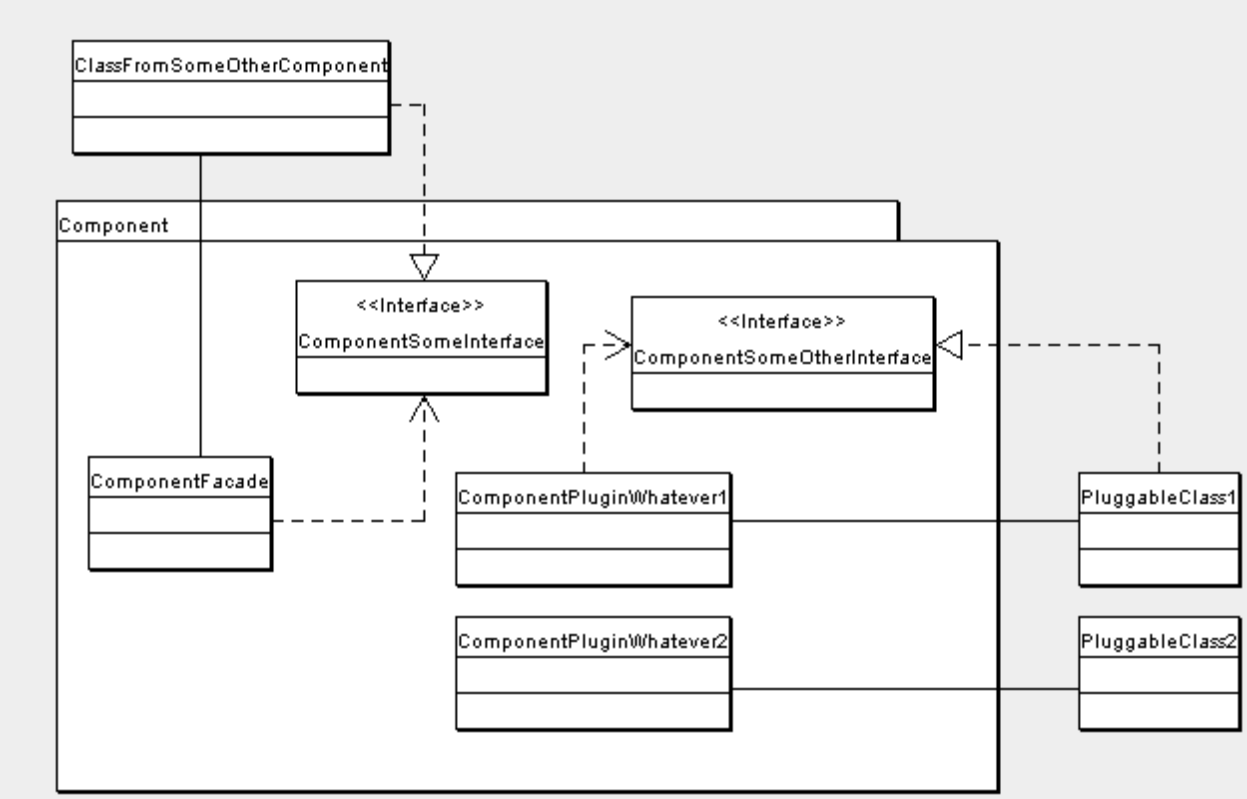
  Subparts of the component can reside in subdirectories of this directory. Auxiliary parts of the components can reside somewhere else.

Each component has a single Facade class that can be used by all other components when using the component. The Facade class is called *`ComponentName`*`Facade` and is located in the component 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 components*).

Each component can also have one or several plug-in interfaces. That is Facade objects where modules or plug-ins can connect themselves to modify or augment the behavior of that component.

The plug-in interfaces are also all located in the component package and called *`ComponentName`*`Plugin`*`Plug-inType`*. Example: `ModelPluginDiagram`, `ModelPluginType`.

If the component uses a callback-technique the callback is always made to an interface. The interface is also in the component package and it is called *`ComponentNamePlug-inType`*`Interface`. Example: `ModelDiagramInterface`, `ModelTypeInterface`.



4.2. Relationship of the components

Each component that is used by other components provide two ways for other components 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 components were clearly defined.

  A Facade class provides the most common functions other components want to do when using that components 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 component 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 components*).

- Calls to public methods

  Traditionally components interface through public methods and public variables. For this reason, always exercise extreme caution when changing the signature of a public method. (See Section 8.1, “How to work against the CVS repository”.)

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

For each component X in ArgoUML that uses the component Y the designer of that component X, must decide if he wants to use calls to public methods when using the component Y (putting a set of import

org.argouml.Y.internals.blabla.*; statements in each file in the files of component X that uses component Y) or just use the Facade class of component Y (putting only one import org.argouml.Y.YFacade; in each file in the component X that uses component Y).

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

If the public calls solution or facade calls solution is used shall be described in the Cookbook's description of component X in the list of used components.

4.3. Definition of layer

Layers are used to organize and clairify the relationships between the different components within ArgoUML.

ArgoUML is built from the bottom and up. Components on a higher level are relying on components on a lower level and never the other way around. A component cannot even rely on a component in the same layer.

This means that when testing a component, it can always be tested with just that component and components on lower levels.

4.4. Layer 0 - Description of components

Layer 0 contains some infrastructure components that just are there for every other layer to use.

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

- Logging

  Calls can be spread all over that would go through some rule set and then end up on file, on the output or not at all.

- Internationalization

  This is the set of files that is a repository of localized strings. Every other module uses these strings in all communcations with the user.

  The Internationalization Component is described in detail in Section 5.11, “Internationalization”.

- JRE with utils

  Every other component can use the classes available with the JRE.

4.5. Layer 1 - Description of components

Layer 1 is the lowest layer. The components in this layer do not rely on any other part (except layer 0) of ArgoUML

to do their work. They can all be tested in full individually i.e. independant of any other component.

- The Model

  The Model contains a modifyable view of the UML model and the diagrams.

  The Model presents several different views and access methods for the information. Among other things, the information can be saved, loaded, examined, and observed.

  The Model is described in detail in Section 5.1, “Model”.

- To do items

  This is the To do items. They can be created, deleted and saved.

  The To Do Items Component is described in detail in Section 5.14, “To do items”.

- The GUI Framework

  This is the framework with menus, tabs, and panes available for the other components to fill with actions and contents.

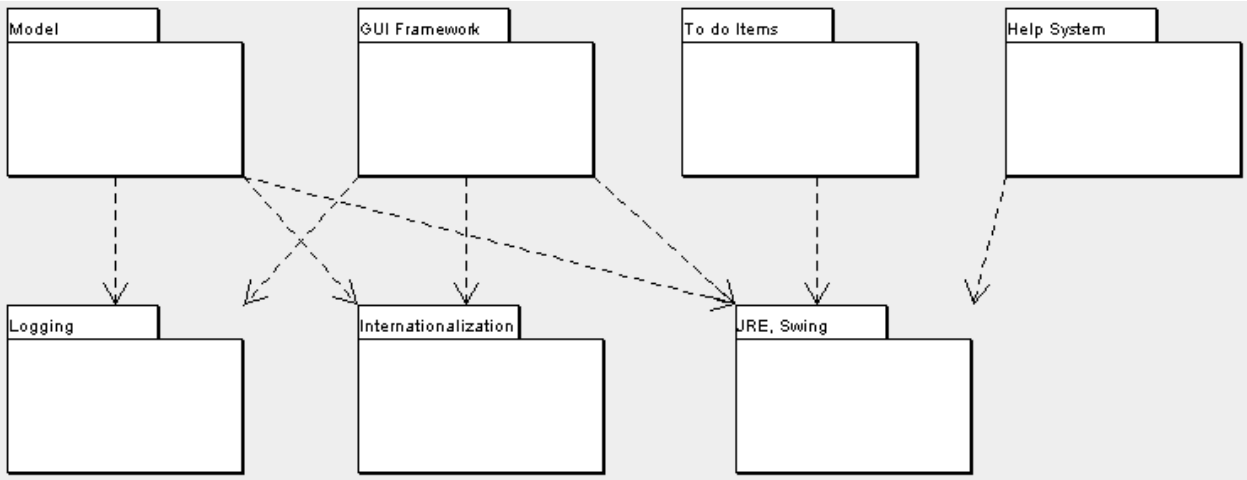
  The GUI Framework Component is described in detail in Section 5.9, “The GUI Framework”.

- Help system

  Not yet implemented.

  This is the component that the other components can call to present some help for the user.

  The Help System Component is described in detail in Section 5.10, “Help System”.



4.6. Layer 2 - Description of components

These components rely on components of layer 1 in order to do their work.

- Diagrams

  This is the diagram view of the model. The notation is a property that belongs in the Diagrams so the different language register their provided notation in the Diagrams component.

  The Diagrams Component is described in detail in Section 5.3, “Diagrams”.

- Property panels

  This is the prop panel view of the model.

  The Property Panels Component is described in detail in Section 5.4, “Property panels”.

- Explorer

  This is the tree view of the model.

  The Explorer Component is described in detail in Section 5.15, “Explorer”.

- Code Generation

  This is the common code for and the point where each language with Code Generation possibility registers.

  The Code Generation Component is described in detail in Section 5.6, “Code Generation Component”.

- Reverse Engineering

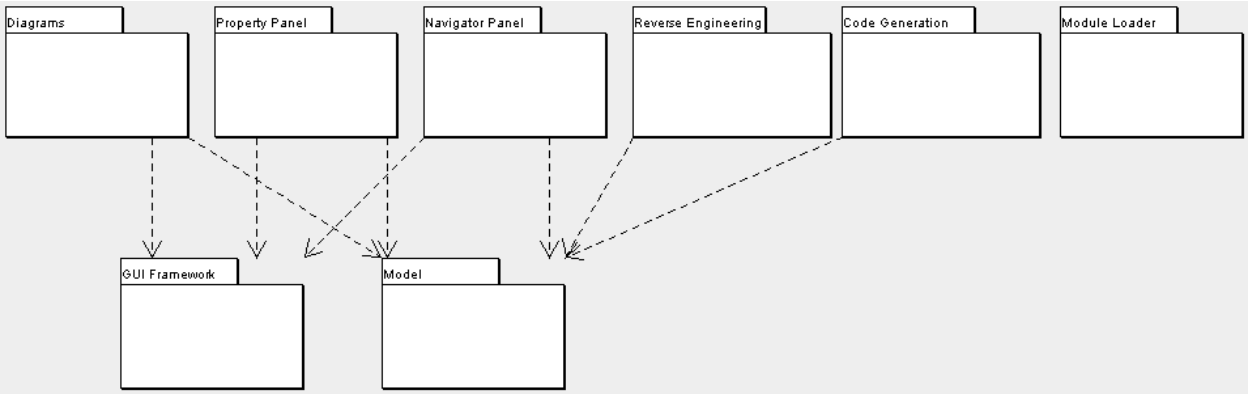
  This is the common code for and the point where each language with Reverse Engineering possibility registers.

  The Reverse Engineering Component is described in detail in Section 5.5, “Reverse Engineering Component”.

- Module loader

  This is the load mechanism for loading all Layer 3 components and other modules into ArgoUML.

  The Module Loader Component is described in detail in Section 5.16, “Module loader”.



4.7. Layer 3 - Description of components

These components are primarily connected through the pluggable interfaces meaning that they can be individually disabled using the module loader.

- Java Code generation, Reverse engineering

  This is the ArgoUML connection to the java language.

  The Java Component is described in detail in Section 5.7, “Java - Code generations and Reverse Engineering”.

- Other languages - Code generation, Reverse engineering

  Languages are plugged into the notation, the import (reverse engineering), and code generation.

  See Section 5.8, “Other languages”.

- Critics and checklists

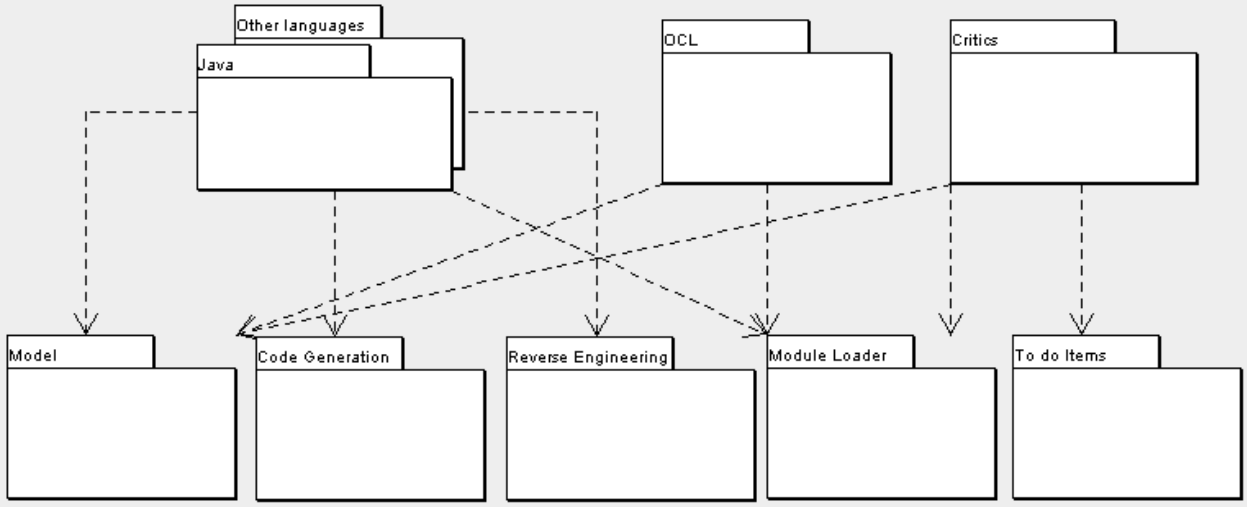
  This is the critics.

  The Critics Component is described in detail in Section 5.2, “Critics and other cognitive tools”.

- OCL

  This is the editing of the OCL strings.

  The OCL Component is described in detail in Section 5.17, “OCL”.



Chapter 5. Inside the components

Warning

This chapter is currently under rework with new component organization.

...

5.1. Model

Purpose - to provide the data structures that keep track of the model and the diagrams. This comes with a complete set of methods to modify the model and register interest in changes to the model.

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

The Model is a Layer 1 component. See Section 4.5, “Layer 1 - Description of components”.

This is implemented using NSUML to implement the UML model.

ArgoUML uses several factories and helperclasses to manipulate the NSUML model. The NSUML model itself does not define enough 'business' logic to be directly used and the factories and helperclasses provide a centralized place for accessing this 'business' logic. Per section of chapter 2 of the UML 1.3 specification there is one factory and one helper. They are placed in their own packages. The package name convention is: `org.argouml.model.uml.SECTIONNAME` where sectionname is one of the following:

- foundation
- foundation.core
- foundation.extensionmechanisms
- foundation.datatypes
- behavioralelements
- behavioralelements.commonbehavior
- behavioralelements.statemachines
- behavioralelements.usecases
- behavioralelements.collaborations
- behavioralelements.activitygraphs
- modelmanagement

Each package has at least a helper and a factory in it. The factories contain all methods that deal with creating and building modelelements. The helpers contain all utility methods needed to manipulate the modelelements.

Both helpers and factories are singletons. The static method to access them is `getFactory` for the factory and `getHelper` for the helper.

5.1.1. Factories

The factories contain at least for each modelelement a create method. Example: `createClass` resides in `CoreFactory` in the package `org.argouml.model.uml.foundation.core`. Besides that, there are several buildmethods to build classes. The build methods have a signature like

```
public MODELELEMENT buildMODELELEMENTNAME(params);
```

.

Each build method verifies the wellformednessrules as defined in the UML spec 1.3. The reason for this is that NSUML does not enforce the wellformedness rules even though non-wellformed UML can lead to non-wellformed 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 create methods in the factories. You are strongly advised to use a build method or, if there is none that suits your needs, to build one thereby reusing the allready existing build methods and utility methods in the helpers. One reason for this is that the eventlisteners for the newly created modelelement are setup correctly.

TODO: Am I allowed to call the factories from any thread?

5.1.2. Helpers

The helpers contain all utility methods for manipulating modelelements. For example, they contain methods to get all modelelements 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 modelelement 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 modelelements 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.

TODO: Am I allowed to call the helpers from any thread?

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:

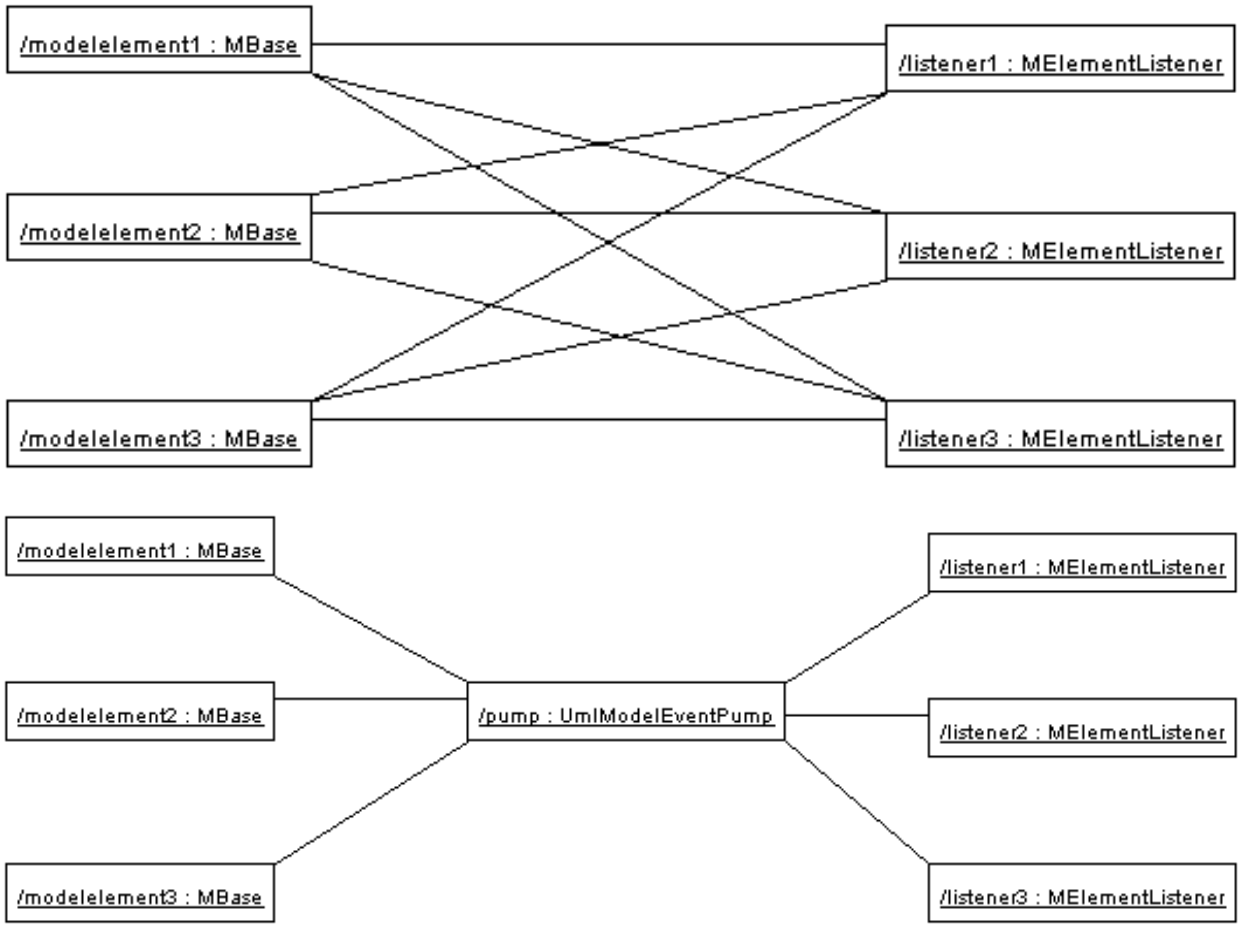
1. The model factories, responsible for creation and deletion of modelelements
2. The model helpers, responsible for utility functions to manipulate the modelelements and
3. The model event pump, responsible for sending model events to the rest of ArgoUML.

In other paragraphs the model factories and the model helpers are already introduced. Therefore we won't discuss them here again.

The model event pump is the gateway between the model elements and the rest of ArgoUML. Events fired by the modelelements are catched 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 comparision 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 modelelement.



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 modelevent (from now on a MEvent) has a name that uniquely identifies the type of the event. In most cases the name of the MEvent 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 MEvent and the property changed in the model. Therefore most listeners that need MEvents are only interested in one type of MEvent 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 Sec-

tion 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 modelelement or even for all events fired by a certain type of modelelement. 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(MElementListener listener, MBase modelelement, String eventName)
```

like this:

```
UmlModelEventPump.getPump().addModelEventListener(this, modelelementIAmInterestedIn, "IamInterestedI
```

Now your object this gets only the `MEvents` 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(MElementListener listener, MBase modelElement, String eventName)
```

on `UmlModelEventPump` like this:

```
UmlModelEventPump.getPump().removeModelEventListener(this, modelelementIAmInterestedIn, "IamInterest
```

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 modelelements. This works with the method:

```
addModelEventListener(MElementListener listener, MBase 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 modelelement. 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(MElementListener listener, MBase modelelement, String[] eventNames)
```

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

Thirdly there is a very powerfull method to register your listener to ALL events fired by a ALL modelelements of a certain class. You can understand that using this method can have severe performance impacts. Therefore use it with care. The method is:

```
addClassModelEventListener(MElementListener listener, Class modelClass)
```

There are also methods that allow you to register only for one type of event fired by all modelelements of a certain

class and to register for a set of types of events fired by all modelements 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 eventpump 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. The future

Some people might wonder if we cannot make a better interface between the model and the rest of ArgoUML if we use our own event types. These people are right. There are plans to replace the MElementListener with PropertyChangeListeners and using the PropertyChangeEvents instead of the MElementEvent. This has not been done yet since this involves a lot of work and testing.

Next to this, it is likely that the implementation of the Event pump itself is going to change. 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).

Besides that, the current implementation does not use the standard data structure that Swing uses for event registriation (i.e. `javax.swing.EventListenerList`). We are in the process of researching if it is a good thing to use this standard instead of our own implementation.

5.1.4. How do I...?

- ...add a new model element?

  Make a parameterless build method for your NSUML modelelement in one of the UML Factories (for instance `CoreFactory`). Stick to the UML 1.3 spec to choose the correct Factory. The package structure under `org.argouml.model.uml` follows the chapters in the UML spec so get it and read it! In the buildmethod, create a new modelelement 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 modelelements. 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 modelelements to your modelelement make a

`buildXXXX(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 modelelements 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 5.3.2.5, “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.

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 Layer 3 component. See Section 4.7, “Layer 3 - Description of components”.

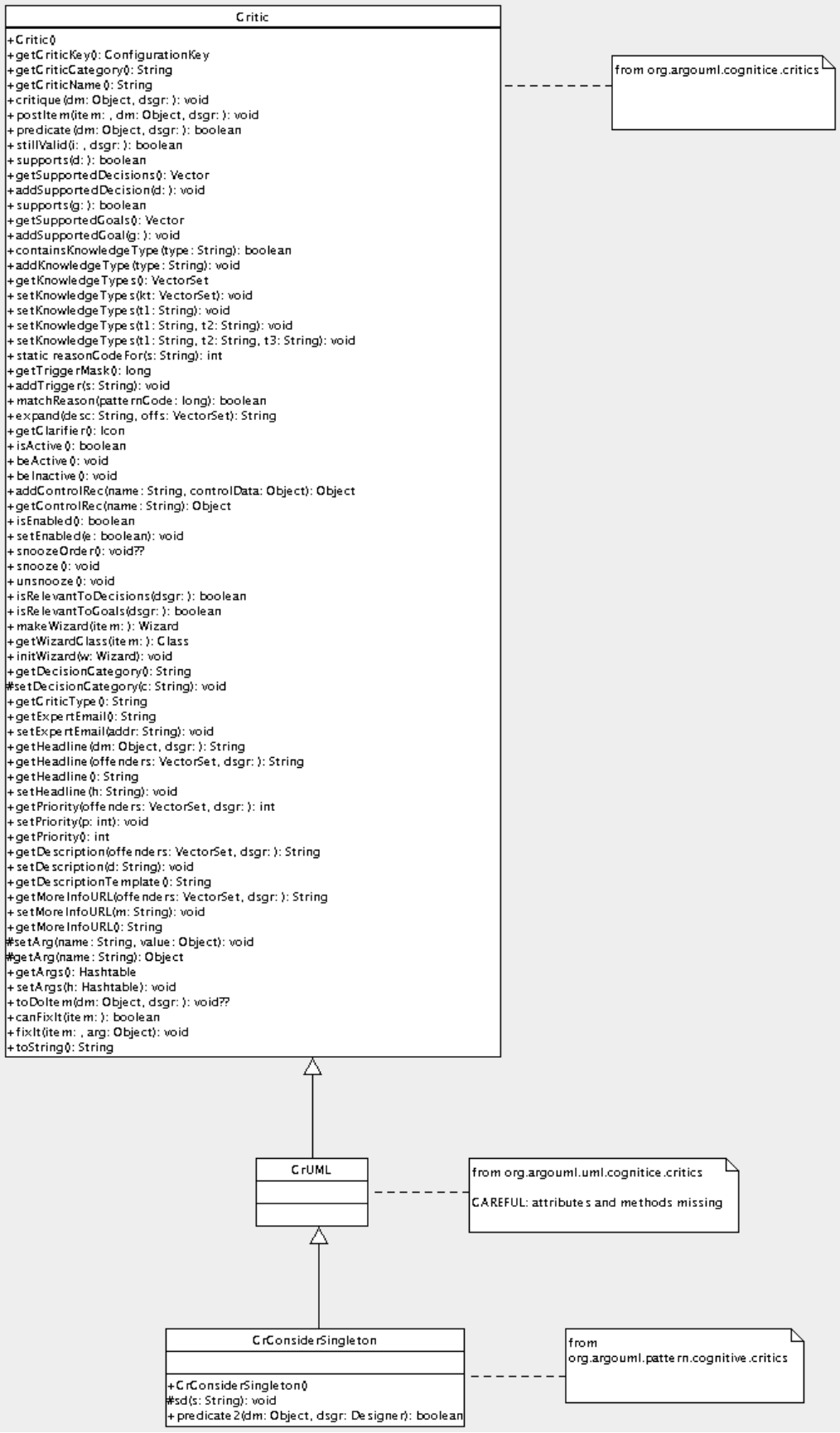
The Critics component depends on the Model that it works against to take all decisions and the To Do Items used to present the information.

This component contains the following main class types:

- Critics provide help to find artifacts in the model that do not obey simple design "rules" or "best practises".
- 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.pattern.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 Modelelements 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 agains modelling 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 argouments and calls the following methods of `CrUML`:

  - **setResource("Cr*XxxxYyyyZzzz*");** 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).
  - **addSupportedDecision(CrUML.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 MetaClass*");** where *UML MetaClass* is a UML MetaClass, with initial lower capital, e.g. "associationEnd". The intention is that critics should only trigger for elements (or children) of particular UML metaclasses. 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 metaclasses.

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 (an NSUML 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 `UMLCognitiveResourceBundle.java`. You need to add two keys for the head and description, which will be named respectively `Cr`*`XxxxYyyyZzzz`*`_head` and `Cr`*`XxxxYyyyZzzz`*`_desc`. There are plenty of examples to look at there. The other files `UMLCognitiveRe-` (`sourceBundle_en_GB.java`, `UMLCognitiveResourceBundle_es.java`, ... 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:

```
  public 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`.

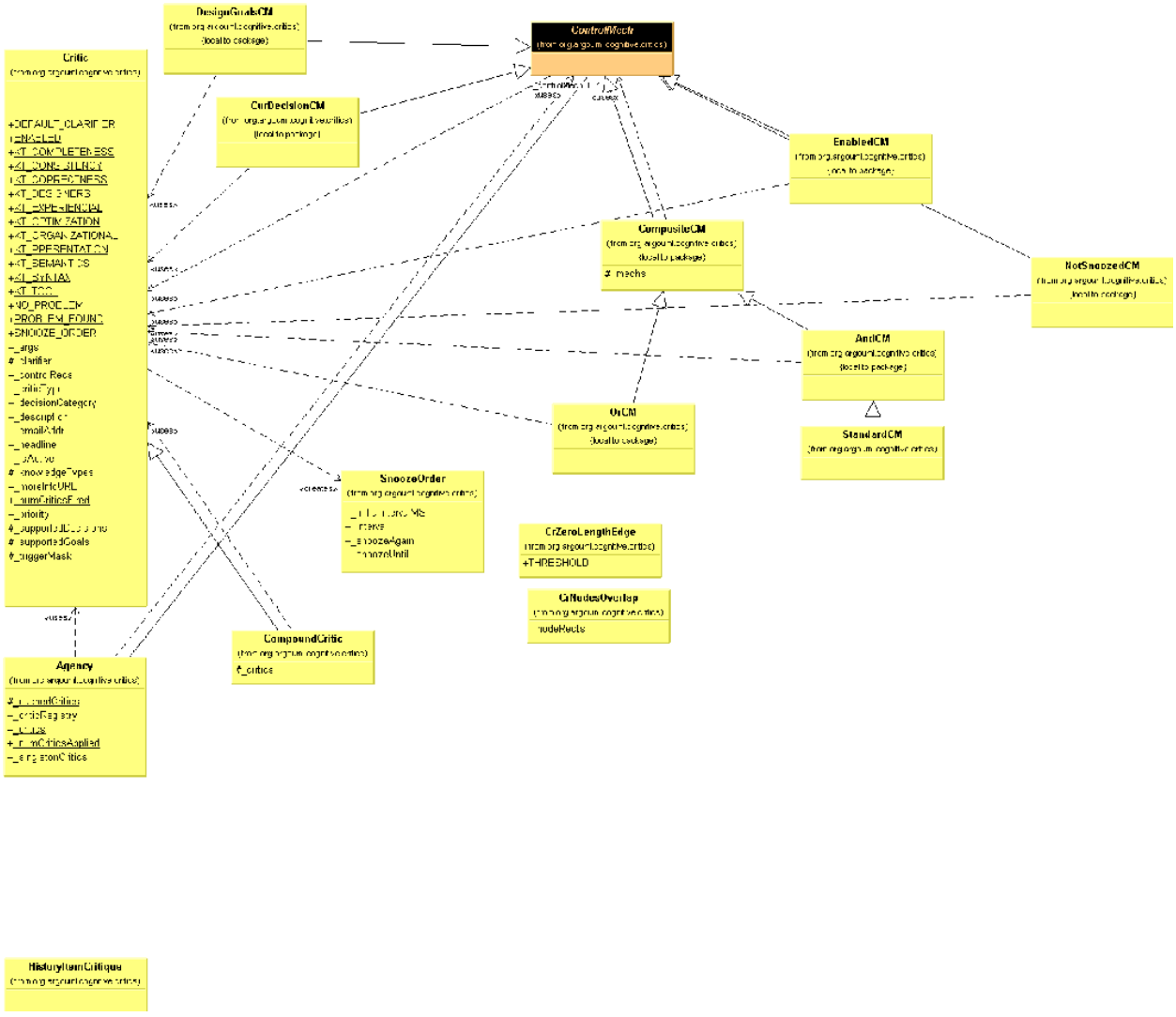
- ...fix a critique?

  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 NSUML 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.

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



5.3. Diagrams

Purpose - to generate a graphical view of the diagrams in the model with tools. The contents of the diagrams and

model is modifyable. TODO: Notation!

The Diagrams will be located in `org.argouml.???`.

The Diagrams is a Layer 2 component. Section 4.6, “Layer 2 - Description of components”.

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

5.3.1. 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 and renderer.
2. Create a new property panel class that will be displayed in the property tab windown 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 labelled «`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.

5.3.2. How to add a new Fig

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

5.3.2.1. Adding to the toolbar

Find the diagram object in `uml/diagram/XXXX/ui/UMLYYYYDiagram.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 toolbar action is declared as a protected 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.

```
protected static Action _actionUseCase =
    new CmdCreateNode(MUseCaseImpl.class, "UseCase");
```

The first argument is the class of the node to create from NSUML, the second a textual tool tip.

For creating associations we have:

```
protected static Action _actionAssoc =
      new CmdSetMode(ModeCreatePolyEdge.class,
                     "edgeClass", MAssociationImpl.class,
```

```
                    "Association");
```

The first argument is a GEF class that defines the type of behavior wanted (in this case creating a poly-edge). The second and third arguments are a named parameter used by `ModeCreatePolyEdge` ("edgeClass") and its value (`MAssociationImpl.class`). The final argument is a tooltip.

The toolbar is actually created by defining a method, `initToolBar()` which adds the tools in turn to the toolbar (a protected member named `_toolBar`).

The default constructor for the diagram is declared private, since it must not be called directly. The desired constructor takes a namespace as an argument, and sets up a graph model (`UseCaseDiagramGraphModel`), layer perspective and renderer (`UseCaseDigramRenderer`) for nodes and edges.

5.3.2.2. 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 a child of the GEF class `MutableGraphSupport`, and should implement `Mutable-GraphModel` (GEF), `VetoableChangeListener` (Java) and `MElementListener` (NSUML).

5.3.2.3. 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 `YYYYDiagramRenderer.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 NSUML node object and `getFigEdgeFor()`, which provides a fig object to represent a given NSUML edge object.

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

5.3.2.4. 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 must provide constructors for both a generic fig, and one representing a specific NSUML 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 model has changed.

5.3.2.5. Creating a new Fig (explanation 2)

Assuming you have your modelelement already defined in the model and your PropPanel for that modelelement 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 modelelement 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 overriden method `protected void modelChanged()`. 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 `textEdited(FigText 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: delegate to ParserDisplay. See the method `parseAttribute` in ParserDisplay for an example. Stick to the Notation you are using to have the right parsing scheme. There is work to be done here but please don't make it an even bigger mess :)

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 implenting 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 *`modelelement`*`.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 modelelement. Probably the parent of your modelelement (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 PropPanels, most of them don't use actions, they implement them directly as methods in the PropPanel themselves. Please don't do that but use an action so we have one place of definition.

5.4. Property panels

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

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

The Property panels is a Layer 2 component. See Section 4.6, “Layer 2 - Description of components”.

Currently the PropPanels for the diagrams are in `org.argouml.uml.diagram.ui` and the property panels for the other object are in `org.argouml.uml.ui.`*`NS-UML path`*.

5.4.1. Adding the property panel

Warning

This description is old and the property panels has 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.

Property Panels are found as class PropPanelXXX.java, where XXX is the UML metaclass. They are in sub-packages of org.argouml.uml.ui corresponding to the XXX NSUML packages, which in turn correspond to their section in the chapter 2 of the UML 1.3 spec. This packaging is essential for their lookup through Java reflection.

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 NSUML classification are provided in org.argouml.uml.ui.

Typically the constructor for the new 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, it should be placed in `org/argouml/Images` and a call to `ResourceLoader.lookupIconResource()` made (note this is a method of a GEF class). This is usually added to `PropPanelModelElement`. For our example we have had to add `Extend.gif`.

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 property panel is created as a grid with a predefined number of columns (2 if there are only a few fields, 3 if there are a lot). Into each row of each column is placed a caption and a corresponding field.

Adding a caption or field is through one of a small number of utility methods which require you to specify which column and which row and also a `weighty` parameter to specify the amount of padding to be added when fields are stretched to fit a column. Vertical padding is distributed in proportion to `weighty` amongst all fields in the column that have non-zero `weighty` values.

Tip

You should always ensure at least one field or caption in each column has a non-zero value for `weighty`. If you wish everything fixed size and floated to the top, make the value for the final caption in the column non-zero.

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 `UMLTextField` class.
- A drop down box of options that can be selected, with an icon to the right allowing navigation to the property panel for the currently selected item. For example the Stereotype field. Supported in general by the `UMLComboBox` class and more speficically by its subclass for stereotypes, `UMLStereotypeComboBox`.
- A non-editable text box, with a pop-up menu that allows opening, addition, deletion, moving up and moving down of entries. For example the Generalizations field. Supported by the `UMLList` class. The list model is usually provided by a sub-class of `UMLModelElementListModel`. There is a varient `UMLModelElementListLinkModel` which adds a link option to the pop-up menu, allowing connection to existing model elements (used for the Extension Points field for example).
- A set of check boxes for modifiers. Supported by the `PropPanelModifiers` class.

Examples of these in more detail now follow.

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`), possibly empty, or extends relationships from this use case.

Rather than a straight `JList`, we use its child, `UMLList`, which implements the `MouseListener` and NSUML `ElementListener` interfaces.

The constructor for `UMLList` requires two arguments, a list model and a flag to indicate whether the list is navigable, i.e. responds to the mouse.

The list model should be a subclass of `UMLModelElementListModel`, a subclass of the Swing `AbstractListModel` that implements the NSUML `ElementListener` interface.

5.4.1.1.1. The list model

In our example we create UMLExtendListModel. Its constructor should take three arguments:

1. The container, where this list is being built. I.e. the `PropPanelUseCase` (from which we can then derive the NSUML `MUseCase`, which is the “target” of the extends relationship).
2. A string naming an NSUML event that should force a refresh of the list model. A null value will cause all events to trigger a refresh. The best way to identify the event you want to use is to look at the NSUML source for the container object (`MUseCaseImpl` in our example) for calls to `fireXXX()`. The first argument is the name of the event (in our case `extend`). There is no definitive list, but from the NSUML source, these are all the names of events that are used:

    - action
    - actionSequence
    - activator
    - activityGraph
    - actualArgument
    - addition
    - aggregation
    - alias
    - annotatedElement
    - argument
    - association
    - associationEnd
    - associationEndRole

- associationRole
- attribute
- attributeLink
- availableContents
- availableFeature
- availableQualifier
- base
- baseClass
- baseElement
- behavior
- behavioralFeature
- binding
- body
- bound
- callAction
- changeability
- changeExpression
- child
- classifier
- classifierInState
- classifierRole
- classifierRole1
- client
- clientDependency
- collaboration
- collaboration1
- comment
- communicationConnection
- communicationLink
- componentInstance

- concurrency
- condition
- connection
- constrainedElement
- constrainedElement2
- constrainingElement
- constraint
- container
- contents
- context
- createAction
- defaultElement
- defaultValue
- deferrableEvent
- deploymentLocation
- discriminator
- dispatchAction
- doActivity
- dynamicArguments
- dynamicMultiplicity
- effect
- elementImport
- elementImport2
- elementResidence
- entry
- event
- exit
- expression
- extend
- extend2

- extendedElement
- extender
- extenderID
- extension
- extensionPoint
- feature
- generalization
- guard
- icon
- implementationLocation
- include
- include2
- incoming
- initialValue
- instance
- instantiation
- inState
- interaction
- internalTransition
- isAbstarct
- isAbstract
- isActive
- isAsynchronous
- isConcurent
- isDynamic
- isInstantiable
- isLeaf
- isNavigable
- isQuery
- isRoot

- isSpecification
- isSynch
- kind
- link
- linkEnd
- location
- mapping
- message
- message1
- message2
- message3
- message4
- method
- modelElement
- modelElement2
- multiplicity
- name
- namespace
- nodeInstance
- objectFlowState
- occurrence
- operation
- ordering
- outgoing
- ownedElement
- owner
- ownerScope
- package
- parameter
- parent

- participant
- partition
- partition1
- powertype
- powertypeRange
- predecessor
- presentation
- qualifiedValue
- qualifier
- raisedSignal
- receiver
- reception
- recurrence
- referenceState
- representedClassifier
- representedOperation
- requiredTag
- resident
- residentElement
- script
- sendAction
- sender
- signal
- slot
- source
- sourceFlow
- specialization
- specification
- state
- state1

- state2
- state3
- stateMachine
- stereotype
- stereotypeConstraint
- stimulus
- stimulus1
- stimulus2
- stimulus3
- structuralFeature
- subject
- submachine
- submachineState
- subvertex
- supplier
- supplierDependency
- tag
- taggedValue
- target
- targetFlow
- targetScope
- templateParameter
- templateParameter2
- templateParameter3
- top
- transition
- trigger
- type
- useCase
- value

- visibility
- when

3. A flag to indicate that a label “none” should be used when the list is empty.

Quite usually it is sufficient to just invoke the constructor of the parent class.

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 int recalcModelElementsSize()` Return the number of elements in the list (zero if empty).

`protected MModelElement getModelElementAt(int index)` Return the element at the given index in the list, or null if there isn't one.

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 buildPopUpForIndex(UMLPopupMenu popup, int index)` Build the popup 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 NSUML entity. There seem to be three ways to do this, in order of preference 1) use a utility from the `MMUtil` class, 2) use the NSUML Factory class to create what you want 3) use new on a `MXXXImpl` class. Whilst 1) is best, most of the `MMUtil` routines are not yet general enough.

Be sure to set it up (don't forget e.g namespace etc). Remember also to change anything that references the newly created entity.

Warning

The 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)`
: part of the NSUML MElementListener interface. Called when an add event happens, i.e. some NSUML 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)`
: part of the NSUML MElementListener interface. Called when a remove event happens, i.e. some NSUML 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)`
: these are all required as part of the NSUML MElementListener 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 element)`
: 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 NSUML 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 NSUML object associated with the container (some child of `PropPanel` usually) that holds this list model.
`final UMLUserInterfaceContainer getContainer()`	returns 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 moveDownCollection(Collection oldCollection, Object item)`	helps in adding the "add" function to the list. Swaps the element, item, with index+1 in the given oldCollection.
`static protected java.util.List moveDownList(List theList, int theIndex)`	helps in writing the "move down" function. Swaps the elements at theIndex and index-1. Not clear why it doesn't return a Collection.
`static protected java.util.List moveUpList(List theList, int theIndex)`	helps in writing the "move up" function. Swaps the elements at theIndex and index-1. Not clear why it doesn't return a Collection.
`static protected MModelElement getElementAt(Collection c, int index, Class requiredClass)`	Helper method for getElementAt(). Finds the element at a specified 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 scrollpane.

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 Toolbar 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 `UMLCombBoxModel`: `UMLCombBoxModel(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 NSUML `MElementEvent` 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 comboBox (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 NSUML 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 everytime 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 UMLUserInterfaceComponents interested in this event, including UMLTextField. It will also dispatch the event to all childs 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 everytime 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 userinput 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 userinput, it is checked if it is valid input. If it is not, a JOptionPane is shown with ' a warning and the change is not commited into the model. If it is not via userinput, 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 userinput 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 classdiagram since handleEvent would popup a warning messagebox.

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. Reverse Engineering Component

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 Component is a Layer 2 component. See Section 4.6, “Layer 2 - Description of components”.

5.6. Code Generation Component

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 component is a Layer 2 component. See Section 4.6, “Layer 2 - Description of components”.

5.7. 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 coverted into a model.

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

The Java component is a Layer 3 component. See Section 4.7, “Layer 3 - Description of components”.

5.7.1. How do I ...?

...

5.7.2. Which sources are involved?

The package org.argouml.uml.reveng is supposed to hold those classes that are common to all 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 used to visualize generated NSUML metamodel objects then.

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...

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

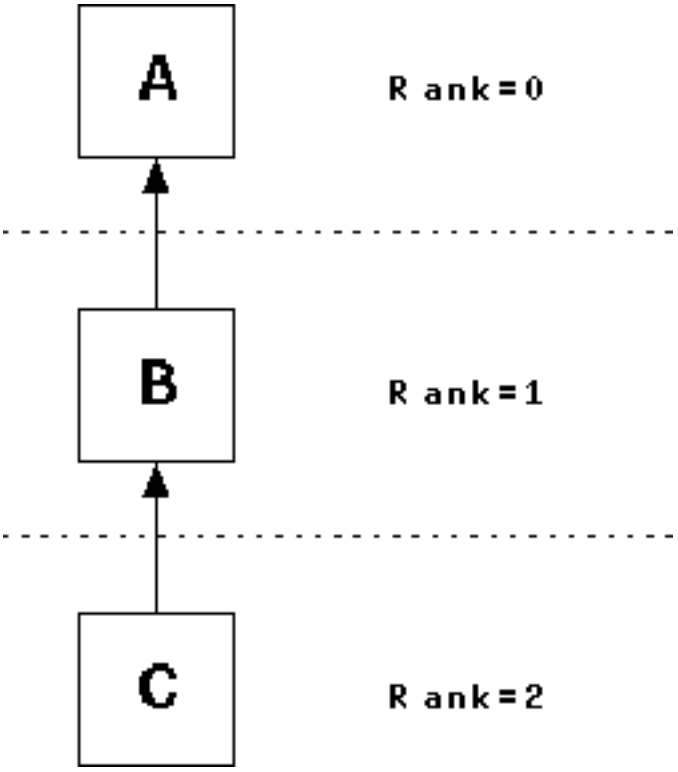
It's an Antlr (http://www.antlr.org) grammar, based on the Antlr Java parser example. The main difference is the missing AST (Abstract Syntax Tree) generation and treeparser. So the original example generates an AST (a treelike data structure) and then traverses this tree, while the ArgoUML code parses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

5.7.4. Which model/diagram elements are generated?

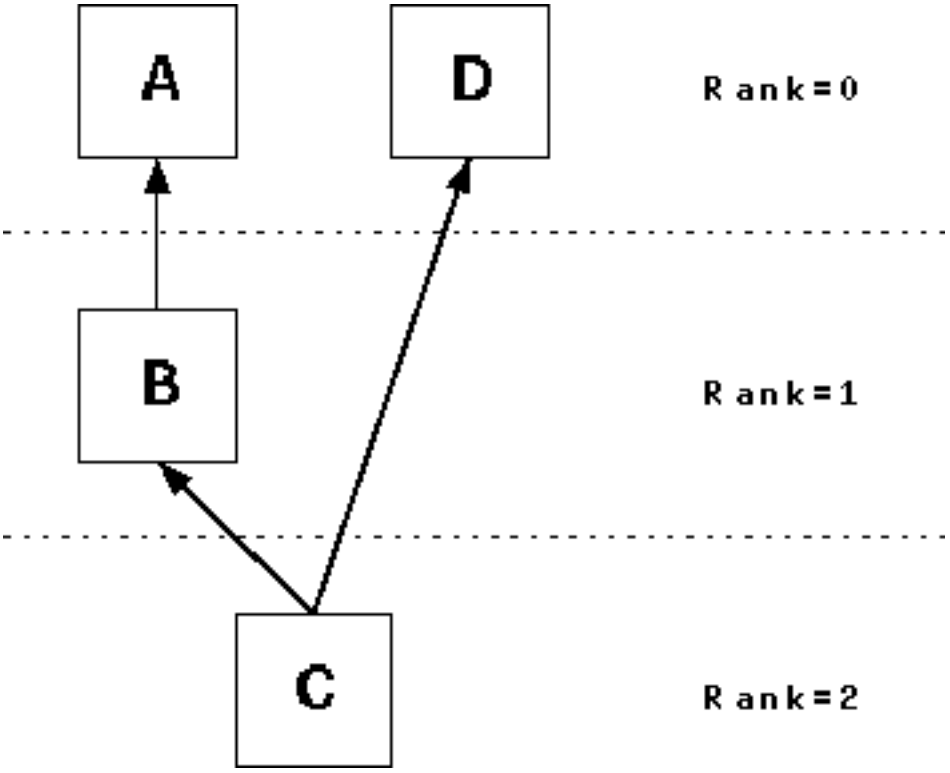
The *context classes hold the current context for a package, class etc. When the required information for an object is available, the corresponding NSUML object is created and passed to the DiagramInterface to visualize it.

5.7.5. Which layout algorithm is used?

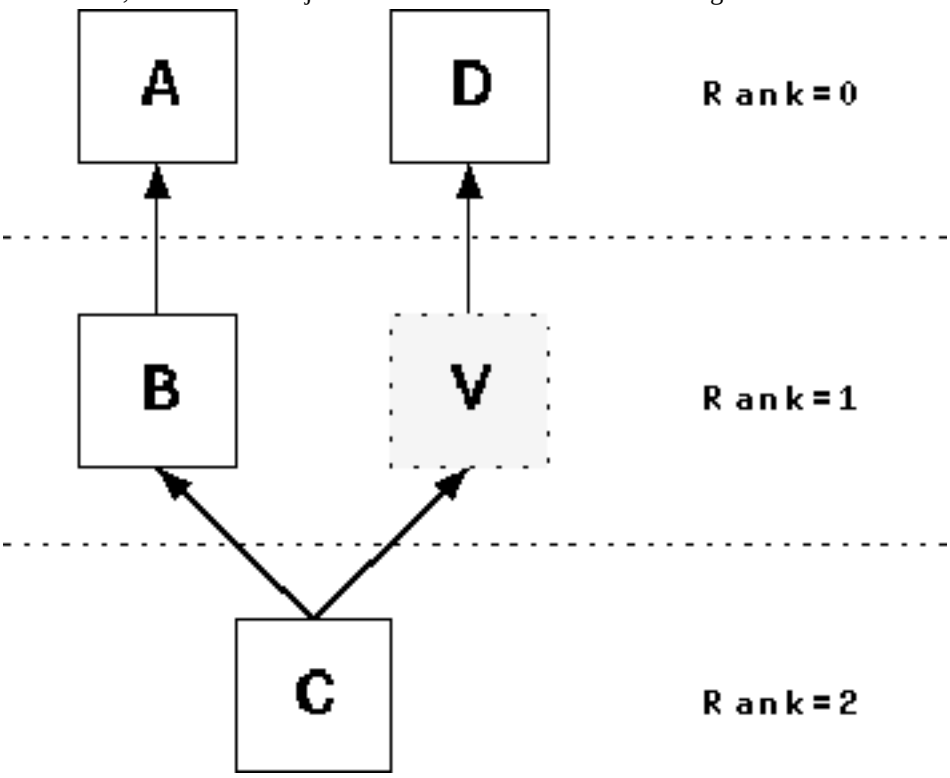
The classes in org.argouml.uml.diagram.static_structure.layout.* hold the Classdiagram layout code. No layout for other diagram types yet. It's based on a ranking scheme for classes and interfaces. The rank of a class/interface depends on the total number of (direct or indirect) superclasses. So if class B extends A (with rank(A)=0), then rank(B)=1. If C extends B, then rank(C)=2 since it has 2 superclasses 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 diagramm, 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 superclasses 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 superclasses here):

D -> C

E -> A and C

F -> A and B

Compute the average value of the indices:

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

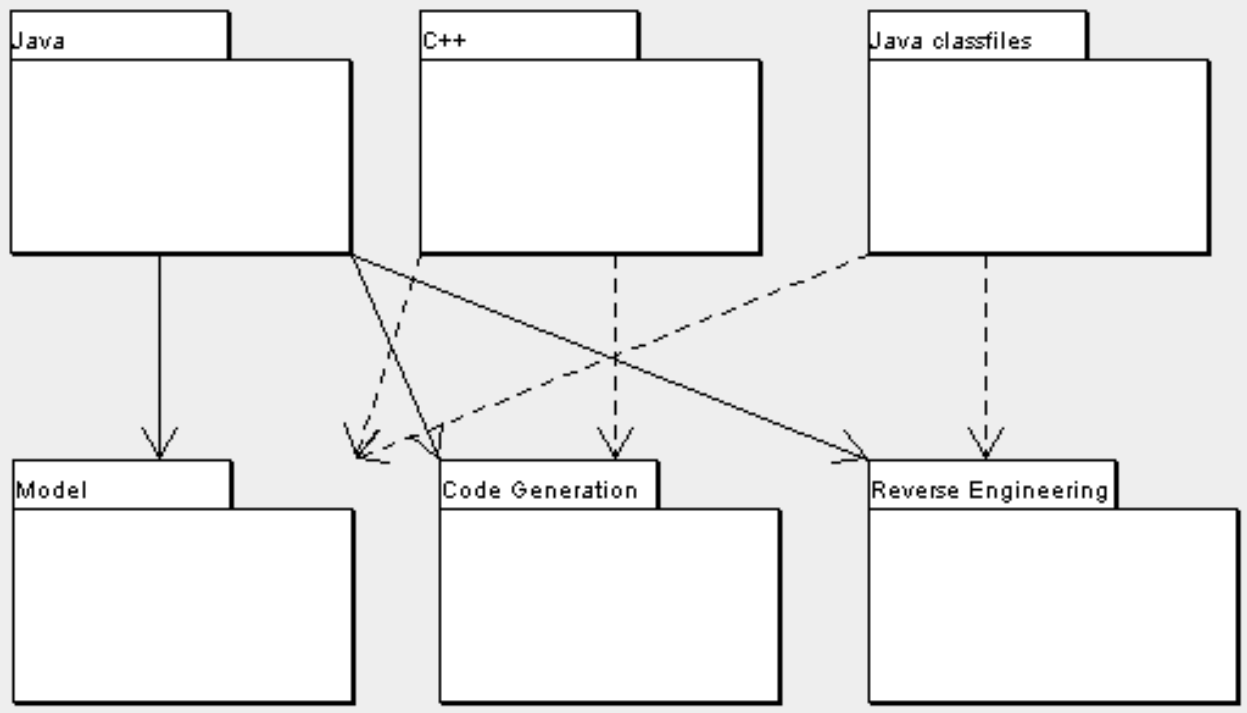
(here are the links, but I can hardly paint them as ASCIIs)

F E D

5.8. Other languages

Each other language supported by ArgoUML has its own component. 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.



5.9. The GUI Framework

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

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

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

The GUI Framework is a Layer 1 component. See Section 4.5, “Layer 1 - Description of components”.

This is implemented directly on top of Swing and Java2.

The GUI has (currently) the following main parts

- The menu
- The toolbar
- Explorer pane (Navigator pane)

  Upper left.

  Contains a tree of the model.
- Multi editor pane

  Upper right.

  Contains the diagrams (could eventually be something else).

- To do pane

  Lower left.

  To do items, different views.

- Details pane

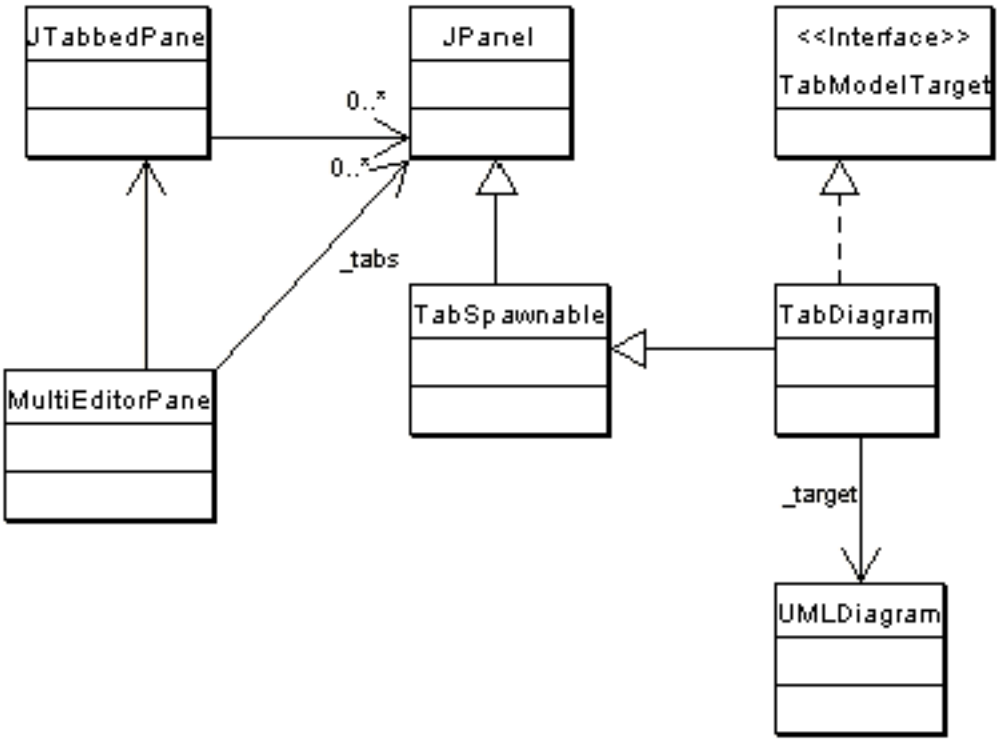
  Lower right.

  Contains a wizard from a To do item, a property panel of the current object, some other view of the current object.

5.9.1. Multi editor pane

The multieditorpane 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 multieditorpane consists of tabs that hold editors as you can see in the classdiagram.



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

The TabDiagram is spawnable. 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.

5.9.1.1. 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
  ```

5.9.2. 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 component) (See Section 5.2, “Critics and other cognitive tools”), Documentation, Style, Source, Constraints (an ocl view of the current object) (See Section 5.17, “OCL”), and Tagged values.

Warning

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

5.9.2.1. 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`.

5.10. 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 Layer 1 component. See Section 4.5, “Layer 1 - Description of components”.

Javahelp or some other help function will probably be used.

5.11. Internationalization

Purpose - to provide the infrastructure that the other components can use to translate strings; to provide the infrastructure that makes it possible to plug in new languages; to administer the default (English U.S.) language; to administer all supported languages.

The Internationalization is located in `org.argouml.i18n`.

The Internationalization is a Layer 0 component. See Section 4.4, “Layer 0 - Description of components”.

The internationalization is currently changing from ListResourceBundles to Property files. This chapter is not updated to fit the change. Please read carefully.

In ArgoUML internationalization (sometimes called i18n) is done using the ListResourceBundle-classes and parts of it is handled by the GEF infrastructure.

There are several sets of Bundle files for different domains within ArgoUML. Each domain has a name and is handled by a file. This is set up in `org.argouml.application.Main`.

5.11.1. 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 consistant 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 with ArgoUML. Each language teams are 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. All localized strings and resources shall be tranlated into the language.

   This is a constant work with keeping up with the changes that will be made to the ArgoUML code since ArgoUML is under fast development.

2. The terminology used shall be correct.

   This requires work in keeping up with the current literature in the domain of ArgoUML.

3. 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 we still have a big backlog of stuff in the GUI that is not localizable just by modifying the resource bundles. Each such thing is a Defect and shall be corrected.

4. 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 underlaying 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.

5.11.2. Ambitions for localization

Let me (Linus Tolke, May 2002) try to define the levels of ambition for us to try to make it possible to discuss where we are going.

1. No translation

   This is the lowest level of ambition that is a "do nothing"-level. This goes for all languages where we have not done anything like Swahili, Polish, South African English, ...

2. Tool translation

   This is the basic level of ambition that each Language Team should aim for. It means that in ArgoUML all strings are localized so that ArgoUML is giving a complete appearance of being a tool for that language.

   Setting this level of ambition for a language (or creating a team for the language) is pointless if there is no window system available for the language in questions. I mean, if neither the people working with Windows, Linux (KDE or Gnome) or java has collected enough interest to do a translation of the basic infrastructure there is no point in doing so for ArgoUML. (My Windows 2k has 80 supported languages so I would think that this is a no-issue.)

3. User environment translation

   This is the next level of ambition that can be set out by a Language Team that works really well and has plenty of translation resources left.

   It means that not only the ArgoUML tool should be translated but also everything around it that the user sees i.e. the User Manual, the Quick Guide, the FAQ, the Users' part of the ArgoUML Web site.

   Setting this level of ambition for a language is pointless if the problem domain does not exist in that language. I mean, if the professionals that use UML or other Software Engineering tools, in their every day work don't use their native language to discuss UML concepts, then there is no use in translating these concepts to their language, they will not use the translation because they are more comfortable with the English concepts. Note that the UML Specification does only exist in English and a natural part of this level of ambition would probably be to translate that.

4. Development environment translation

   Here I mean that everything that the developer of ArgoUML sees shall be translated.

   This begins with this Cookbook, then the Developers' part of the ArgoUML web site and also includes the javadoc comments in the code of ArgoUML and design documentation of included packages such as GEF, NSUML...

   We don't do this in the ArgoUML project.

5.11.3. How do I ...?

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

  Run checkstyle. Search for comments on keys.

- ...start a new Language Team?

  The Language Teams are loosely defined by the web page of language teams on the Tigris site. As soon as the language code and names (at least one) are in place the team is created.

  From that point it is the Language Teams responsibility to do a good job.

- ...find the languages internationalization code for the language you will add: en, es, en_GB,...

  The one you are currently using is shown in the log when ArgoUML starts. Search for lines looking like: `Language: sh Country: unknown`

- ...start the work?

  Look at the files in `org/argouml/i18n`, under `argouml/src_new`.

  Translate all the texts in each of these classes of files.

  This is a lot of extremely qualified work including searching well-known litterature 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/src_new/org/argouml/i18n`. They will have the names: `UMLResourceBundle_`*`language code`*`.java`, `UMLCognitiveResourceBundle_`*`language code`*`.java`, `ActionResourceBundle_`*`language code`*`.java`, `SettingsResourceBundle_`*`language code`*`.java`, `MenuResourceBundle_`*`language code`*`.java`, `DiagramResourceBundle_`*`language code`*`.java`, `TreeResourceBundle_`*`language code`*`.java`, and `NotationResourceBundle_`*`language code`*`.java`.

  Add the language to the JUnit list of tested languages and run the JUnit tests.

  The purpose of this is for you to get the simple JUnit tests to work for your language also.

  This is in the file `argouml/modules/junit/src/org/argouml/util/CheckResourceBundle.java`. Search for the supportedLanguages-array.

  Now you have completed the first iteration of the Tool translation ambition. The work will probably be more maintenance-like from here on.

- ...join an existing Language Team

  Discuss with the Language Team on where the team is in its work and what you can do.

- ...add or modify code with localized things?

  1. Write your code using the same way of handling strings as the surrounding code.

     This means that strings are denoted by "labels" or "tags" and then the resolution of the "tag" is in a `*Domain*ResourceBundle.java`-file.

     The name for the domain is most often specified in a variable like

     ```
     protected static final String BUNDLE = Domain;
     ```

or

```
    protected static final String RESOURCE_BUNDLE = Domain;
```

in a base class to what you are doing and there is a convenience method in the class `org.argouml.application.api.Argo` so normal strings are written

```
import org.argouml.application.api.*;
...
    String localized = Argo.localize(BUNDLE, tag);
```

2. Add your "tag" and resolution in English in the non-localized *Domain*`ResourceBundle.java`-file.

   How do I choose the tag? Jean-Hugues de Raigniac has made a small investigation as to how this is done in the java world and found that there is no real consensus on how to do this. He suggests a hierarchical choice of tags like this:

```
{"docpane.label.since", "Since"},
{"docpane.label.deprecated", "Deprecated"},
{"docpane.label.see", "See"},

{"stylepane.label.bounds", "Bounds"},
{"stylepane.label.fill", "Fill"},
{"stylepane.label.no-fill", "No Fill"},
```

3. Contact all the language-teams so that they can update their files.

Notice that if you somewhere change the meaning of a specific localized thing it would be a good idea to use a new "tag" for the new meaning. This will make it easier for the translation team to spot the modification.

There eledgedly 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.

5.12. 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 component.

Logging is currently implemented using log4j.

ArgoUML uses the standard log4j [http://jakarta.apache.org/log4j/] logging facility. The following sections 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.

5.12.1. 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, components, 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 all DEBUG and INFO log4j messages (see Example 5.2). The purpose of this test is to avoid the creation of the argument.

5.12.2. 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.
2.
3.

Example 5.1. For log4j version 1.2.x

```
import org.apache.log4j.Logger;
...
public class theClass {
...
    private static Logger _cat =
        Logger.getLogger(theClass.class.getName());

...

    public void anExample() {
        _cat.debug("This is a debug message.");
        _cat.info("This is a info message.");
        _cat.warn("This is a warning.");
        _cat.error("This is an error.");
        _cat.fatal("This is fatal. The program stops now working...");
    }
```

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 5.2. Improving on speed/performance

```
if (_cat.isDebugEnabled()) {
    _cat.debug("Entry number: " + i + " is " + entry[i]);
}
if (_cat.isInfoEnabled()) {
    _cat.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/].

5.12.2.1. 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 real 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;
...
        _cat.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());
        _cat.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.

5.12.3. 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 5.3. Various URLs

```
org/argouml/resource/filename.lcf                    ❶

http://localhost/shared/argouml/filename.lcf         ❷

file://home/username/filename.lcf                    ❸
```

❶ *name.lcf*
❷ *name.lcf*
❸ *name.lcf*

5.12.3.1. ...when running ArgoUML from the command line

There are currently two possibilities of running ArgoUML from the command line:

1. `argouml.jar`

2.
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 5.4. Command Line for argouml.jar

```
[localhost:~] billy% java -Dlog4j.configuration=URL -jar argouml.jar
```

Example 5.5. 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>
```

5.12.3.2. ...when running ArgoUML from WebStart

To view the console output, the WebStart user has to set *`Enable Java Console`* in the Java WebStart 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.

5.12.3.3. ...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 5.6. External Execution Property (Arguments)

```
-cp {filesystems}{:}{classpath}{:}{library} -Dlog4j.configuration=URL
    {classname} {arguments}
```

5.12.4. How to Customize Logging...

There are some sample configuration files provided in `org.argouml.resource`. Modify these according to your needs. Or alternatively, you can try configLog4j [http://www.japhy.de/configLog4j] to assist yourself in creating a log4j configuration file.

5.12.5. References

- http://jakarta.apache.org/log4j [http://jakarta.apache.org/log4j/]
- http://www.japhy.de/configLog4j [http://www.japhy.de/configLog4j/]

5.13. JRE with utils

Purpose - to provide the infrastructure to run everything.

The JRE is a Layer 0 component. See Section 4.5, “Layer 1 - Description of components”. 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.

5.14. 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 Layer 1 component. See Section 4.5, “Layer 1 - Description of components”.

5.15. Explorer

Purpose - to provide tree views of the model elements, diagrams and other objects.

The Explorer will be located in `org.argouml.ui.navigator ??`.

The Explorer is a Layer 2 component. See Section 4.6, “Layer 2 - Description of components”.

Several tree views are provided by means of Perspectives. Objects showin in the trees are the model elements, diagrams and other objects (such as profiles, groupings of model elements etc.).

The Explorer is currently shown in the Explorer Pane - the upper left hand pane of ArgoUML. (See Section 5.9, “The GUI Framework”.)

There is also a Explorer Configurator dialog, which allows the user to tailor the existing perspectives and create new perspectives to their needs (although the settings are not persisted).

5.15.1. Details of current implementation

Some of the classes are actually reused by the other tree view, i.e. the todo/critics list.

The Explorer provides some 'history' capability (although this is disabled presently and should problably be factored out) to navigate back to previously selected model elements.

Currently the Explorer Pane is mixed with the GUI framework and actions in `org.argouml.ui`. The Explorer is also dependant on "go rules", which are rules that help identify children nodes for and particular parent. "Go rules" are mixed with Diagrams and Property panels under `org.argouml.uml.diagram.ui` and `org.argouml.uml.diagram.`*`diagram`*`.ui`.

Explorer was previously called Navigator Tree or Navigator Pane.

5.15.2. Requirements

The Explorer must react to user and application events.

User events include

1. selection of a node, which must notify the other views to make the same selection.
2. right click on a node, which brings up a popup menu.
3. selection of another perspective in the Combox box, which must change the tree model to that perspective.
4. node expansion and collapse.

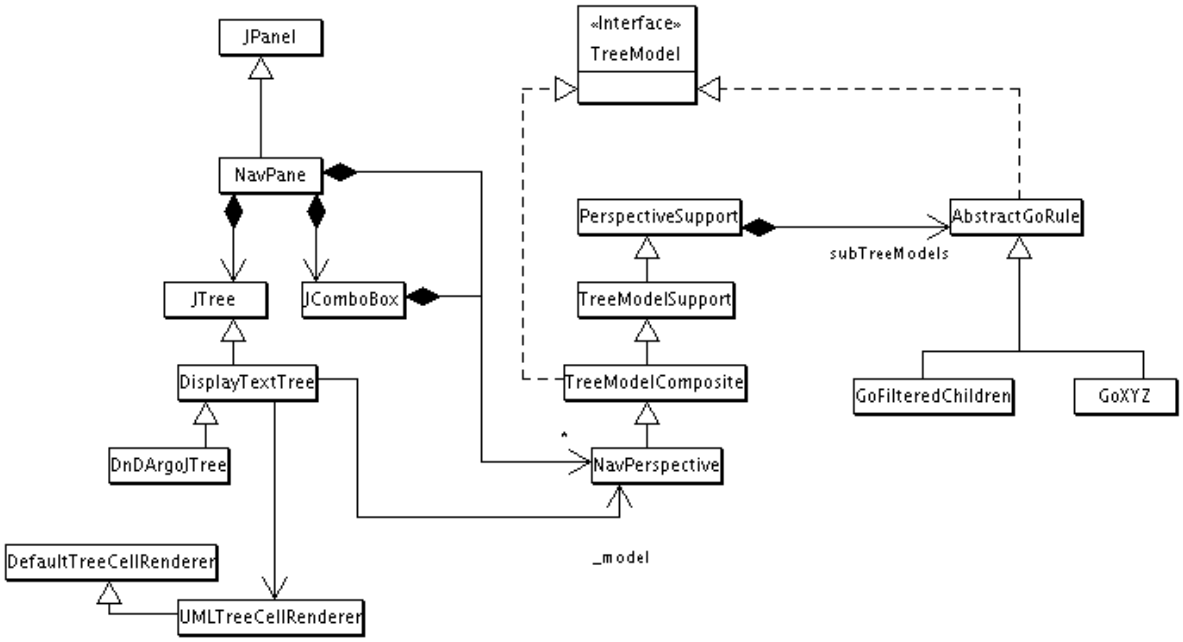
Application Events include:

1. change of project, the tree must update
2. change in selection in another view, any relevant rows to be highlighted.
3. model changed, the tree must update to reflect additions/deletions and name changes in the model.

5.15.3. Key Classes

The Explorer contains Swing components, and there is currently no abstraction layer (to help implement a IDE plugin for example.

There is a `JTree` subclass, `DisplayTextTree`, for the tree, with a `TreeModel`, Perspective. The Perspective is made of several 'Go rules', this enables Argo to convert the graph-like structure of a uml model into a tree and it is easily extensible.



There are several different ways of implementing a `TreeModel` in Swing. ArgoUML already has a model (the NSUML MModel instance), so we create a class (`NavPerspective`) that encapsulates the MModel instance and implements `TreeModel` (contains methods that enable the JTree to calculate the tree structure). This implementation does not build/load a tree model on instantiation, rather it provides children nodes for any given parent node when the user requests a node expansion. The `NavPerspective` delegates the task of calculating child nodes to the Go rules, making the tree model very flexible. However, the price for this (at the moment) is slow node expansion times for large models.

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 `DisplayTextTree`.

Event handling is done in `DisplayTextTree` and `NavigatorPane`.

5.15.4. How do I ...?

- ...add another perspective?

  - at runtime perspectives can be changed using the `NavigatorConfigDialog`
  - hard code it

    Perspectives are build in the `NavigatorPane`, so you can add your one there. Also create new Go rules that are subclasses of `AbstractGoRule`, then register them with the Perspetive instance using `addSubTreeModel(TreeModel)`.

- ...improve the PopUp menu?

  Look in the `NavigatorPane` class. The menu is built at runtime in reaction to a mouse event.

5.16. Module loader

Purpose - to provide the mechanisms to load (and unload) the Layer 3 and auxiliary modules.

The Module loader will be located in `org.argouml.?`.

The Module loader is a Layer 2 component. See Section 4.6, “Layer 2 - Description of components”.

Currently the module loader is located in `org.argouml.application.modules.ModuleLoader` with interfaces in `org.argouml.application.api`.

This handles the enabling and disabling of every module.

An idea on how it could work: It is then the modules responsibility to connect and register to the component or components it is going to work with using that components Facade or Plugin interface.

For details on how to build a module see Section 6.2, “Modules and PlugIns”.

5.16.1. What the ModuleLoader does

The ModuleLoader is looking for module jars. It actually scans through all jars available in the argo ext dir 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 classes to be loaded. Also notice that the Specification-Title and Vendor must be specified correctly for this to work.

5.17. 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 component. See Section 4.7, “Layer 3 - Description of components”.

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. Historicaly 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 component 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 is not yet updated to discuss layers.

This section explains some general concepts which come in handy, when programming in ArgoUML.

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

This section is not yet updated to discuss layers.

6.2.1. Differences between modules and plugins

The ArgoUML tool provides a basis for UML design and potentially an executable architecture environment for other applications. This is solved by clear interfaces between the ArgoUML core and the extensions. Extensions are called modules and the classes within the modules that attach to ArgoUML core are called plugins.

- 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

  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

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

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. Using Modules

When modules are used they can't be distinguished from the rest of the ArgoUML environment.

6.2.2.4. How do I ...?

- ...create a module?

- ...tell when a module is loaded?

6.2.3. Plugins

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”), 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)

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 in the java code

This section is not yet updated to discuss layers.

The previous section describes how modules and plugins are connected on the java level totally independant 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 plugins 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. How do I ...?

- ...create a new source-code module.

  Suggestion, copy from the `junit` module as described here.

  Make a copy of `argouml/modules/junit` into `argouml/modules/`*`yourname`*.

  Remove `junit.jar` from `argouml/modules/`*`yourname`*`/lib`.

  Add any jar you need to `argouml/modules/`*`yourname`*`/lib`.

  Edit `argouml/modules/`*`yourname`*`/module.properties`

  Edit references to `junit.jar` in `argouml/modules/`*`yourname`*`/build.xml` to any new jars you need.

  Edit `argouml/modules/yourname/src/org/manifest.mf`.

  Reorganize the source files as necessary. Something like org.argouml.*`yourname`* as the package root.

- ...get Argo to use a plugin?

  Once you've created a jar file with a plugin 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 plugin. This will copy the jar file to the proper directory in the main ArgoUML build target. You can test your plugin by running **build run** in the `src_new` directory.

  If you need to install the jar "the hard way", try the following steps.

  -
Edit->SettingsEnvironment`${argo.ext.dir}`

  -
  -

Chapter 7. Organization of ArgoUML documentation

Linus Tolke

This chapter contains written down ideas on what goes into what 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 persistant value.

1. The code, variablenames, class names
2. The javadoc
3. The cookbook
4. The web site in CVS
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 7.1. Bits of documentation

Bit	Audience	Main purpose	Contains
The 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 9, *Standards for coding in ArgoUML* for more information.
The 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.
The 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 set up. Be a store of the agreed solution	Instructions on how to add new functions and behavior. Instructions on how to do the chores around maintenance (build a release, publish a release, build the docu-

Bit	Audience	Main purpose	Contains
		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.	mentation 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.
The web site in CVS	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.
The 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 texts within the running 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.
The 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.

Chapter 8. CVS in the ArgoUML project

8.1. How to work against the CVS repository

The CVS 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 dependancy 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 dependancy 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 **cvs update** in src_new to make sure that you do not forget to commit any file and to make sure that no one else has commited anything in the mean time.

   Remember that if you do not commit all the files from src_new that **cvs update** 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 (**cvs update** shown U, or no longer pertinent) 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 inconsistant 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 main branch in cvs, create a branch to work in. Then when your work is complete, you merge the branch into the main branch.

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.

8.2. Creating and using branches

We use the following standards in ArgoUML:

- Released versions get the tag *VERSION_X_X_X*
- 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
  - work is a literal
  - *`explanation`* is something like javahelp, propertypanel, cppcodegeneration
  - *`owner`* is a self explaining code for the owner of the branch, e.g. tlach (Thierry Lach) or mklink (Markus Klink).

Merging branches together is causing some work. So please use them sparingly and announce your intentions before on the mailing list.

8.2.1. How do I ...?

- ...commit stuff?

  You have made, the change, tested it and are satisfied with it.

  Do a **cvs update -d** and see that only the files you have changed are marked as modified. If files are updated or patched by this command, please recompile and test again.

  Do a **cvs diff** on each of the files and verify that only the lines you have changed are modified.

  Do a single **cvs commit** for all the files included in the change. This reduces the risk that someone else updates in the middle of your work and also reduces the amount of notifications of commits sent out. Include changes to documentation and JUnit tests if applicable.

  Don't forget to update the corresponding issue (if any) in Issuezilla i.e. set it to RESOLVED/FIXED.

- ...get my update or patch into CVS if I don't have CVS write rights?

  Contact any of the active developers on the list and send them your updates. They're very nice about it the first few times.

Supposing that you have checked out CVS as guest, then after you have mailed a diff or file to an active developer, and he has entered it in CVS your checked out copy contains the change but is not in sync and the next **cvs update** will result in an merge error. The simplest way to solve this is to do remove all files modified by you before doing the **cvs update**. The **cvs update** will restore all the files from the CVS repository and you can start with the next update.

- ...get a list of the currently active working branches?

  You can't from CVS. You need to follow the announcements of created and discontinued branches on the mailing list to know what branches are interesting.

- ...create a branch for my work on *xxxyyy* and start work on that branch?

  This assumes that you have a checked out copy of argouml

  1. Change directory to the directory where argouml is checked out.
  2. Enter the argouml directory: **cd argouml** or **chdir argouml**
  3. Create your branch: **cvs tag -b *work_xxxyyy_myname***

     myname is is a self explaining code for you (your Tigris login).
  4. Change your checked out copy to be on the branch: **cvs update -r *work_xxxyyy_myname***
  5. Do your work!
  6. Check in your changes in the branch: **cvs commit -m'*Blablabla*' [file]**
  7. Continue working and checking in!

- ...move my work from my working branch into the release?

  This is done when your work with the feature xxxyyy is finished and you have decided/received clearance to enter it in the main branch.

  1. Change directory to the directory where argouml is checked out.

     If you are just working on one feature at a time this is the place where you have a checked out copy on the branch in question. If not, this could be any checked out copy of the source that does not contain any uncommited changes.
  2. Enter the argouml directory: **cd argouml** or **chdir argouml**
  3. Move the checked out copy that you are working on to the main branch: **cvs update -A**
  4. Merge the changes from the branch into your checked out copy: **cvs update -j *work_xxxyyy_myname***
  5. Compile and run all your tests again.

     This is to verify that the merge was all right, no one else had done any changes that in the meantime that has in any way modified the work made in the branch.
  6. Commit your changes in the main branch: **cvs commit -m'*xxxyyy entered in the main branch***

7. Discontinue your branch!

   From this point on it is important that you do not reuse your branch for any work. Only check it out for the purpose of examining how things were in the branch. Make sure that all other developers that have been looking at your branch also knows that it is discontinued.

- ...look at someone else's work in a branch?

  You need the name of the branch, i.e. the *work_xxxyyy_hisname*.

  There are two alternatives:

  - Check out argouml or part of it on that branch: **cvs co -r *work_xxxyyy_hisname* argouml**
  - Update your copy of argouml to be on that branch: **cvs update -r *work_xxxyyy_hisname***

    Make sure that your copy does not have any uncommited code or else your uncommited code will be present in your checked out copy on the branch. This could, on the other hand, be useful if you want to test if your uncommited code works also with the additions on that branch.

8.3. Other CVS comments

This is included in the cookbook because it seems that there are persons within the project that don't have the in-depth knowledge of CVS nor the interest or need to acquire it. For that reason some simple questions are answered here for use of CVS in the project.

- Why do I get double lines? Why do I get ^M at the end of each line? Why do I get the whole checked out file on a single line?

  CVS is line oriented. It stores in the repository the concept of a new line after each line. It is the CVS clients (the program you have installed on you machine) responsibility to convert the conceptual new line to the correct new line character on your system.

  ### Note

  This is only so for normal files (not marked with -kb in cvs).

  If files are moved from one system to another or for that matter checked out on one system and used and edited on another (NFS, SMB, ...) this is not done correctly. There could also be CVS clients out there, not doing this correctly.

  Systems known to the author (Linus Tolke) are Unix uses LF, DOS/Windows uses CR-LF, Mac uses CR.

  Most of the time this really doesn't matter because the editors and java compiler on all systems are very forgiving.

  There are however some cases when this is cumbersome.

  1. When an editor (or developer) decides to "fix-it".

This means that the editor (or the developer) goes through the file and removes ^M on every line or something else that touches every line in the file.

This is a problem because the subsequent commit will also touch every line in the file making that file unmergeable. This means that every developer that had it modified in a branch or in a checked out copy will have no help from cvs when doing his merging.

Remember that you never know what other developers are working with.

This is fixed by not doing any such fixes and doing a **cvs diff** before each check in so that your editor has not done this for you.

2. When cvs clients and file systems are not in sync

   This could result in one of several things. Either each line gets an extra empty line when commited, or the whole file turns out to be on the same line.

   This is the case on several files in the repository at the moment (August 2002, Linus) and can be cumbersome for the developers.

   These cases should be fixed because the files are no longer readable. For the first case, removing every other line (the empty ones) can in some cases be done without cvs having problems with merging later on. For the second case, with a single long line, this will be very problematic so even though it might cause problems for other developers it is better to do this as soon as possible.

   When this is fixed, let the fix be the only thing done in that commit.

   To avoid this in the future, always do a **cvs diff** before doing your change to make sure that only the lines that you have actually modified will be changed by cvs and not the whole file.

Files that are binary that shall be stored in cvs shall be marked as binary. They are marked with the admin flag -kb. This means that the line ending conversion mechanism will not be applied on those files and they will be exactly the same on all systems. This is good for jars, gifs, and other such files.

8.4. CVS repository contents

This chapter describes what parts of the CVS 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.

This chapter is organized as the CVS repository itself and everything is in alphabetical order.

- `build`

  Directory where the built things end up.

  There is actually no real need to keep this in CVS. It is there just as a place holder.

- `conf`

  Not used. Empty.

- `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.

- `extra`

  Not used. Empty.

- `lib`

  A set of jar files.

  This directory contains the jar files of products used by the ArgoUML (such as log4j, nsuml).

  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 versioning informations, the same way as shared libraries in Unix world: foo-x.y.z.jar, bar-x.y.z.jar, etc...

- `modules`

  Contains source level modules of ArgoUML.

  Source level modules are modules that can be compiled and deployed independantly (after) the rest of ArgoUML. Each module is located in its own subdirectory. This is the list as it looks now (March 2003).

    - `jscheme`

      Module that allows to extend ArgoUML using scheme.

    - `junit`

Old directory with JUnit tests. These should be migrated to and all new JUnit tests should be created in the directory `tests`.

- `menutest`

  Test module that tests the plugin interface for the menus.

- `php`

  Language generating, Notation and reverse engineering for PHP.

- `cpp`

  Code generation for C++.

- `csharp`

  Code generation for C#.

- `src`

  Source code.

  This will contain one directory for each component within ArgoUML. They will all compile and be tested with controlled dependencies to other components.

- `src_new`

  All source code for ArgoUML including pictures of icons.

- `tests`

  All source code for JUnit tests of everything that is in the src_new directory. See Section 2.4, “The JUnit test cases”.

- `tools`

  All 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.)

- `www`

  This is all the static contents of the web site. See Section 2.3.2.1, “How the ArgoUML web site works”.

Chapter 9. Standards for coding in ArgoUML

The coding style for ArgoUML is the following

- Each file starts with some header info: file, version info, copyright notice, classes in this file (if more than one), original author (if you want). Like this:

```
// $Id$
// Copyright (c) 2003 The Regents of the University of California. All
// ...

// Classes: blabla, blabla (all classes of interest in this file)
// Original Author: who ever
```

- All instance variables are private and their names begin with an underscore. If the variable should be accessible then add public or protected accessor methods with the same name as the variable without the underscore with "get" or "set" prepended. For example: `_lineWidth`, `setLineWidth()`, and `getLineWidth()`.
- In general, write short code. If a method will fit comfortably on one line, then put it on one line.
- Use javadoc for each class, instance variable, and method. 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.
- Indicate places of future modifications with

```
// TODO: reason
```

- Name all classes with an initial uppercase letter, and all variables and methods with a lowercase one. I use the `allTogetherWithCaps` naming style. Name static variables with an underscore and an inital capital letter, e.g., `_PossibleLanguages`. Name constants with all upper case and underscores, e.g., `GRIP_MARGIN`.
- To emphasize clusters of classes we are using what we call the binomial naming style (I am sure others have thought of this also): The root class of the cluster has a short name (e.g., Layer), other members of the cluster use that name as a prefix (e.g., LayerGrid). This makes many of the class name longer than they might be normally (e.g., Grid would be shorter). But this provides a lot of context without having to look at a class inheritance diagram. It is also very nice when you have to look at an alphabetical list of classes. I try to name class clusters so that they are not lexigraphically close others (e.g., the Net cluster used to be named Model, but that lexigraphically overlapped the Mode cluster).
- 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.
- If possible use lines shorter than 80 charaters wide.

  This is exactly as it is stated in the Sun Code Conventions. It is here just for the emphasis.
- 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.

- For everything else follow Code Conventions for the Java Programming Language [http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html]!

9.1. Settings for Eclipse

These style guides correspond to the following settings in Eclipse:

- In Preferences => Java => Code Formatter => New Lines

  None of the boxes "Insert a new line before opening brace", "Insert new lines in control statements", "Clear all blank lines", "Insert new line between 'else if'", or "Insert a new line inside an empty block" are checked.

- In Preferences => Java => Code Formatter => Line Splitting

  Maximum line length is 80.

- In Preferences => Java => Code Formatter => Style

  None of the boxes "Compact assignment" or "Indentation is represented by a tab" are checked.

  Number of spaces representing a tab: 4. This should probably be read as Number of spaces representing a level of indentation.

- In Preferences => Java => Java Editor => Appearance

  Displayed tab width: 8

  "Insert space for tabs (see Formatting preferences)" checked. There seems to be no way of having tabs set at width 8 and the indentation level set at 4 at the same time so we must let Eclipse generate code without tabs to obey the Sun Coding standard.

9.2. Settings for NetBeans

These 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).

9.3. Settings for Emacs

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)))
```

Chapter 10. Further Reading

10.1. 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.

10.1.1. 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.

10.1.2. Where to find it

LINK: Robbins Dissertation [http://argouml.tigris.org/docs/robbins_dissertation/]

10.2. Martin Skinners Dissertation

Enhancing an UML Modelling Tool with Context-Based Constraints for Components

10.2.1. Abstract

Noch vor der Erstellung eines detaillierten Entwurfs hilft ein Spezifikationsmodell eines komponenten-basierten 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.

10.2.2. Where to find it

LINK: Martin Skinners dissertation [http://www.cocons.org/publications/CCL_plugin_for_ArgoUML.pdf]

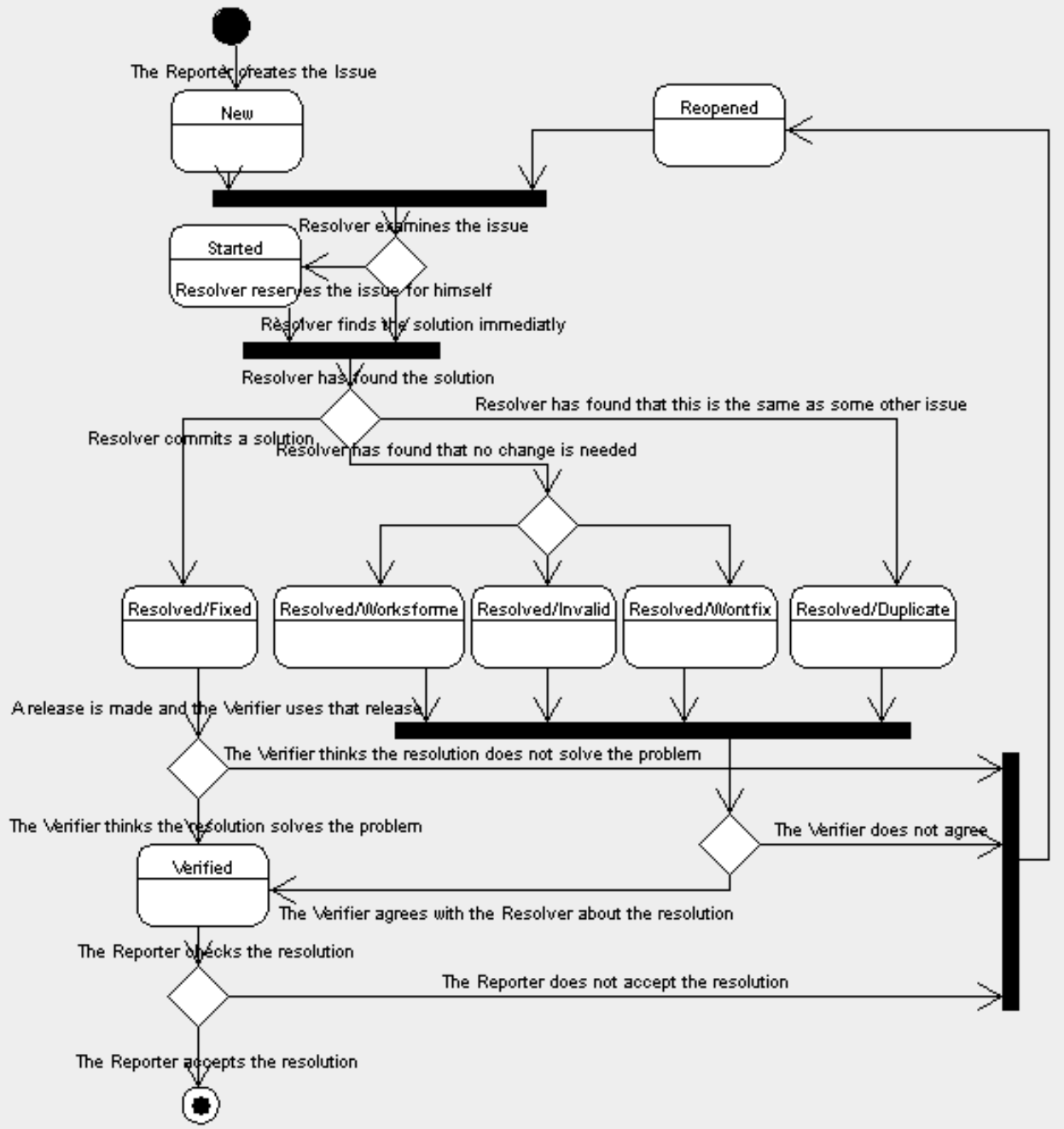
Chapter 11. Processes for the ArgoUML project

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].

11.1. The big picture for Issues

Here is the big picture of the life of an Issue.



11.2. 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/project/www/docs/issue_lifecycle.html] and for that reason it is not a complete list.

11.2.1. Priorities

The priorities are used in the following manner in ArgoUML:

- P1 - Fatal error

  ArgoUML cannot start. Crashes program, jvm or computer.

- P2 - Serious error

  Information lost.

- P3 - Not so serious error

  Functions not working. Strange behavior. Exceptions logged.

- P4 - Confusing behavior

  Incorrect help texts and documentation. Inconsistant behavior. UI not updated. Incorrect javadoc.

- P5 - Small problems

  Spelling errors. Ugly icons. Excessive logging. Missing javadoc.

11.2.2. Resolutions

- LATER and REMIND

  Not used.

- 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.

11.3. Roles Of The Workers

The roles described below are per issue, i.e. for every issue, there is at least a reporter, a resolver and a verifier. 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.

11.3.1. 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]. For entering new issues, registering (as described in 1.3) is not

required.

- 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 verified state only. 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 verified state only. 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.

11.3.2. 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 CVS (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 CVS.

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.

11.3.3. 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
- Mark the issue as "verified"

  If the Verifier can conclude that the problem does not exist or the feature/enhancement is now present the issue is marked as verified.
- 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.

11.4. 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 artefact. 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 is-
   sue_status=REOPENE
   [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=NEW&D].
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 sofar 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".

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.

11.5. 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.

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 resolu-

   re
   so
   lu
   ti
   o
   n
   =
   W
   O
   R
   K
   S

FORME].

2. Run the specified release of ArgoUML as 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.
   - 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.

5. If you during this, have discovered other problems than the ones that are stated in the Issue, create new Issues stating 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.

11.6. 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.

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 resolu-

&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.

11.7. How to Close an Issue

This is performed by the person that originally raised the Issue or by the QA responsible for that area. 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 Issuesilla.

1. Pick any Issue that is Verified and that you have raised or that you have found and refrained from raising because somebody else already had written it. The list of all VERIFIED issues [http://argouml.tigris.org/issues/buglist.cgi?component=argouml&issue_status=VERIFIED].

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, put the Issue in Status CLOSED.
   - If you are not satisfied but the problem is solved as it is written in the Issue, put the Issue in Status CLOSED 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.

11.8. How to relate issues to problems in subproducts

ArgoUML uses some product internally and is to some respect very dependant on the well functioning of these products. This are products like GEF, NS-UML, ocl, log4j, xerces, jre, ...

Occasionally a bug found in ArgoUML is found to be a problem in one of these subproducts and cannot or is extremely complicated to fix within ArgoUML.

If this happens this is the way to register that Issue in Issuezilla.

This can be performed by any member of the project (any role). There might be special skills involved depending on the nature of the Issue.

Do the following:

1. During your examination of an issue you find that it is a problem in one of the ArgoUML subproducts (GEF, NS-UML, ocl, jre, ...).

2. Write a comment in the issue stating which one of the subprojects that is the problem (and what the problem is).

3. Post a bug in that subproducts bug reporting tool (or find the bug already registered).

   I am assuming that there is such a tool for all the subproducts. If there isn't for the product in question, then explain the problem to the responsible person for this product so that we are sure that the problem is communicated.

4. Set the Issue in Issuezilla to RESOLVED/WONTFIX and enter the reference from the subproducts bug reporting tool and if possible the URL to the bug reporting tool or to the bug in question.

The person responsible for the sourceing of the subtool in question (Currently (December 2002) the subproduct responsible role is not explicitly pointed out but could be anyone that feels that something must be done.) does the following for each new release of a subproduct.

1. Looks at the new release of that subproduct to see if any of the outstanding issues against that subproduct are in fact fixed in the release.

2. If any of the issues are fixed, then he weights together the importance of the issues fixed, the amount of work needed to fit the new version of the subproduct instead of the old one, the planned releases of the subproduct with other issues promised to be fixed, and the current release plan of ArgoUML. From this he decides wether it is time to do the update of the subproduct within ArgoUML or to wait.

3. If he decides that it is time to update, he adds the new version of the subproduct, does all the needed work within ArgoUML, tests and commits everything, puts the issues indeed fixed in VERIFIED/WONTFIX, and also closes the bugs registered in the subproducts bug reporting tool.

Index

N

O

P

R

S

T

U

W

X