

AUTOFOCUS 2



- Developer Guide -

Yakov Benilov, Tobias Hain, Jan Romberg, Florian Hölzl

Lehrstuhl Professor Manfred Broy
Technische Universität München

hain@in.tum.de

May 5, 2006

Contents

| | | |
|----------|---|-----------|
| 1 | Installation | 2 |
| 1.1 | Requirements | 2 |
| 1.2 | Getting AUTOFOCUS 2 | 2 |
| 1.2.1 | Integration in Eclipse IDE | 3 |
| 1.2.2 | Customizing build path / library setup | 7 |
| 1.3 | Launching of AUTOFOCUS 2 | 8 |
| 1.4 | Directory structure | 8 |
| 1.5 | Compatibility | 9 |
| 1.6 | Tools FAQ | 9 |
| 1.6.1 | Ant | 9 |
| 1.6.2 | Eclipse | 9 |
| 1.6.3 | Together | 13 |
| 2 | Design | 13 |
| 2.1 | Plugin mechanism | 13 |
| 2.1.1 | Usage of Plugin mechanism | 14 |
| 2.2 | GUI components | 15 |
| 2.2.1 | Menu bar | 15 |
| 2.2.2 | Tool bar | 15 |
| 2.2.3 | Repository | 16 |
| 2.2.4 | Window Manager | 16 |
| 2.2.5 | Model browser with pop up menus | 16 |
| 2.2.6 | Logging mechanism with hyperlinks | 16 |
| 2.2.7 | shared GUI logic | 17 |
| 2.3 | Event propagation | 17 |
| 2.4 | View Component | 18 |
| 2.4.1 | SSDView | 18 |
| 2.5 | Editors | 19 |
| 2.5.1 | GEF | 19 |
| 2.5.2 | GEF usage concept | 22 |
| 2.5.3 | Porting Property Editors | 23 |
| 2.5.4 | GUI highlighting | 24 |
| 2.6 | Visitors for traversing trees of model objects | 24 |
| 2.7 | Management of a model's status using StatusManager | 27 |
| 2.8 | Example: Using Visitor to build a StatusChecker | 29 |
| 3 | Configuration | 29 |
| 3.1 | Justification | 32 |
| 3.2 | Description | 32 |
| 3.2.1 | Allow internal classes access to "system-wide" properties | 32 |

| | | |
|-------|--|----|
| 1 | INSTALLATION | 2 |
| 3.2.2 | Allow the user to change these properties via a GUI in the program | 33 |
| 3.2.3 | Allow the user to change these properties in a config file | 34 |
| 3.3 | Miscellaneous notes | 35 |
| 3.3.1 | SuperTableCellEditor, SuperTableCellRenderer | 35 |
| 3.3.2 | LabelledString | 35 |
| 3.3.3 | ComboBox, LabelledStringComboBox | 35 |
| A | GEF Discussions | 36 |
| A.1 | How to use Grouping | 36 |
| A.2 | Overriding paint() | 37 |

1 Installation

1.1 Requirements

In order to develop and run AUTOFOCUS 2 the following software is required:

- Java SDK (<http://java.sun.com>)
- Ant build tool (<http://ant.apache.org>)

These optional tools are useful:

- Jikes java compiler (<http://oss.software.ibm.com/developerworks/opensource/jikes/>)

Jikes has the same functionality as SUN's javac, but is much faster. If jikes is present on your system, adjust `af2.properties.global` to compile using jikes.

1.2 Getting AUTOFOCUS 2

All AUTOFOCUS 2 documentation and source code is controlled by the configuration management tool CVS. The following settings are required to check out AUTOFOCUS 2:

| | |
|------------------|--|
| CVS server: | <code>cvsbroy.informatik.tu-muenchen.de</code> |
| repository path: | <code>/home/proj/cvs/cvsroot</code> |
| module: | <code>af2</code> |
| connection type: | <code>ssh</code> |

A guide for setting up CVS can be found here: <http://www4.in.tum.de/~hain/tumguides/cvs-guide.html>

A guide for setting up SSH without having to enter passwords can be found here: <http://www4.in.tum.de/~hain/tumguides/ssh-guide.html>

1.2.1 Integration in Eclipse IDE

The next section describes how to set up an Eclipse (www.eclipse.org) project for AUTOFOCUS 2. Since Eclipse offers integrated CVS, SSH and ANT support no additional tools need to be installed.

Launch Eclipse and create a new project by selecting Menu File / New / Project. In the next dialog select Java : Java Project (see figure 1).

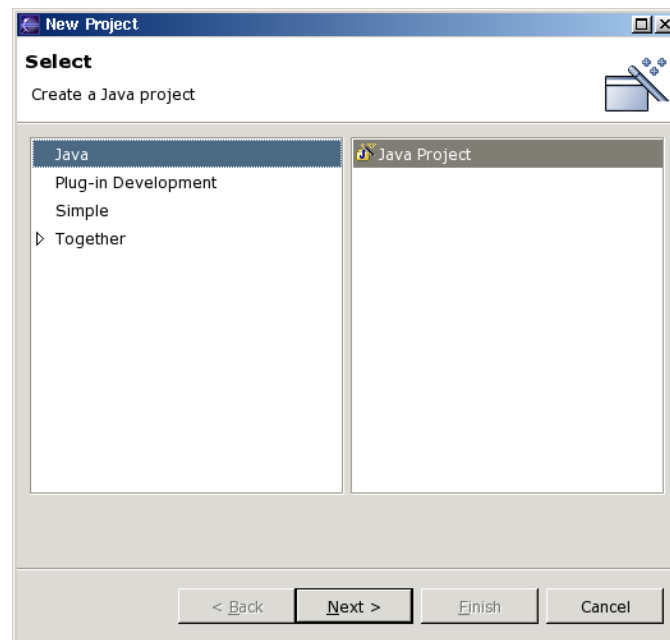


Figure 1: Eclipse - New Project Dialog

Name the project "af2", and click Finish (see figure 2).

The new project has been created and needs to be connected to the CVS. Right click on the project in the package explorer and select "Team / Share Project".

Fill in the two dialogs also depicted in figure 3 and figure 4.

Since only the sources are interesting af2/Software is checked out. Example models and documentation should be checked out outside of Eclipse.

You should be told module "af2/Software" already exists remotely. The question whether you like to synchronize with your local project should be answered with "Yes". The "Select Tag" dialog lets you specify the CVS branch to check out - select "HEAD".

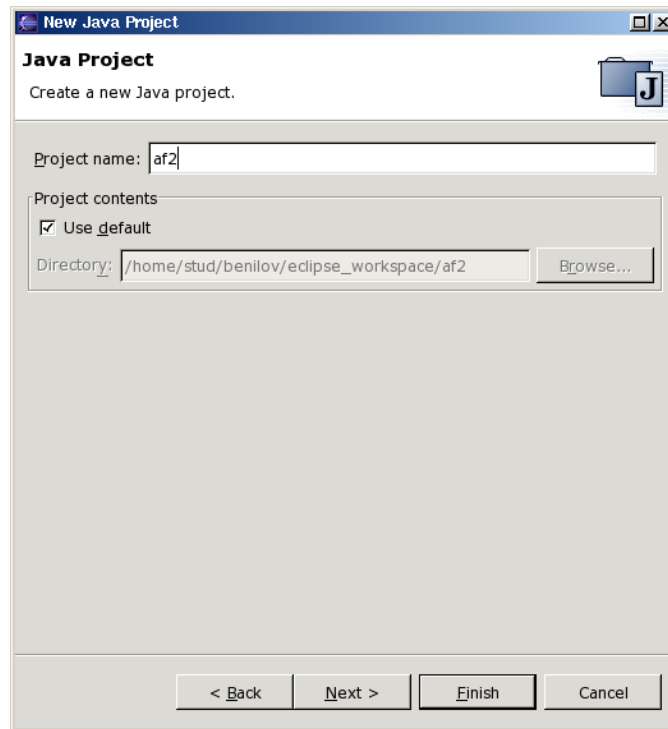


Figure 2: Eclipse - New Java Project Dialog

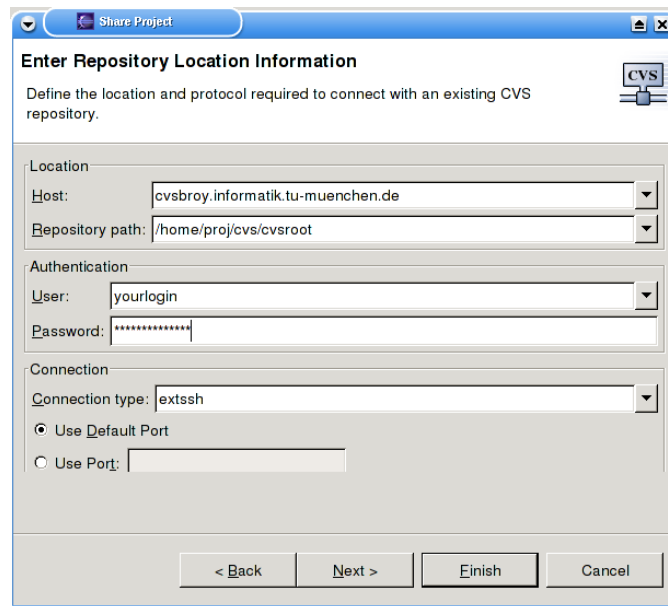


Figure 3: Eclipse CVS dialog

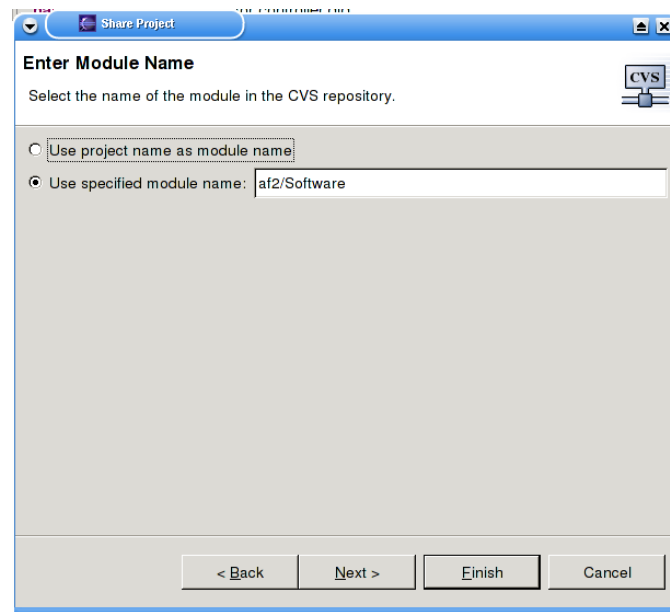


Figure 4: Eclipse CVS dialog

Note: If you hit an error while you are running through the above steps and no files from CVS are checked out, repeat the process once again starting with context menu on project: "Team / Share Project". This time choose the existing repository at the "Share Project with CVS Repository" dialog.

Now back at the main Eclipse desktop, you should see a Synchronize view (it can also be brought up by clicking the "Synchronize" tab at the bottom of the desktop). In this view, right-click on the project name "af2" (it is the root of the tree) and select "Update from Repository" in the appearing context menu. The source files are checked out from the CVS repository into the local repository.

Note: In Eclipse 3.1 the synchronisation is part of the share project wizard. Here, you can update directly by using the context menu option "Update" or you can finish the wizard and update from the tree view by using "Team / Update". Before hitting finish in this wizard step you should disable the "launch commit wizard" option, because you do not need to commit eclipse specific files like ".project".

Integrating Ant scripts On the main Eclipse desktop, select the menu *Window Show View Ant*. In the Ant View, click on the flashlight ("Add Buildfiles with Search"); leave all the default options on the next dialog and click "Search". After the search has completed, run the build script *AutoFocus2 > build* in order to build required libraries e. g. *validas.jar*.

After the build has finished (“BUILD SUCCESSFUL” should be displayed in the console output tab”), right-click on the project in the Package Explorer view and select “Refresh”. This step is necessary in order for Eclipse to pick up the new compiled libraries.

Setting Up Paths Select the menu *Project Properties*. Then select *Java Build Path Source tab Add Folder*. Add the following paths:

- af2/AutoFlex/src
- af2/Metamodel (excluded: Classes/; Interfaces/)
- af2/Misc
- af2/MSc
- af2/ODL

Unfold af2/Metamodel after you included it to the list of source files, select “Included: All” and click the “Edit”-button on the right hand side. Include only the directories beginning with af, quest and validas.

When this is done, make sure that in *Java Build Path Default output folder* says “af2/bin”. The dialog should look like figure 5.

Note:Eclipse 3.0 bug: Sometimes the selection tree of Java Build Path Source tab Add Folder is empty. Resize it, when it opened and the contents shows up.

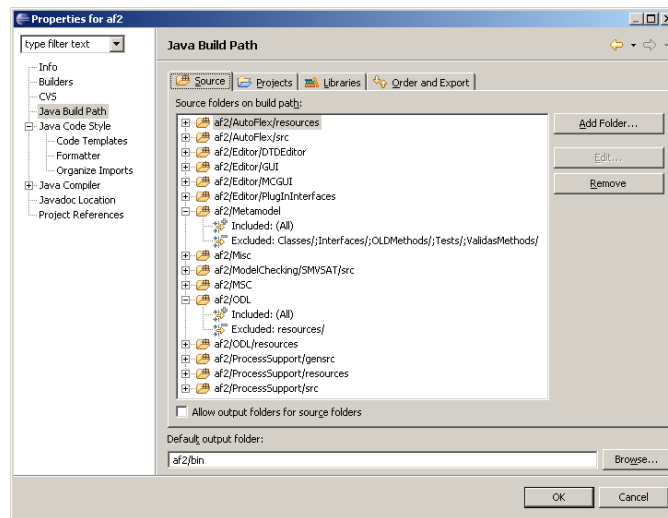


Figure 5: Eclipse Java Source Path

Now open the Libraries tab of *Java Build Path Add Jars...* Select all these jar files:

- all in *af2 > lib* except *odl.jar, misc.jar, validas.jar*
- all in *af2 > Autoflex > lib*
- Tools/JCoverage1.05/lib/log4j/1.2.8/log4j.jar

The dialog should look similar to figure 6.

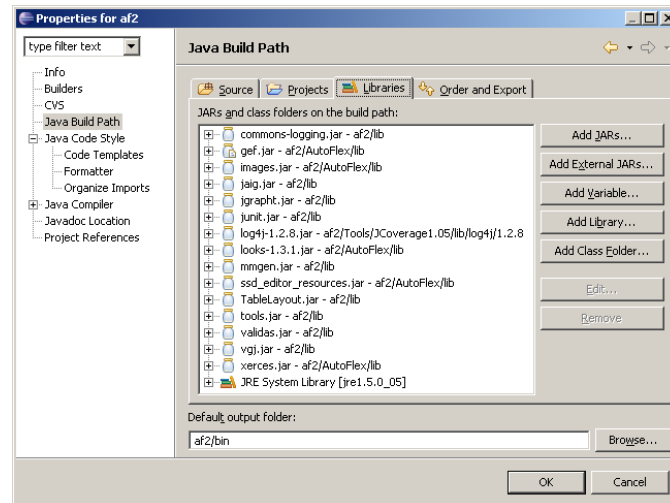


Figure 6: Eclipse Java Libraries

This Eclipse setup should not mark the source directories with errors. If this is the case either the current CVS version is broken (usually only a few files don't compile then). Or review the Eclipse setup if there are plenty of compile errors.

You may want to exclude `build.xml` files from source directories. Otherwise they are copied as resources to the build directory of Eclipse and e. g. show up twice in the Ant view. You may do this by adding file excluding filters in the project properties *Java Build Path Source Tab*.

1.2.2 Customizing build path / library setup

Normally you do not intend to compile all sources with Eclipse, but you want to concentrate on sources relevant for your developing. For example, there is no need to have ODL packages in your build path, if you do not intend to use it or develop in it.

You can replace the following source folders from your build path and instead include the given library or vice versa (In the table "af2/" is the project root).

| source folder | library |
|---|---|
| af2/Editor/DTDEditor | af2 > lib > plugins > DTDEditorPlugin.jar (*) |
| af2/Editor/GUI | af2 > lib > afgui.jar |
| af2/Editor/MCGUI | af2 > lib > mcgui.jar |
| af2/Metamodel | af2 > lib > validas.jar |
| af2/Misc | af2 > lib > misc.jar |
| af2/MMGen | af2 > lib > mmgen.jar |
| af2/Modelchecking/SMVSAT/src | af2 > lib > smvsat.jar |
| af2/ODL | af2 > lib > odl.jar |
| af2/plugins/OSGiBundle-Editor/{src, res} | af2 > lib > plugins > OSGiBundle-Editor.jar (*) |
| af2/ProcessSupport/{resources, gensrc, src} | af2 > lib > procsupp.jar |

You can also use this table to setup source associations within Eclipse. Thus you have source code available for debugging.

(*) add “-DPluginDir=./lib/plugins” to your VM arguments in eclipse run configurations, to tell AUTOFOCUS 2 the correct plugin path w.r.t. eclipse starts.

1.3 Launching of AUTOFOCUS 2

```
ant run // for compiling and running AutoFOCUS 2
ant clean // for cleaning all compiled classes
```

1.4 Directory structure

The root directory contains the ant script build.xml with the targets run and javadoc.

- AutoFlex
application gui and editors
- lib
contains jars of afparser, mmgen, jaig, msc2stdinterface
- Metamodel
contains the metamodel sources
- Misc
referenced sources from metamodel
- Tools
some tools required for metamodel compilation e.g. JavaCC

- logging.cfg
log4j configuration

1.5 Compatibility

- Don't use spaces in the directory names. This is a limitation of the current ant implementation on Windows. Version 1.5.2 does have some problems with arguments passing to the javac task. May be relaxed later.
- If compilation gets interrupted (user request) or fails at certain points the incremental build system does not always recompile all required classes and misses some. Try calling ant target "clean" on af2/build.xml.

1.6 Tools FAQ

1.6.1 Ant

FAQ 1: I get "BUILD FAILED file:../af2/Software/build.xml:58: af2.global.properties (re-)generated. Please restart. (This is no fault.)"

A: A template file was got updated, just run the ant target again and everything should be fine.

FAQ 2: I get "BUILD FAILED AutomaticTest/build.xml:108: Could not create task or type of type: junit."

A: You need to find a copy of junit.jar, the code that does JUnit work, and tell Ant where it is. The easiest way to do this is to go to Window -> Preferences, Ant -> Runtime. In the Classpath tab click on Global Entries and then Add External JARs....

Eclipse already has a copy of JUnit, so in the dialog find your Eclipse plugins directory ([ECLIPSE_DIR]/plugins/) and go to the JUnit plugin (presently org.junit_3.8.1) and select junit.jar. Now all of your Ant scripts will know how to do the <junit> Ant task.

1.6.2 Eclipse

FAQ 1: Beim Ausführen eines Ant-Targets erhalte ich folgenden Fehler:

[javac] BUILD FAILED: file:../build.xml:xy: Error running javac.exe compiler"

A: Benutzerumgebungsvariablen setzen:

- JAVA_HOME auf das Verzeichnis des J2SDK (z.B. JAVA_HOME=C:\Programme\j2sdk1.4.x).
- PATH setzen bzw. erweitern um %JAVA_HOME%\bin;%JAVA_HOME%\lib

Unter Windows 2000/XP geschieht dies unter Start / Systemsteuerung / System / Erweitert / Umgebungsvariablen.

Die Fehlermeldung verschwindet auch, wenn im Ant Skript die entsprechende Javac Task Zeile auf `<javac ... fork="no" />` geändert wird. Dies führt allerdings zu FAQ 2.

FAQ 2: Beim Ausführen eines Ant-Targets schlägt das Löschen fehl.

A: Bitte build.xml anpassen, so dass der Javac Task im forked Modus aufgerufen wird: `<javac ... fork="yes" />`.

Siehe auch:

- FAQ 1
- "Windows Note:When the modern compiler is used in unforked mode on Windows, it locks up the files present in the classpath of the `<javac>` task, and does not release them. The side effect of this is that you will not be able to delete or move those files later on in the build. The workaround is to fork when invoking the compiler." <http://ant.apache.org/manual/CoreTasks/javac.html>

FAQ 3: Beim Ausführen eines Ant-Targets erhalte ich folgenden Fehler:

"[javac] BUILD FAILED: file:../build.xml:xy: Error creating temporary file"

A: Die Ursache ist der `<javac>` task, der im forked mode eine temporäre Datei anlegt. Dies ist zusätzlich abhängig von der Anzahl der zu übersetzenden Quelldateien (siehe auch <http://ant.apache.org/manual/CoreTasks/javac.html>). Die temporäre Datei wird in dem Verzeichnis angelegt in dem Eclipse gestartet wurde.

Abhilfe: Eclipse in einem Verzeichnis starten, in dem der Benutzer Schreibrechte besitzt. Wird Eclipse unter Windows beispielsweise mittels Desktop-Verknüpfung gestartet, so kann in den Eigenschaften dieser "Ausführen in" entsprechend angepasst werden. Alternativ kann auch das Eclipse Verzeichnis fuer den Benutzer zum Schreiben freigegeben (z. B. C:\Programme\Eclipse2.1) werden.

FAQ 4: Starten von Eclipse bricht mit folgendem Fehler ab:

'Problems during startup. Check the ".log" file in the ".metadata" directory of your workspace.'

A: Bitte das Eclipse Verzeichnis (bzw. wenn vorhanden Eclipse\workspace) fuer den Benutzer zum Schreiben freigegeben (z. B. C:\Programme\Eclipse2.1).

Beim Mehrbenutzerbetrieb, nicht C:\Programme\Eclipse2.1\workspace freigeben, sondern Eclipse mit der option "-data" starten. Der Option muss das Benutzer Home Verzeichnis mit Schreibrechten folgen (z.B.

"-data_Z:\username" oder "-data_C:\DOKUME~1\username"). Die Option kann hinter einer Desktop-Verknüpfung versteckt werden, so dass dies dem Benutzer keine Umstände bereitet.

FAQ 5: Beim Ausführen eines Ant-Targets erhalte ich folgenden Fehler:
"Cannot use classic compiler, as it is not available. A common solution is to set the environment variable JAVA_HOME to your JDK directory."

A:

- Anweisungen von FAQ 1 befolgen.
- Unter Window / Preferences / Ant / Runtime / Additional class-path entries / Add JARs... C:\Programme\j2sdk1.4.x\lib\tools.jar hinzufügen
- siehe auch http://www.eclipse.org/eclipse/faq/eclipse-faq.html#users_16

FAQ 6: Beim Klicken auf Ant Targets geschieht nichts.

A: Fehler in Eclipse 2.1; Workaround:

- Eclipse neu starten
- evtl. hilft auch Eclipse beim Start mehr Speicher zuzuweisen
siehe auch http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.user/tasks/running_eclipse.htm

FAQ 7: Beim Kompilieren von Ant Targets aus Eclipse werden Fehlermeldungen im Consolen Fenster angezeigt. Leider sind diese nicht mit dem Quellcode im Editorfenster verlinkt.

A: Fehler in Eclipse 2.1; Workaround:

Quelldateien nicht als Link ins Filesystem im Eclipse Workspace anlegen.

FAQ 8: Beim Ausführen eines Ant-Target erhalte ich folgende Fehlermeldung:
"BUILD FAILED: file:d:/home/local/workspace/AF2/Misc/build.xml:53: Unable to find a javac compiler; com.sun.tools.javac.Main is not on the classpath. Perhaps JAVA_HOME does not point to the JDK"

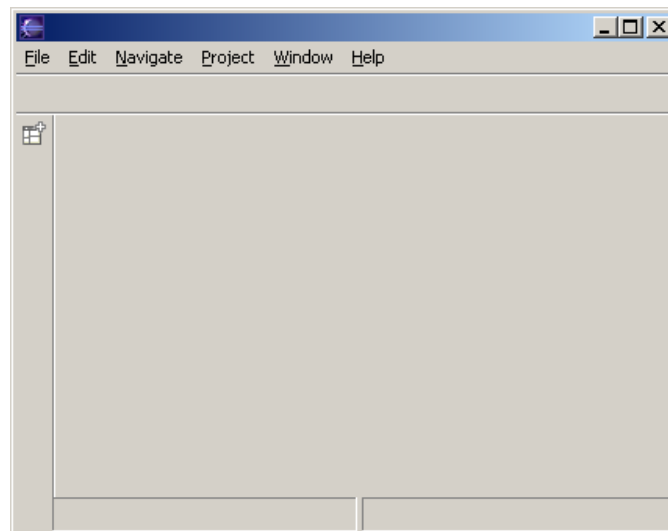
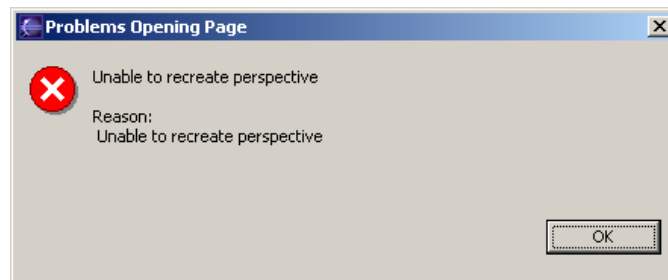
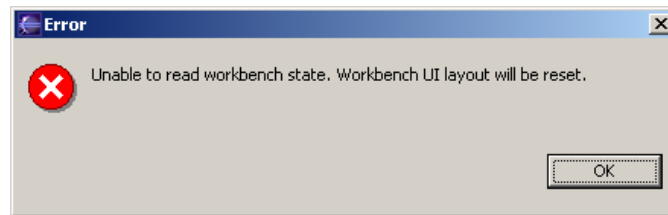
A:

- Gleiches vorgehen wie bei FAQ 5.

FAQ 9: Sonstige komische Fehlermeldungen beim Kompilieren von Quellcode mit Ant Skripten

A: Der Javac Ant Task hat unter Windows zumindest in der Ant Version 1.5.2 Schwierigkeiten mit Leerzeichen in Pfaden. Daher sollte der Eclipse Workspace in einem Verzeichnispfad liegen, der keine Leerzeichen aufweist.

FAQ 10: Corrupt or empty Workspace



A: Move or rename `eclipse_workspace/.metadata/.registry`. Before doing that exit Eclipse. The exact failure reason is unknown - no data loss encountered yet by deleting `.registry`.

FAQ 11: Sometimes the selection tree of Java Build Path Source tab Add Folder is empty.

A: This is a bug in Eclipse 3.0. Resize the window, after it opened and the contents shows up.

FAQ 12: I have compile problems using J2SDK 1.5 (Java Version 5) and Eclipse 2.x/3.x.

A: Currently AUTOFOCUS 2 does only work with J2SDK 1.4. Install the latest J2SDK 1.4 version. If you don't have administration privileges on your computer, you might not be able enable 1.4 as the default JVM. In this case supply the Eclipse command with the parameter "`-vm_PathToSDK1.4\bin\javaw.exe`".

FAQ 13: I get "Unsupported major.minor version 49.0", while running ant tasks.

A: You have compiled some files with j2sdK1.5.0 (=Java 5) and trying to execute them with j2sdk1.4.2 (=Java 1.4). Run "ant clean" to delete all files and recompile with only j2sdk1.4.2.

1.6.3 Together

FAQ 1: Starten bricht mit Fehlermeldung ab:

"Number of instances of Node-locked license xy exceeds the maximum allowed. Click Continue to select a different type of license."

A: Die Lizenzen sind nicht für genügend Rechner verfügbar. Togethersoft (bzw. Borland) ist benachrichtigt. Workaround:

- Netzwerkkabel vor Start von Together abziehen
- Together starten
- Netzwerkkabel wieder anstecken (allein schon wegen dem Benutzer Netzlaufwerk Z:)

2 Design

This chapter describes architectural styles for use in AUTOFOCUS 2.

2.1 Plugin mechanism

The plugin mechanism offers the ability to easily create and distribute multiple versions of AUTOFOCUS 2 with different features. A slim AUTOFOCUS 2 is required for example for students writing their diploma thesis in order to reduce the complexity and allow short orientation periods. Furthermore the plugin mechanism supports features of different maturity

without jeopardizing the overall application. The plugin mechanism allows to assemble the application out of components after compile time. It eases adding/removing of features without touching gui code, which formally always had to integrate different features.

The plugin mechanism currently offers the following features:

- a plugin is supposed to offer some functionality on a set of meta-model elements (MMElements)
- for each of these MMElements the context menu in the gui is extended with an entry allowing to access the plugin
- plugins are informed if they are supposed to quit and the plugin may perform e. g. data storage in this case
- plugins may use an external window for user in/output (JFrame; see DTD Editor for example) or an internal window inside AUTOFOCUS 2 (JInternalFrame). In the case of an internal frame the plugin should not worry about window management (e.g. bring to front, back, iconify, close). Instead the plugin mechanism is supposed to offer this functionality in a uniform way to all plugins. However this has not been done yet, but is very quick to implement after one agrees on how to use toolbars an internal frames (see `de.tum.autoflex.editor.gui.EditorFrame.java` for a start).

2.1.1 Usage of Plugin mechanism

- create a new directory `af2/Software/plugins/MyPluginName`
- inside `MyPluginName` create at least the subdirectories `src` for your source code and `resources` for resources like images and property files.
- create a plugin class. This class *must* obey to three conventions:
 - The class has to implement the interface `de.tum.autofocus.plugins.EditorPlugin`.
 - The class must have the same name as the jar achive in which you will distribute your plugin.
 - The class must be in the package `de.tum.autofocus.plugins`.
- copy an existing `build.xml` (e. g. `Editor/DTDEditor/build.xml`) and modify it to your needs
- add your plugin to `af2/Software/build.xml` and extend `af2.deploy.run.properties` with a boolean property declaring whether your plugin get's compiled in or not
- see JavaDoc for more information

Loading resources from plugins Loading resources like images or classes from plugins differs from conventional coding style. The reason is that the resources are bundled in the jar file of the plugin and need to be loaded from there instead of the classpath of the JVM.

Therefore the following two statements do not work in plugin environments since both try to look up the resources in the JVM classpath:

```
ClassLoader . getResource ()
ClassLoader . loadClass ()
```

Also this statement might cause problems:

```
object . getClass (). getResource ()
```

The reason is that `getResource()` extends the give resource name with the package name. This causes problems if your plugin in package `de.tum.autofocus.plugins` extends some super class in another package and this super class tries to load resources. The dynamic dispatch results in a wrong resource path in this case.

Here is the way that works for plugins as well as conventional coding:

```
object . getClass (). getClassLoader (). getResource ()
object . getClass (). getClassLoader (). loadClass ()
```

2.2 GUI components

The following section lists the most important classes of the AUTOFOCUS 2 GUI. They may be used to integrate new functionality

2.2.1 Menu bar

File: `de.tum.autoflex.base.mainframe.MainMenuBar`

Description: Drop down menus are created and managed here. This class listens to the `TreeBrowser` and some menus are enabled/disabled based on what's selected in the tree browser.

2.2.2 Tool bar

File: `de.tum.autoflex.base.mainframe.MainToolBar`

Description: This class manages the persistent toolbar which appears near the drop down menus. It contains some simple functionality to dynamically change based on what state the views are (currently it only changes based on whether a repository is open or not)

2.2.3 Repository

File: `de.tum.autoflex.RepositoryManager`

Description: (formerly called `QuestBrowser.java`) This class manages repositories. It can load/close repositories from the file system; it also keeps track of which repository is currently loaded, which one was last loaded. The class is also designed to keep track of whether any changes have been made to the currently open repository, but this hasn't been fully implemented yet due to incompatibilities between some of the editors. This tracking is done so some bits of the GUI can be enabled/disabled based on whether the repository has changed or not (e.g. save buttons).

2.2.4 Window Manager

File: `de.tum.autoflex.base.WindowManager`

Description: This class manages the creation and deletion/closing of different editor and view windows. It also takes care of some GUI aspects e.g. the help menu, the about box. However it should not contain any code that is shared across several components. This class is heavily intertwined with `Command.java`

2.2.5 Model browser with pop up menus

Files: `de.tum.autoflex.editor.gui.browser.*`

Description: These are classes associated with the tree browser (`de.tum.autoflex.editor.gui.browser.B`). New items can be made visible by extending the abstract class `ModelTreeNode` and adding the newly created class to `updateChildren()` of the parent object in the tree. Pop up menus may be defined in `getPopupMenu()`.

2.2.6 Logging mechanism with hyperlinks

You can now include HTML-like links in your log messages. These will be navigable in the error log, e.g. the browser tree will select the referenced element on clicking.

all you need is to insert a

```
<a href="?<rep-qual-name>">My element</a>
```

tag in your log message (don't forget the question mark!), where `<rep-qual-name>` is the "repository-wide qualified name" of the element, i. e. including the project name. so `component.getQualifiedName()` won't work, but

```
component.getProjectSuper().getName() + '.' + component.getQualifiedName()
```

will. If there's two elements of the same qualified name (e.g. a component A and a port A), the wrong element may be selected

see example usage in `de.tum.automode.clock.SimpleClockContext`

2.2.7 shared GUI logic

File: de.tum.autoflex.base.wm.Command

Description: This class is designed to contain all shared GUI functionality. Any feature that's duplicated across more than one GUI component should be pointed to this class. E.g. the Exit command can be invoked either

- from the File → Exit
- by clicking on the cross in the right-top corner of the program window
- by hotkey
- or by context menu

This class provides methods to:

- open editor windows
- create new model "items"
- rename/delete/annotate model "items"

Some methods are also provided here for "saving" and "error message" prompts; these are implemented to prevent duplication of code and to provide a consistent look and feel to the program.

2.3 Event propagation

Changes of the model are propagated to and from the model by the object `ModelChangeEvent`. Its constructor looks like this

```
| public ModelChangeEvent(Object source, Object modelElement, String changedAttrib
and holds the following information:
```

- the source of the changes:
 - another object of the application
 - or an object representing the user request
- the changed model element:
 - changed attributes are indicated by "changed attribute"
 - the parent model element in case of added/removed composition relations
- the name of the changed attribute

- the added new model element

Here an example for a new sub mode added to an existing parent mode:

```
inserter = new Inserter ();
inserter . setPosition ( Point );

parentModelElement . fireObjectInserted (
    new ModelChangeEvent ( inserter , pParentMode , "" , newMode ));
```

The Inserter contains a position that may be relevant to certain views.

2.4 View Component

While the metamodel component captures the concepts of the AUTOFOCUS 2 description technique, the view component captures additional information in order to layout and display different views like SSD, STD, EET on top of the metamodel. Therefore view components typically hold information like the position, size and color of metamodel elements. The main purpose of a view component is to provide an interface to the persistent view data independently of a possible editor technology.

Here's a list of some design decisions for the view component:

| design decision | reason |
|---|---|
| contains only persistent data | separation of concerns |
| all view data is independent of editor technology | separation of concerns |
| views and editors respect model changes | changes in one editor may result in changes in other editors (e.g. inserting new ports). Instead of each editor talking to other editors (adds too many dependencies), the views and editors observe the metamodel. <code>ModelChangeEvents</code> are used to propagate changes. Editors therefore often don't insert new figures directly, instead they propagate changes to the model, which informs them of the new figures to be inserted. |

2.4.1 SSDView

| view class | attribute | meaning |
|------------------|-----------|---|
| SSDPortView | Location | center of the port |
| SSDComponentView | Location | upper left corner of component |
| | Size | a <code>Rectangle</code> specifying the size of the component |

2.5 Editors

2.5.1 GEF

Since the documentation supplied with GEF has not been updated this section will give an introduction to the most important concepts of GEF. A simplified class diagram featuring the most important concepts of GEF is depicted in figure 7.

Modes GEF uses modes to dispatch user events like mouse clicks or keyboard strokes. Some modes can be recognized by the user: e. g. when there is a toolbar to select different tools to apply in the drawing area. GEF offers modes for some often used functionality: see `org.tigris.gef.base.ModeModify | ModeSelect` for examples which may be reused.

New modes should be implemented by extending `FigModifyingModeImpl`.

TODO: PlaceModes, Reuse Modes

ModeSwitch Mode switches may be performed by calling the corresponding methods of `ModeManager`, which is attached to every editor. However there are situations, where this is not the intended solution of the GEF framework and two of those exceptions are explained in this section.

using Globals: Since modes are handled by the `ModeManager`, which is attached to every editor, the editor needs to be known in order to perform mode switches. Consider the case, when the user selects a specific tool in a GUI toolbar. Usually the selection applies to the editor the mouse enters next after the selection. In most cases the tool selection corresponds to a GEF mode (see `ModeManger`). However the target editor, where the mode is to be set is not known in the moment of tool selection. For these cases the following mechanism is intended:

`org.tigris.gef.base.Globals` can store one mode (see `Method mode()`), which can be used for communication between different components of the GUI. This process is automated by GEF in the following sense: The toolbar puts the mode which corresponds to a specific tool into `Globals`. When the mouse enters a specific editor, this editor will retrieve and set the mode from `Globals`. In order to use this automatism all the developer has to do is attaching `CmdSetMode` to the toolbar (see for example `de.tum.autoflex.editor.gui.ssd.SsdToolBar`).

The *second mechanism* presented here is related to the modes itself. Some Modes may decide to finish when receiving certain events. An example is a mode which allows to draw new items and quits as soon as the mouse button is released. In this case the mode is supposed to call its `Mode.finish()` method, which will e. g. restore the original mouse pointer. It's very important to recognize that `Mode.finish()` will also activate the current Mode stored in `Globals` (or the default mode in case there is no current mode).

Be sure always having set the desired mode in `Globals.mode(Mode)` before calling `finish()`!

ModeManager A speciality of GEF is that there can be multiple active modes. This allows to separate the dispatching of user events. Different modes dispatch these events:

- **ModeDragScroll** Editor pane scrolls if the mouse pointer moves near the border of the drawing pane e. g. while dragging figures or rectangle selection.
- **ModePopup** Pop up text messages appear if the mouse moves over figures.
- **ModeSelect** Figures can be selected.

All modes are arranged on a stack. These just presented modes are the default modes of GEF, which cannot be removed from the stack (with `ModeDragScroll` on top). The topmost mode receives a user event first and may decide to react to it. The reacting mode may also decide whether the modes lower on the stack get the chance to see the event. If a mode calls `MouseEvent.consume()` no other modes get a chance to see them. Otherwise the next mode on the stack gets informed about the event.

Anomaly:

`ModeManager` stops propagating `MouseEvent.isConsumed()` down the stack if they are of type:

- key events
- `mouseClicked`
- `mouseEntered`
- `mouseExited()`

However the following events are still propagated (even if `MouseEvent.isConsumed()`) (at least in GEF v0.10.3 - v0.11.1):

- `mouseMoved`
- `mouseDragged`
- `mousePressed`
- `mouseReleased`

The implementation of `ModeSelect` does return immediately if consumed mouse events are received. The behavior is therefore the same as if `ModeManager` would stop propagating these events. However `MousePopup` and `ModeDragScroll` do always react to these events.

If stacking own modes eventually put a `if MouseEvent.isConsumed() return;` as the first operation in the `mouse*` methods!

LayerPerspective A Layer like found in many drawing applications. It contains a collection of `Fig`'s, ordered from back to front. Each `LayerDiagram` contains part of the overall picture that the user is drawing.

DefaultGraphModel Ports, Nodes, Edges

Globals Holds some global info for all editors. Also communicates between palettes and editors by holding the next global Mode.

Editors, Layers, Figs, NetPrimitives, Modes An Editor's `LayerManager` has a stack of `Layer`'s. Normally `Layers` contain `Figs`. Some `Figs` are linked to `NetPrimitives`. When `Figs` are selected the `SelectionManager` holds a `Selection` object. The behavior of the Editor is determined by its current Mode. The Editor's `ModeManager` keeps track of all the active Modes. Modes interpret user input events and decide how to change the state of the diagram. The Editor acts as a shell for executing `Commands` that modify the document or the Editor itself.

JGraph `JGraphFrame` is not used, since it launches an own `JFrame`. Instead `JGraph` is placed into a `JInternalFrame`.

2.5.2 GEF usage concept

Here's a list of some design decisions for the GEF usage concept, which are independent of a specific editor:

| | |
|---|---|
| only one GEF layer | simplicity; access with <code>Globals.curEditor().getLayerManager().getActiveLayer()</code> |
| use <code>Fig.getBounds()</code> , <code>Fig.getLocation()</code> with care | returns always the coordinates of the bounding box for the Fig (don't overwrite!). Often these values can't be directly used to get/set view layer data. |
| compose figures instead of overwriting <code>paint()</code> | composing complex figures with the GEF grouping (<code>Fig.addFig()</code> feature takes care that e.g. bounding boxes are correct. If you decide to overwrite the <code>paint()</code> method, always paint within the <code>getBounds()</code> rectangle |

2.5.3 Porting Property Editors

These are the necessary steps for porting property editors:

1. Make changes in the Editor code. For instance, in the SSD editor, the class `de.tum.quest.vertical.ssdeditor.base.MyModeSelect2` had to be modified. See the SSD editor regarding how the new Properties code should be invoked
2. Create new `XXXXProperties` class for each `ModelElement XXXX` that you desire to have properties for. As an example, copy `de.tum.autoflex.editor.properties.ssd.Com`
 Within this class, all that should be necessary to do is to define all the Attributes for that `ModelElement` that you *don't* want displayed. The easiest way to figure out what you don't want displayed is to just comment everything in the `defineExcludedProperties()` method for the time being, get the whole thing running, and then go back and exclude properties which you don't want shown.
3. Extend the `PropertiesFactory` class (`de.tum.autoflex.editor.properties.PropertiesFactory`) to include your new `XXXXProperties` class. Copy how it's done for the others.
4. Load up AF2 and see whether everything works. Check whether all the attributes display correctly; then check whether the attributes are editable. Some are not-editable by design, and you won't be able to change the corresponding cell in the table (this is true, for example, for Term). If you get an error when you try to set an attribute, it might be because that particular attribute type doesn't have a `GetterSetter` defined, but can't be handled by the `DefaultGetterSetter`. Please investigate how `GetterSetter` classes work for other complex types (e.g. `MIFTerm`, `MIFType`, `Comment`) in the classes `MIFTermGetterSetter`, `MIFTypeGetterSetter` etc, and in the `MetaModelApi` class.

5. Get rid of old Properties code so that it doesn't hang around, as much as possible.

Issues:

- strangeness with instances for ElementPropertiesGUI, Application-PropertiesGUI classes
- no GetterSetter defined for Type (see Type attribute in Port Properties) so errors are displayed when you try to edit it
- Clock attribute of Port uses Term classes, which don't seem to be editable (Term.toText() used as getter). So they have no setter and are thus read-only in the table

2.5.4 GUI highlighting

Highlighting of graphical objects is required by the simulation to display the current state. The message for highlighting a graphical object is encoded in a SimulationValueChangedEvent. Its constructor looks like this

```
public SimulationValueChangedEvent(Object parent, Object element, String message
```

and holds the following information:

- the parent element (in terms of metamodel composition) of the element to be highlighted
- the element to be highlighted
- "1"=highlight, "0"=normal or just the numerical values if those are displayed for current assignment to ports or variables

Here an example for highlighting of a mode:

```
SimulationValueChangedEvent se = new SimulationValueChangedEvent((Mode)aMode,
aMode.fireAttributeChanged(se);
```

2.6 Visitors for traversing trees of model objects

Overview. AUTOFOCUS 2 supports a special technique for traversing trees of objects called *Visitor pattern*. The idea itself is explained in some detail in Gamma et.al., "Design Patterns", 1995. By implementing a Visitor, a developer can perform some check or operation on a tree of objects (such as the object tree representing AUTOFOCUS 2's model, where parent/child relations correspond to the composition relations in the metamodel) without having to add any code to the metamodel classes themselves. In our particular implementation of the Visitor pattern, the code for traversal

of the tree is usually *not* part of the Visitor class itself. For a comparison of conventional and Visitor programming, consider the following non-Visitor example for recursively checking a hierarchical network of components, using some method `check(Component)` (not shown):

```

void myCheck(Project project) {
    for (Iterator components = project.getComponentsIterator();
        components.hasNext();
        ) {
        Component component = (Component)components.next();
        myCheck(component);
    }
}
void myCheck(Component component) {

    check(component);

    for (Iterator subComponents = components.getSubComponentsIterator();
        subComponents.hasNext();
        ) {
        Component subComponent = (Component)subComponents.next();
        myCheck(subComponent);
    }
},

```

Implementing the same operation as above with AUTOFOCUS 2 visitors looks simply as follows:

```

void myCheck(Project project) {
    project.accept(new CheckVisitor());
}

class CheckVisitor extends MetamodelVisitorAdapter {
    public void visit(Project project) {}
    public void visit(Component component) {
        check(component);
    }
}

```

Note that we have done three things in order to get `check(Component)` going:

- Define a class inheriting from `validas.metamodel.visitor.MetamodelVisitorAdapter`
- Override the `visit()` methods of interest with application-specific code
- Call `accept()` on the root element of the model part to be traversed (in this case, the `project` element)

A depth-first traversal of `project`'s model object tree is then performed automatically, without any additional code. The developer may therefore

concentrate on the check or operation performed by the Visitor itself, and tedious traversal code is avoided, making code much shorter and easier to maintain. As an additional benefit, case distinctions based on the runtime class of an object (`instanceof`), such as in the following non-Visitor example:

```
void myCheck(Term term) {
    //a case distinction here
    if (term instanceof Appl) {
        check((Appl)term);
        for (Iterator args = (Appl)term.getArgsIterator();
            args.hasNext();
            ) {
            myCheck((Term)args.next());
        }
    }
    //another case distinction here
    else if (term instanceof Const) {
        check((Const)term);
    } ...
},
```

can be more clearly written as a Visitor:

```
void myCheck(Term term) {
    term.accept(new TermCheckVisitor());
}
class TermCheckVisitor extends MetamodelVisitorAdapter {
    public void visit(Appl appl) { check(appl); }
    public void visit(Const _const) { check(_const); }
    ...
}
```

(Note that some common methods (`check(Appl)`, `check(Const)`) are not shown in the example). Again, the depth-first traversal of an application's (`Appl`) arguments is automatically done by the Visitor mechanism.

Usage. The usage will be clarified by the detailed example shown in Section 2.8. Briefly, a Visitor class is created as follows:

- Create a class that inherits from `MetamodelVisitorAdapter`
- Override `visit()` methods (or, in addition, `leave()` methods for the bottom-up traversal for those metamodel classes that the Visitor shall traverse.
- Call `<root object>.accept(<visitor>)` for the object at the root of the tree to be visited

Note that the Visitor class must also override `visit()` methods (with possibly empty method body) for those classes that have to be traversed between the root object and those objects where the actual work is done. This includes the class of the root object itself, like the empty `visit()` handler for class `Project` in the above example. For instance, if a Visitor does some operation on all `LocVariable` objects of a model, and the traversal starts with a `Repository` object, then the following `visit()` methods must be overridden:

- `visit(Repository),`
- `visit(Project),`
- `visit(Component),`
- `visit(LocVariable),`

even though only the `visit(LocVariable)` method has a nonempty method body.

Caution with views. For efficiency and maintainability reasons, the views of a given model element are only traversed if the visitor overrides a `visit()` or `leave()` method for the root view class. For instance, the code of `Component.accept` will only call `visit()` on the component's views if the visitor declares `visit(SSDComponentRootView)` or `leave(SSDComponentRootView)`:

```
//validas.metamodel.Component
if (visitor.visits(SSDComponentRootView.class)) {
    for (Enumeration views = getViews(); views.hasMoreElements();) {
        ((Accepting)views.nextElement()).accept(visitor);
    }
}
```

So in case you'd like views to be traversed by your visitor (e.g. `SSDComponentView`) you should override the `visit()` or `leave()` method for the corresponding root view (e.g. `SSDComponentRootView`) in any case, possibly with an empty method body if no processing is necessary for the root view.

2.7 Management of a model's status using StatusManager

Motivation. Motivation für diese Klasse die Beobachtung, dass in bestimmten Teilen von AUTOFOCUS/AUTOFOCUS 2 (z.B. Simulator, Clock checker, Import) bis zu 3/4 des Codes aus "Konsistenzchecks" besteht. In einigen Fällen sind es auch automatische "Transformationen" wie z.B. Views reparieren beim Import. Diese Checks und Transformationen werden immer wieder neu geschrieben, sind aber trotzdem je nach Anbindung unvollständig. Man kriegt also je nach Anbindung unterschiedliche Fehlermeldungen, z.T. ist das Zeug unvollständig, bei Änderungen am MM etc.

hat man den mehrfachen Aufwand bezgl. Wartung usw.usw. Das Package `de.tum.autoflex.status` mit der zentralen Klasse `StatusManager` bzw. `TransformingStatusManager` löst das Problem auf folgende Weise:

- Es gibt eine zentrale Registry `AbstractStatusManager`, wo sich alle Checks und Transformationen registrieren können.
- Die Registry verwaltet eine Reihe von symbolischen Konsistenzannahmen/Invarianten, im folgenden "Status" genannt.
- Status-Elemente sind (partiell) durch eine Implikationsrelation geordnet, z.B. "Simulateable" => "AllComponentsHaveBehavior"
- Die Registry verwaltet weiterhin eine Reihe von Checks, die jeweils für einen "assumed" Status bei Erfolg einen (stärkeren) Status "garantieren"
- Eine erweiterte Version der Registry (`StatusManager` verwaltet eine Reihe von Transformationen, die jeweils für einen "assumed" Status bei Erfolg einen (stärkeren) Status "garantieren" bzw. herbeiführen
- Ein Check (ohne Seiteneffekte sprich Änderungen des Modells) implementiert das Interface `StatusChecker` und gibt bei Aufruf seiner Methode `hasSideEffects()` den Wert `false` zurück. Eine Transformation (mit Seiteneffekten) implementiert ebenfalls `StatusChecker` und gibt für `hasSideEffects()` den Wert `true` zurück.
- Für Checks und Transformationen existiert eine komfortable Adapterklasse, `StatusCheckerAdapter`.
- Als *eigentlichen* Nutzen für Anwendungsentwickler verwaltet die Registry für alle notwendigen MM-Objekte die aktuelle Status-Information und erlaubt das Ausführen der notwendigen Checks/Transformationen auf der Basis des aktuellen Status. Für Checks kann der Anwendungsentwickler also mit einem einfachen

```
if (StatusManager
    .instance()
    .check(mmelement, "MyStatus")) {...}
```

sicherstellen, dass er ohne weitere Checks bestimmte Sachen einfach voraussetzen kann.

Für eine Transformation wäre ein Beispiel die Transformation "view transformer", die für den assumed Status "null" (keine Annahme) den guaranteed Status "ViewsNicelyLayouted" herbeiführt. Diese ist in Abschnitt 2.8 beschrieben. Meinetwegen gilt dann noch

```
"ViewsNicelyLayouted" => "ModelElementsHaveViews"
```

usw.usw.

It is preferable to register all status elements before doing the first check. Otherwise, some re-caching will be performed.

2.8 Example: Using Visitor to build a StatusChecker

The following class `ViewTransformer` in Figs. 8 and 9 implements a `StatusChecker` with side effects (transformer) using the `Visitor` mechanism. Its purpose is to guarantee the status `ViewsNicelyLayouted` based on a null status assumption, i.e. no assumption. We left out the code dealing with the layout issues themselves, concentrating only on the `StatusChecker` and `Visitor`-related aspects.

In order to use `ViewTransformer`, the status `ViewsNicelyLayouted` and the transformer `ViewTransformer` first have to be registered with class `StatusManager`:

```
TransformingStatusManager manager =
    TransformingStatusManager.instance();
manager.addStatus("ViewsNicelyLayouted");
manager.addStatusChecker(ViewTransformer.instance());
```

Note that the `instance()` method ensures that there is only one globally available object of classes `TransformingStatusManager` or `ViewTransformer`, respectively.

After both the status and the status transformer have been registered, the call

```
TransformingStatusManager manager =
    TransformingStatusManager.instance();
MMElement myElement = ...;
try {
    manager.transform(myElement, "ViewsNicelyLayouted");
} catch (NoSuchCheckerException e1) {
    e1.printStackTrace();
}
```

will cause `myElement` to be transformed by `ViewTransformer` if the status `ViewsNicelyLayouted` does not already hold for `myElement`. Note that the caller has to take care of `NoSuchCheckerExceptions`, which correspond to programming errors (e.g. if the developer forgot to register `ViewTransformer`).

3 Configuration

The configuration of the application has changed since `AUTOFLEX`.

```
package de.tum.autoflex.base;

import org.apache.log4j.Category;
import validas.metamodel.model.*;
import validas.metamodel.views.SSDView.*;
import validas.metamodel.views.STDView.*;
import validas.metamodel.views.EETView.*;
import validas.metamodel.visitor.MetamodelVisitorAdapter;
import de.tum.autoflex.status.StatusCheckerAdapter;

public class ViewTransformer extends StatusCheckerAdapter {
    static Category _log = Category.getInstance(ViewTransformer.class);
    static ViewTransformer _instance = null;

    public static ViewTransformer instance() {
        if (_instance == null) {
            _instance = new ViewTransformer();
        }
        return _instance;
    }

    public String getGuaranteedStatus() {
        return "ViewsNicelyLayouted";
    }

    public String[] getAssumedStatus() {
        return null;
    }

    public String getName() {
        return "view_transformer";
    }

    public boolean hasSideEffects() {
        return true;
    }

    public boolean check(Project project) {
        ViewTransformVisitor vtv = new ViewTransformVisitor();
        project.accept(vtv);
        return vtv.result;
    }
}
```

Figure 8: Example class ViewTransformer, Part 1

```
public boolean check(Component component) {
    ViewTransformVisitor vtv = new ViewTransformVisitor();
    component.accept(vtv);
    return vtv.result;
}

class ViewTransformVisitor extends MetamodelVisitorAdapter {
    //some member declarations here
    //[...]

    Project lastProject;

    //always true
    boolean result = true;
    public void visit(Project _project) {

        //using a member attribute to pass the current project down to subse
        lastProject = _project;
    }

    public void visit(Component _component) {
        //some component-specific processing here
        //we can refer to lastProject if we like
        //[...]
    }

    public void visit(Automaton _automaton) {
        //some automaton-specific processing here
        //we can refer to lastProject if we like
        //[...]
    }

    public void visit(MSC _msc) {
        //some MSC-specific processing here
        //we can refer to lastProject if we like
        //[...]
    }
}
}
```

Figure 9: Example class ViewTransformer, Part 2

3.1 Justification

Changes to the Application Configuration code were implemented for the following reasons:

- to improve readability of code
- to cut amount of code repetition
- to improve maintainability of code

3.2 Description

The configuration code has to provide the following functionality:

1. allow internal classes access to "system-wide" properties e.g. default Look&Feel of the program
2. allow the user to change these properties via a GUI in the program
3. allow the user to change these properties in a config file

Below is a description of how each one is now implemented.

3.2.1 Allow internal classes access to "system-wide" properties

There exists a class called Configuration. This class stores and controls access to system-wide properties. It's best to think of this class as a wrapper for a `java.util.Properties` object (which is itself basically a text hashtable). The Configuration has two main methods:

```
setProperty (String name, String value)  
getProperty (String name)
```

`setProperty` is passed a "name" String of the form "abc.def.ghi", and some value String. "abc.def.ghi" is the name of the property which is about to be set. The value String is then saved with the name as the key. Similarly, `getProperty` retrieves the value of the property with the key String.

All the properties are stored in name, value String pairs; this is done to simplify saving/loading the configuration information to/from the hard drive.

However, most of the properties are not in fact Strings, but some other type of information e.g. Color. This means that the requestors of property information are more interested in the underlying Object that the property represents, not its string representation. Because of this, the Configuration class contains some parsing methods; one for each datatype. Consider an example:

Say, we are interested in the property with the key "LinkSegment.Color". The call

```
String str = Configuration.getProperty("LinkSegment.Color"); (1)
```

will yield a string representation of the `LinkSegment.Color` property. However since we are probably interested in the `Color` object, the call:

```
Color c = Configuration.parseColor(Configuration.getProperty("LinkSegment.Color"));
```

will retrieve the underlying `Color` object stored in the property.

The String representations of most data types are the obvious `toString()` return String (e.g. for numbers). For `Color` objects, it is the RGB integer value (stored as a String, obviously) obtained by the `Color.getRGB()` method. Other things, e.g. `LabelledStrings`, are stored as a "label:data" String.

At the moment, there is no "reverse parsing" available (e.g. which systematically turns the Object into its String representation to be saved). This means that to save something, you have to know its representation. e.g.

```
Color c = Color.green;
Configuration.setProperty("LinkSegment.Color", c.getRGB()+"");
```

will save the `Color` object back into its property.

Suggestion: *Implement reverse parsing for every data type so that code trying to retrieve properties doesn't have to know the representation.*

3.2.2 Allow the user to change these properties via a GUI in the program

Application properties are displayed in table form. They are also logically grouped into tabs; there is a table for each tab.

Each table is a `IstinlineJTable` object, coupled with a `PropertyModel` object (which extends `TableModel`). The `PropertyModel` supplies the `Model` (data) for the table. The `PropertyModel` is itself also quite simple; it contains a list of `Property` objects. Each `Property` object is displayed as a row on the table. If a change is made in the table, the `PropertyModel` passes the change onto the `Property` object.

The `Property` object contains most of the functionality. It has a name and value, and a `Type` object associated with it. Because each `Property` object represents an application property, the name is of the form "abc.def.ghi". The value is the value of the application property associated with the name. The `Type` object provides:

- the underlying class of the property
- a table cell renderer for this property
- a table cell editor for this property
- a string representation for this type of property

Table cell renderers and editors specify to the `JTable` class how to present the provided information. Renderers govern how things are displayed. E.g. numbers and strings are displayed in the obvious way; however Colors are displayed by colouring the table cell in the colour being represented. Editors govern how the cell behaves when being edited (again numbers and strings are edited in the default way, where as other things are displayed as comboboxes etc).

For our implementation, a few custom editors and renderers are provided for improved usability (i.e. to limit the amount of "bad" data which the user can enter into the system).

Suggestion: *Some data types have either no error checking in-built (e.g. `Float`, `Double`, `Long`) or very rudimentary (e.g. `Integer`). This can be implemented by creating custom editors, instead of using the default ones.*

When the user makes a change to the property in the table, (as mentioned above) the table model notifies the underlying `Property` object; the `Property` object then saves the property back into the `Configuration` object. The string representation of the property is obtained from the `Type` object.

The `Property` object also contains a lot of information which is consequently used by the model and the table to display the property. For instance, the label to be displayed in the table, or the possible choices (the range) of the property, if available/applicable, or in which tab the object is displayed. For this purpose each displayable property in the `Configuration` has some "helper" properties associated with it. For example, the property:

```
"Font.Type"
```

will imply that its "helper" properties also exist:

```
"Font.Type.propDesc" // contains the label shown in the table
"Font.Type.propType" // e.g. String, Color, ComboBox, LabelledComboBox etc
"Font.Type.propOptions" // (this one is specific to properties of type
"Font.Type.displayInTab" // in which tab this property is to be displayed
```

At the initialisation of a `Property` object, an attempt will be made to read in all the necessary helper properties, to glean all the necessary information.

The number/names of tabs are not hard-coded in any place; when the application property window is instantiated, all available properties are looked over; the ones that are displayable are turned into `Property` objects. Then for each different tab name in these `Property` objects, a new tab is created.

3.2.3 Allow the user to change these properties in a config file

The `Configuration` class is able to read properties from a file, and to save them to a file. Because the `Configuration` class uses a `Properties` object (which provides file saving functionality) saving the configuration is very simple.

Loading configuration is also simple; first, default properties are created (by the `Configuration.defineDefaultProperties()` method), then the file properties are read; finally the method `Configuration.overwriteWith(Properties)` overwrites the default properties with ones which are defined in the file.

Upon exiting AUTOFOCUS 2, a save is called on the Configuration, dumping the configuration to file.

3.3 Miscellaneous notes

3.3.1 SuperTableCellEditor, SuperTableCellRenderer

The application properties JTable uses SuperTableCellEditor and SuperTableCellRenderer as the cell editor and renderer, respectively. This is done because of a limitation of Sun's JTable code. For a JTable, you are allowed to specify different renderers and editors for different columns and classes. E. g. you can specify that editor X will be used to display Strings in column 2, and editor Y will be used to display Colors in column 2. What you *cannot* do is specify different editors for cells which have the same column and Class. This presents a problem if I want to display a normal String in row 1, and a combobox of Strings in row 2.

To get around this, special "super" editor and renderer classes have been created; they keep track of the editors/renderers for each cell, and use the right one when a specific cell is selected.

3.3.2 LabelledString

This is a new class, created because of the need to sometimes display one thing and have something else as the underlying data. E.g. when displaying choices for Look&Feel, the label for a choice is its name e.g. "pvc", and the underlying data is the class responsible for that look&feel. That way, when the choice is presented to the user, the user only sees "pvc" in the GUI. However when "pvc" is selected the system immediately knows which class it should use for the Look&Feel. A labelled string is usually represented (e.g. in configuration files and in the Configuration class) as a label:data pair, separated by a colon.

3.3.3 ComboBox, LabelledStringComboBox

A ComboBox and a LabelledStringComboBox are things used by the application properties GUI to display information. A combobox is a straight choice of Strings, where as a LabelledStringComboBox is (as the name suggests) a choice of LabelledStrings. The application properties which use these will have extra "helper" properties defined in the configuration file/the default

configuration (in the Configuration class): ".propOptions" and ".propLabelledOptions" for ComboBox and LabelledStringComboBox, respectively (see the Configuration.defineDefaultProperties() method for examples)

A GEF Discussions

GEF offers a rich API and often several solutions are possible to implement a certain feature. This section will summarize some of the discussions we had on how to use GEF. It's here for reference purpose.

A.1 How to use Grouping

Grouping of graphical shapes is a standard feature of graphical frameworks. The impact of using grouping is summarized here.

grouping component and port:

- + since component and ports are selected as one item, GEF automatically moves the ports when dragging the component. For Reference: GEF ModeModify.mouseDragged() takes care of that.
- o selection of port needs to be treated separately since the whole group is selected by default. This means additional work, but since the treatment of moving of ports on the border of components is special anyway, it does not mean any additional coding effort in this case.
- the selection frame (automatically drawn by GEF) of the component has different sizes depending on whether the component has ports or not. This could be fixed by overriding FigComponent.makeSelection(), but results in a lot of GEF redundant code, which would have to be copied from SelectionManager.makeSelectionFor() and e. g. SelectionResize.paint().

not grouping component and port:

- Dragging of components does not also move their ports. Either selection has to change when selecting components (also select ports automatically) or ModeModify.mouseDragged() has to be changed to also include moving of ports.

Both solutions do not seem perfect. Grouping of components and ports is currently preferred, since it does result in less "copying" of GEF code, which reduces maintenance efforts with evolving GEF frameworks.

A.2 **Overriding paint()**

Overriding of the `paint()` method in figures is possible. But the following issues should be known:

- Don't draw beyond `getBounds()`, otherwise screen refresh get's into problem. The result is pixel trash when moving objects.
- Maybe overriding `paint()` could be avoided by composing the current figure of other primitives e. g. by grouping. See `FigComponent`.