

# ADAMS

Advanced Data mining And Machine learning System

Module: adams-core

0000111000011111111000001110000110000011001111110  
0001101100011000001100011011000111000111011000011  
001100011001100000110011000110011110111011000000  
011000001101100000110110000011011011011001111110  
01111111101100000110111111110110000011000000011  
0110000011011000001101100000110110000011011000011  
011000001101111111001100000110110000011001111110  
0000111000011111111000001110000110000011001111110  
00011011000110000010001101000111000111000011  
001100011100000110000011011000011011000000  
011000001100000110000011000001101111110  
01111111101100001101111111011000001100000011  
0110000011011000001101100000110110000011011000011  
011000001101111111001100000110110000011001111110  
0000111000011111111000001110000110000011001111110  
000110110001100000110001100111000111011000011  
00110001100110000011001100011001111011101100000  
011000001101100000110110000011011011011001111110  
0111111110110000011011111111011000001100000011  
0110000011011000001101100000110110000011011000011  
000110110001100000110001100111000111011000011  
01100001100110000011001100011001111011101100000  
01100000110110000011011111111011000001100000011  
0110000011011000001101100000110110000011011000011  
011000001101111111001100000110110000011011000011  
011000001101111111001100000110110000011001111110

Peter Reutemann

March 5, 2012

©2009-2012



THE UNIVERSITY OF  
**WAIKATO**  
*Te Whare Wānanga o Waikato*



Except where otherwise noted, this work is licensed under  
<http://creativecommons.org/licenses/by-sa/3.0/>

# Contents

<b>I</b>	<b>Using ADAMS</b>	<b>7</b>
<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Flows</b>	<b>11</b>
2.1	Actors . . . . .	11
2.2	Creating flows . . . . .	13
2.2.1	Hello World . . . . .	14
2.2.2	Processing data . . . . .	22
2.2.3	Control actors . . . . .	26
2.2.3.1	Have some <i>Tee</i> . . . . .	26
2.2.3.2	Pull the <i>Trigger</i> . . . . .	27
2.2.3.3	Branching – or how to grow your flow . . . . .	28
2.2.3.4	Further control actors . . . . .	29
2.3	Global actors . . . . .	31
2.4	External actors . . . . .	32
2.5	Templates . . . . .	33
2.5.1	Static use . . . . .	33
2.5.2	Dynamic use . . . . .	33
2.6	Variables . . . . .	35
2.7	Temporary storage . . . . .	39
2.8	Debugging your flow . . . . .	42
<b>3</b>	<b>Visualization</b>	<b>45</b>
3.1	Image viewer . . . . .	45
3.2	Preview browser . . . . .	46
<b>4</b>	<b>Tools</b>	<b>49</b>
4.1	Flow editor . . . . .	49
4.2	Flow runner . . . . .	49
4.3	Flow control center . . . . .	49
4.4	Text editor . . . . .	49
<b>5</b>	<b>Maintenance</b>	<b>51</b>
5.1	Placeholder management . . . . .	51
5.2	Named setup management . . . . .	51
5.3	Variable management . . . . .	51
5.4	Favorites management . . . . .	51

<b>6 Customizing ADAMS</b>	<b>53</b>
6.1 Main menu . . . . .	53
6.2 Properties files . . . . .	53
 <b>II Developing with ADAMS</b>	 <b>55</b>
<b>7 Tools</b>	<b>57</b>
7.1 Subversion . . . . .	57
7.2 Maven . . . . .	57
7.2.1 Nexus repository manager . . . . .	57
7.2.2 Configuring Maven . . . . .	58
7.2.3 Common commands . . . . .	59
7.2.4 3rd-party libraries . . . . .	59
7.2.4.1 Installing locally . . . . .	59
7.2.4.2 Uploading to Nexus . . . . .	60
7.3 Eclipse . . . . .	61
7.3.1 Plug-ins . . . . .	61
7.3.2 Setting up ADAMS . . . . .	61
 <b>8 Using the API</b>	 <b>63</b>
8.1 Flow . . . . .	63
8.1.1 Life-cycle of an actor . . . . .	63
 <b>9 Extending ADAMS</b>	 <b>65</b>
9.1 Dynamic class discovery . . . . .	65
9.1.1 Additional package . . . . .	65
9.1.2 Additional class hierarchy . . . . .	66
9.1.3 Blacklisting classes . . . . .	66
9.2 Creating a new actor . . . . .	66
9.2.1 Creating a new class . . . . .	67
9.2.2 Option handling . . . . .	68
9.2.2.1 Example . . . . .	68
9.2.3 Variable side-effects . . . . .	69
9.2.4 Graphical output . . . . .	69
9.2.5 Textual output . . . . .	69
9.2.6 Creating an icon . . . . .	69
9.2.7 Creating a JUnit test . . . . .	69
9.3 Creating a new module or project . . . . .	70
9.3.1 Module . . . . .	70
9.3.2 Project . . . . .	71
9.4 Main menu . . . . .	71
 <b>Bibliography</b>	 <b>73</b>

# List of Figures

2.1	Flow editor with an empty new flow ( <i>File</i> → <i>New</i> → <i>Flow</i> ) . . .	13
2.2	Popup menu for adding a new actor . . . . .	14
2.3	Selecting a different actor . . . . .	15
2.4	Searching for <i>StringConstants</i> actor . . . . .	16
2.5	Help dialog for the <i>StringConstants</i> actor . . . . .	16
2.6	Tab displaying help for the <i>StringConstants</i> actor . . . . .	17
2.7	Tab displaying the non-default options for the <i>StringConstants</i> actor . . . . .	18
2.8	Adding the <i>Hello World!</i> string . . . . .	18
2.9	Flow after adding the <i>StringConstants</i> actor . . . . .	19
2.10	Adding another actor <i>after</i> the current one . . . . .	19
2.11	Searching for the <i>Display</i> actor . . . . .	20
2.12	The complete <i>Hello World</i> flow . . . . .	20
2.13	The output of <i>Hello World</i> flow . . . . .	21
2.14	Adding an additional actor . . . . .	22
2.15	Adding the <i>Convert</i> transformer . . . . .	23
2.16	Configuring the <i>Convert</i> transformer . . . . .	23
2.17	Extended <i>Hello World</i> flow . . . . .	24
2.18	Output of the extended <i>Hello World</i> flow . . . . .	24
2.19	Customizing the <i>StringReplace</i> transformer . . . . .	24
2.20	Output of further extended <i>Hello World</i> flow . . . . .	25
2.21	The <i>Hello World</i> flow with <i>Tee</i> actors. . . . .	26
2.22	The log file generated by the <i>Tee</i> actors. . . . .	27
2.23	A customized <i>ConditionalTee</i> actor. . . . .	27
2.24	The <i>Hello World</i> flow with an additional <i>Trigger</i> actor. . . . .	28
2.25	The modified log file generated with the additional <i>Trigger</i> actor. . . . .	28
2.26	The <i>Hello World</i> flow using a <i>Branch</i> actor. . . . .	29
2.27	Outputting parallel processed strings in a single global <i>Display</i> actor. . . . .	31
2.28	Adding a sub-flow generated from a template to an existing flow. . . . .	33
2.29	The options of the <i>UpdateVariable</i> template. . . . .	34
2.30	The added sub-flow. . . . .	35
2.31	The asterisk (“*”) next to an option indicates that a variable is attached. . . . .	36
2.32	Using a variable to control what file to load and display. . . . .	36
2.33	Using a variable to control what external flow to execute (flow). . . . .	37
2.34	Using a variable to control what external flow to execute (output). . . . .	37
2.35	Flow demonstrating the temporary storage functionality. . . . .	39

2.36	Output of flow demonstrating the temporary storage functionality.	40
2.37	Flow demonstrating the LRU cache storage functionality. . . . .	40
2.38	Display of the temporary storage during execution. . . . .	41
2.39	Output of flow demonstrating the LRU cache storage functionality.	41
2.40	The control panel of the <i>Breakpoint</i> actor. . . . .	42
2.41	Example flow with <i>Breakpoint</i> actor. . . . .	43
2.42	The <i>Expression watch</i> dialog of the <i>Breakpoint</i> actor. . . . .	43
2.43	The <i>Inspection</i> dialog of the <i>Breakpoint</i> actor for the current token.	43
2.44	The <i>Inspection</i> dialog of the <i>Breakpoint</i> actor for the current flow.	44
3.1	Displaying a fractal in the Image viewer. . . . .	45
3.2	Preview browser displaying an image. . . . .	46
3.3	Preview browser displaying a flow. . . . .	46
3.4	Preview browser displaying an plain text file. . . . .	47
7.1	texlipse configuration for the <i>adams-core</i> module. . . . .	62

**Part I**

**Using ADAMS**





# Chapter 1

## Introduction

ADAMS is the result of a research project processing spectral data that required extensive preprocessing and parallelism. The workflow approach seemed the best way of dealing with this problem. A system was required, that was easy to extend and make it easy for the user in setting up workflows quickly.

Initially, the workflow engine of choice was Kepler [1], being both written in Java and designed for the science community. In order to bring machine learning to Kepler, the KeplerWeka project [2] was started. Over time we realized that we spent a lot of time rearranging actors (i.e., the nodes in the workflow) on the workflow canvas, whenever we needed to add more preprocessing or another layer of complexity to it. Even with Kepler's support for sub-workflows (which are opened up in separate windows), it soon became apparent that this was not an optimal solution.

Since most of the processing merely was loading data from the database and then preprocessing it (forking as well and different preprocessing in parallel), we decided to implement a very basic workflow engine ourselves, with a tree-like structure (1-to-1 and 1-to-n connections). Using a simple *JTree* for representing the structure of the flow, implied the relationship between the actors and no time was spent on rearranging them anymore. We could finally concentrate on setting up the flow to process the data.

Over time, ADAMS grew and more and more actors for various domains (machine learning, scripting, office, etc.) were added. Not all projects that use ADAMS as base-platform needed all the available functionality. This initiated the modularization of ADAMS and represents the current state of the platform. Derived projects now merely have to add dependencies to existing modules in order to gain additional functionality – without hassle.

*Have fun – The ADAMS team*



## Chapter 2

# Flows

Workflows, or *flows* for short, are at the center of ADAMS. Most activities can be expressed in a series of steps. Using a flow to define them is just logical. The advantage of using flows to describe activities is they document all the steps that are happening, making it easy to reproduce results. For instance for machine learning experiments, reproducibility is very important. Therefore, capturing every step, from the preprocessing of the raw data to the actual running of experiments and evaluating them, is essential.

The following section introduces the basic concepts of flows in the ADAMS context and how to set them up. Advanced topics are covered as well, like global actors and variable support.

### 2.1 Actors

A single step or node in a flow is called *actor*. There are various kinds of actors:

- **standalone** – no input, no output
- **source** – only generates output
- **transformer** – accepts input and generates output
- **sink** – only accepts input

A special kind of actor is the **control** actor, which controls the flow execution in some way. This can be a simple *Stop* actor, which merely stops the whole flow when executed. Or, it can be a *Branch* actor, which forwards the input it receives to all its sub-branches.

An actor that accepts input, like a *transformer* or *sink* is called an *InputConsumer*. An actor that generates output, like a *transformer* or a *source*, is called *OutputProducer*.

Each *InputConsumer* returns what types of data it does accept and is able to process. For some actors, this changes based on the parameters. The same applies to *OutputProducer* ones, which also return what type of data they are generating. Once again, the type of output data can change, depending on parametrization.

Before a flow is being executed, the compatibility of the actors is checked. This includes basic checks like the one that no *InputConsumer* can come after a *sink*, since that one doesn't generate any output. Additionally, the types of

output generated and the types of generated input are checked whether they are compatible. If one transformer generates floating point data, but the next transformer only accepts strings, this will result in an error.

There are two special types of data that an actor can return for accepted input or output:

- *object* – which can be any type, but not an array.
- *unknown* – which can be any type, even an array.

Data itself is not being passed around directly, but in a container called *Token*. This container allows additional storage, like provenance information. Actors that support provenance – *ProvenanceHandler* – update the provenance information in the container before forwarding it.

## 2.2 Creating flows

The basic flow layout is as follows:

- *[optional] standalone(s)* - for static checks or static operations when the flow is started
- *source* – for generating tokens that will get processed by subsequent actors.
- *[optional] transformer(s)* – for processing the tokens.
- *[optional] sink* – for displaying or storing the processed tokens.

The tool for creating – and also running – flows is the *Flow editor*. Figure 2.1 shows the default view of the editor when starting it up. Actors can be added to a flow by either dragging them from the *Actors* tab on the right-hand side onto the flow or by using the right-click menu of the left-hand side pane.

The *Search* box of the *Actors* tab on the right-hand side allows to search in the actor names and their description.



Figure 2.1: Flow editor with an empty new flow (*File* → *New* → *Flow*)

You can edit as many flows in parallel as you want, e.g., for copy/pasting actors or other setups between them. Also, you can run them in parallel as well.

### 2.2.1 Hello World

The first flow <sup>1</sup> that we will be setting up now is very simple: a *source* will output the string *Hello World!* and a *sink* will display it then. For simplicity, we will just use the right-click menu for adding the actors to this flow.

Since this is our first flow, we want the display to be as verbose as possible. Hence make sure to check *Show input/output* from the *View* menu. This will display what types of inputs and outputs the various actors accept and/or generate. Once you are familiar with the actors, you might want to turn this feature off again, especially when the flows become larger.

First, we need to add the *source* that outputs the string. The *StringConstants* source can output an arbitrary number of strings that the user defined. We will use this actor in this simple example.

Select the actor that you want to add an actor before, after or below. In this case, starting with an empty flow, this is the *Flow* actor. Now right-click and select *Add beneath...* from the menu, as shown in Figure 2.2.



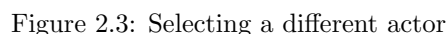
Figure 2.2: Popup menu for adding a new actor


ADAMS tries to suggest an actor depending on the context where the actor will get placed. But due to the large amount of actors, most of the time you will choose a different one. You can do this by simply clicking on the button – showing an icon of a hierarchical structure – in the top-right corner of the current dialog. A new popup will be displayed right next to the button (see Figure 2.3).

Due to the large amount of available actors <sup>2</sup>, most of the time it is quicker


<sup>1</sup>adams-core-hello\_world1.flow

<sup>2</sup>ADAMS automatically filters actors that won't fit where you want to place a new actor. Using the *strict mode*, you can also filter the actors that might only be compatible, like general purpose ones. Each module can define such rules. Also, the initial actors that ADAMS suggests are based on pre-defined rules of what actors are commonly placed in certain situations. If there are more than one suggestions, a combobox with all the class names is displayed in



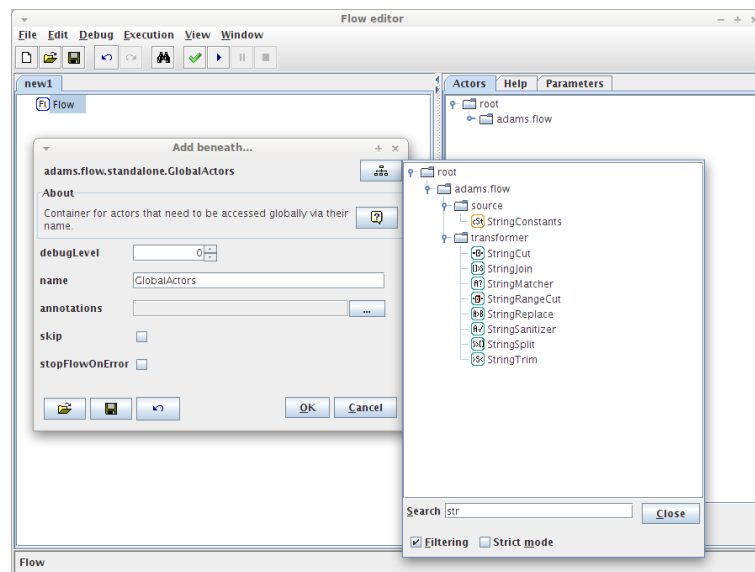
Now click on the *StringConstants* actor to select it. The *About* description only displays some of the information about an actor (normally only the first sentence of the general description). If you want to know more about an actor and its options, just click on on the  button in the *About* box. For the *StringConstants* actor this opens a dialog like shown in Figure 2.5. For quick info on options, you simply hover with your mouse over one of the options and a tool tip will come up with the description.

The Flow editor offers help also through the tabs on the right hand side. The *Help* tab (Figure 2.6) displays the same information as the aforementioned dialog, but without the need of opening a new dialog. You merely have to select an actor on the left hand side in order to display its help screen. The *Parameters* tab is a shortcut to see all the options of the currently selected actor which differ from the default options (2.7). This can be quite handy when quickly going through multiple actors, checking their values. Especially actors with lots of options.

So far, you have only configured an object (a simple string in this case). Now you have to add the string object to the list, in order to use it. Just click on the  button on the right-hand side (a *red* plus sign indicates a recent

---

the `GenericObjectEditor` instead of a simple label.

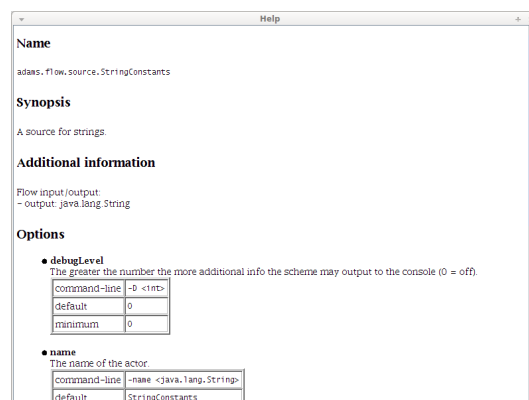
Figure 2.4: Searching for *StringConstants* actor

change, *green* indicates that nothing has changed). If you wanted to output more than just one string, for each of them you would bring up the dialog again, enter the value and add it to the list. After adding all the necessary items, confirm the dialog by clicking on the *OK* button.

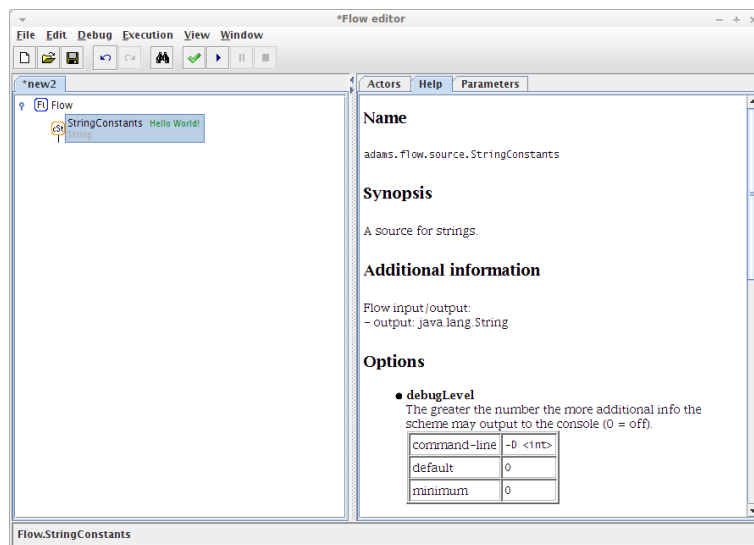
This finishes our set up of the *StringConstants* actor and you can confirm the dialog with *OK* button. Figure 2.9 shows the resulting flow.

For our simple *Hello World* example, we don't need an data processing using *transformer* actors, only a *sink* that will display our data. The *Display* actor can be used for displaying textual data. This actor adds the string representation of each token that it receives as a new line in its text area.

For adding the *Display* actor, right-click on the previously added *StringConstants* actor and select *Add after...* as shown in Figure 2.10.

Figure 2.5: Help dialog for the *StringConstants* actor



Figure 2.6: Tab displaying help for the *StringConstants* actor

Once again, bring up the class tree dialog with all the actors by clicking on the button in the top-right corner of the actor dialog. This time, you have to search for *Display*. As soon as you have entered *dis* you will see the dialog showing a filtered class tree as shown in Figure 2.11.

Select the *Display* actor, just like you did with the *StringConstants* actor. Since we don't have to configure anything for this actor – it merely displays our data – you can just confirm it by clicking on *OK* again.

This completes our flow for this simple example and you can save the set up. The final flow is shown in Figure 2.12.

With the flow finished, we can now execute it. In the Flow editor menu, select *Execution* → *Run*. Or use the keyboard shortcut **Ctrl+R**. Figure 2.13 shows the result output.

Well done, your first flow is set up and produces output!

Figure 2.7: Tab displaying the non-default options for the *StringConstants* actorFigure 2.8: Adding the *Hello World!* string

Figure 2.9: Flow after adding the *StringConstants* actorFigure 2.10: Adding another actor *after* the current one

Figure 2.11: Searching for the *Display* actorFigure 2.12: The complete *Hello World* flow



Figure 2.13: The output of *Hello World* flow

### 2.2.2 Processing data

Of course, for simply outputting some string, you don't need a workflow engine. The idea of a workflow is to be able to define all steps for processing the data, not just simply loading and displaying it.

The following steps extend our flow <sup>3</sup> with some string processing: first, turning the initial string into upper case and, second, appending some text at the end.

Basic string processing can be performed with the *Convert* transformer. This actor allows you to choose a conversion class that performs the actual transformation.

Since our flow only consists of a source and a sink, we need to insert the transformer in between the two of them. In our example we right-click on the sink actor and then choose *Add here...*, as you can see in Figure 2.14. But you can also right-click on the source and then choose *Add after...*.

The *Add here...* action always moves the actor on which you clicked one further down and adds the chosen one at the current position. The *Add after...*, adds the chosen actor after the one that you clicked on.



Figure 2.14: Adding an additional actor

Now choose the *Convert* transformer from the class tree, e.g., by searching for it, as displayed in Figure 2.15.

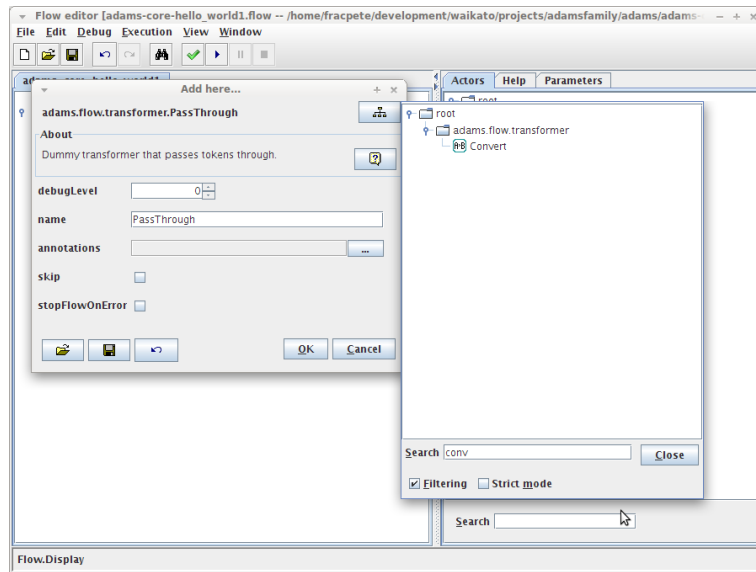
Change the type of *conversion* to *UpperCase* (Figure 2.16).

The flow should now look like Figure 2.17 and, when you execute it, produce output as shown in Figure 2.18. This concludes our first string processing step.

The second string processing step <sup>4</sup> requires adding a custom string at the end of the actor outputting *HELLO WORLD!*. We can achieve this by using the *StringReplace* actor, which allows us to perform string replacements using

<sup>3</sup>adams-core-hello-world2.flow

<sup>4</sup>adams-core-hello-world3.flow

Figure 2.15: Adding the *Convert* transformer

regular expressions<sup>5</sup>. In this case, the replacement is very simple: replacing the end of the string (“\$”) with the string that we want to append “How are you today!” (see Figure 2.19).

Executing the flow now will produce output as seen in Figure 2.20.

<sup>5</sup>For more information see [http://en.wikipedia.org/wiki/Regular\\_expressions](http://en.wikipedia.org/wiki/Regular_expressions).

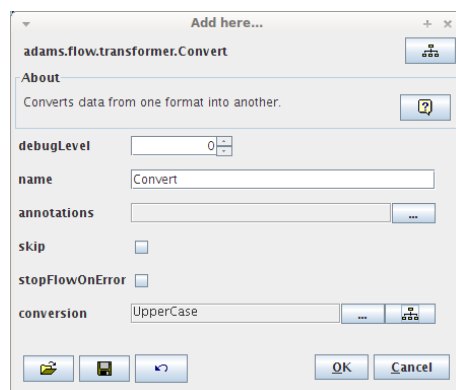
Figure 2.16: Configuring the *Convert* transformer

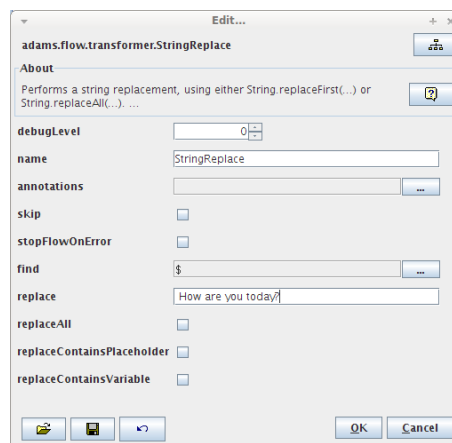
Figure 2.17: Extended *Hello World* flowFigure 2.18: Output of the extended *Hello World* flowFigure 2.19: Customizing the *StringReplace* transformer





Figure 2.20: Output of further extended *Hello World* flow

### 2.2.3 Control actors

So far, we have only covered linear execution of actors, where one actor is executed after the other. For this linear approach, a workflow still seems like overkill. In the following sections we will introduce *control actors*, which *control* the flow of data within the flow in some way or another.

#### 2.2.3.1 Have some Tee

The *Tee* actor, like the Unix/Linux *tee* command, allows you to fork off the data that is being passed through and re-use it for something else. For example for debugging purposes, when you need to investigate the data generation at various stages throughout the flow.

In the following example <sup>6</sup> we will use the *Tee* actor to document the various stages of transformation that the *Hello World!* string goes through. Three *Tee* actors will be placed in the flow: one right after the *StringConstants* source, the next after the *Convert* transformer, and the last after the *StringReplace* transformer. Each time, a *DumpFile* sink will be added beneath the *Tee* actor, pointing to the same log file. In our example, we are using `/tmp/out.txt` - adjust it to fit your system. By default, the *DumpFile* actor overwrites the content of the file if it already exists. This is fine for the first occurrence, but for the second and third one we need to check the *append* option. Otherwise we will lose the previous transformation steps. The fully expanded flow is shown in Figure 2.21. Figure 2.22 shows the generated log file in a text editor.

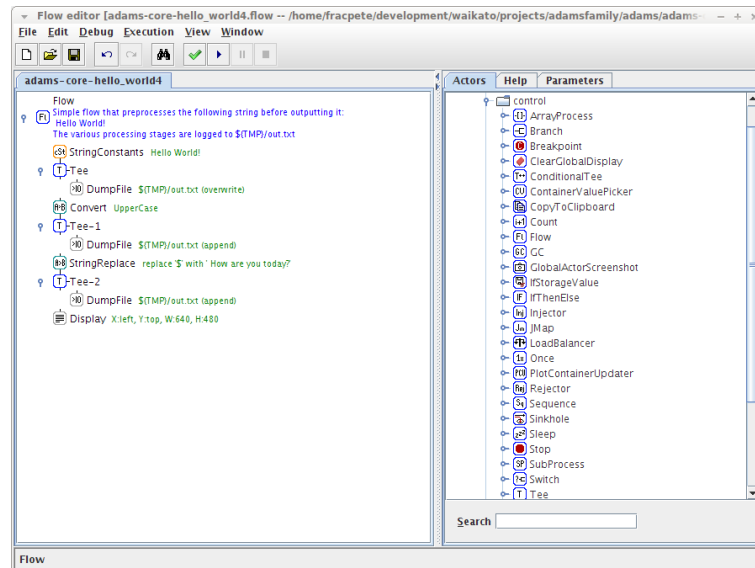


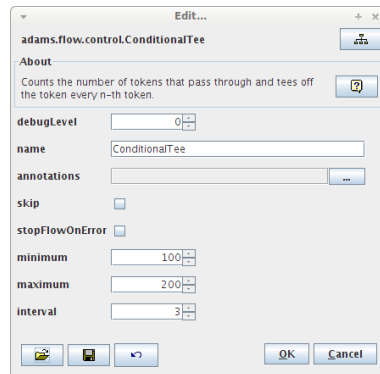
Figure 2.21: The *Hello World* flow with *Tee* actors.

The *ConditionalTee* control actor is an extended version of the simple *Tee* actor. This actor keeps track of the number of data tokens passing through. This allows you to specify rules for when to fork off the data tokens. For instance,

<sup>6</sup>adams-core-hello\_world4.flow

Figure 2.22: The log file generated by the *Tee* actors.

you can configure it that only every third token gets forked off, starting with the 100th one and stopping with the 200th token (to be precise, the first token output is the 102nd and the last one the 198th one). See Figure 2.23 for an example of this set up.

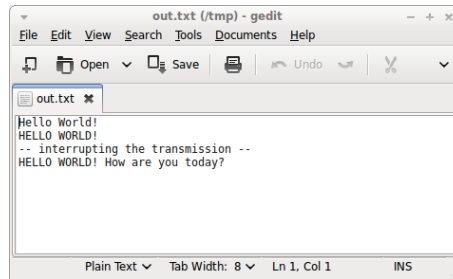
Figure 2.23: A customized *ConditionalTee* actor.

A close cousin to the *ConditionalTee* actor is the *Count* actor. This actor offers the same conditions as the *ConditionalTee* for the tee output, but instead of forking off the current data token, it forks off the number of tokens it has encountered so far. Very useful when trying to keep track of how much data has been processed.

### 2.2.3.2 Pull the *Trigger*

The *Trigger* control actor is used to initiate the execution of a sub-flow. In contrast to the *Tee* actor, the *Trigger* does not fork off any token, it merely triggers the execution of the actors defined below it. Since no data is being forked off, a source actor is required in the sub-flow to kick off the other actors. Using a trigger<sup>7</sup> we can inject another string into the log file that was generated in the previous example, as Figure 2.24. Figure 2.25 shows the modified log file in a text editor. The *Trigger* actor is also the only other control, besides the *Flow* control actor, that allows *standalone* actors to be added to it.

<sup>7</sup>adams-core-hello.world5.flow

Figure 2.24: The *Hello World* flow with an additional *Trigger* actor.Figure 2.25: The modified log file generated with the additional *Trigger* actor.

### 2.2.3.3 Branching – or how to grow your flow

So far, we have only processed data in a sequential way. The *Branch* actor allows the parallel processing of the same token. Each sub-branch receives the same token for further processing. In Figure 2.26, we re-use our simple example, outputting *Hello World* in parallel, displaying the results in two different *Display* actors<sup>8</sup>. The second sub-branch processes the original string further. As you can see from this example, as soon as you have more than one actor, you need to encapsulate the actors in a *Sequence* control actor. The default setting of the *Branch* actor is to process the branches in separate threads, taking the maximum number of cores/CPU's of the underlying architecture into account. But it is also possible to enforce a sequential execution of the sub-branches, by setting the *number of threads* to **0**. There are two reasons for this:

1. *Resources* – If the branch is located deeper in the flow with other parallel execution happening, spawning too many threads can slow down the sys-

<sup>8</sup>adams-core-hello-world6.flow



Figure 2.26: The *Hello World* flow using a *Branch* actor.

tem more than it could help in the optimal case. In such a scenario, it is advised to turn off parallel execution.

2. *Ordering* – In certain cases, the same data needs to be processed several times, but the order of the in which this occurs is important. For instance, an integer token could be used to create a sub-directory in which to store the value of the integer token in a file. These two sub-branches need to get executed one after the other, of course.

#### 2.2.3.4 Further control actors

The *Branch*, *Tee* and *Trigger* control actors are just some of the more commonly used ones. ADAMS comes already with a wide variety of control actors. In the following a short introduction to the others:

- *ArrayProcess* – instead of unraveling an array with *ArrayToSequence* and then packaging again with *SequenceToArray*, this actor allows to perform an arbitrary number of sub-steps on an incoming array.
- *Breakpoint* – Used for debugging a flow. See 2.8 for more details.
- *ClearGlobalDisplay* – Can be used to clear globally defined graphical actors, e.g., a *SequencePlotter*. See 2.3 for more information on global actors.
- *ConditionalTee* – Basically like the *Tee* actor, but it allows you to impose constraints on when to tee off the tokens.
- *ContainerValuePicker* – Since ADAMS only allows *1-to-1* and *1-to-n* connections, multiple outputs are usually packaged in *containers*. The values in the container can be accessed by their name (check the specific actor's documentation on what the names are) using this actor.
- *CopyToClipboard* – Places the string representation of the current token passing through onto the system's clipboard.

- *Count* – In contrast to *ConditionalTee*, this actor tees off the number of tokens it has encountered so far. Useful for lengthy processes, if you want to keep track of how many tokens you have processed so far.
- *GC* – For explicitly executing the Java garbage collection.
- *GlobalActorScreenshot* – For taking screenshots of a globally defined (graphical) actor, whenever a token passes through this control actor. See 2.3 for more information on global actors.
- *IfStorageValue* – An *if-then-else* source that executes the *then* branch if the specified storage value exists. Otherwise it executes the *else* branch, which needs to have a source actor for generating actual data.
- *IfThenElse* – A control statement, which evaluates a boolean condition in order to decide in which branch to pass on the incoming token.
- *Injector* – Allows you to inject tokens into the stream of tokens.
- *JMap* – If available, i.e., using a JDK instead of JRE, you can output information on what objects are currently present in the JVM. Useful for hunting down memory leaks.
- *LoadBalancer* – Spawns off threads for incoming tokens to process the tokens independently in the sub-flow defined below this actor.
- *Once* – A tee actor that only tees off the first token it encounters. A simplified *ConditionalTee* so to speak.
- *PlotContainerUpdater* – Allows one to update the *name*, *x* or *y* value stored in a plot container. Useful for post-processing of plot containers, e.g., for scaling.
- *Rejector* – Rejects tokens container data containers that have error messages attached.
- *Sequence* – Allows to specify multiple actors that get executed one after the other, with the output of one actor being the input of the next.
- *Sinkhole* – Does not pass on tokens if the specified boolean expression evaluates to *true*.
- *Sleep* – Suspends the flow execution for the specified number of milliseconds.
- *Stop* – If executed, stops the flow execution.
- *SubProcess* – Like *Sequence* actor, but the last actor definitely has to produce output, i.e., cannot be a *sink*.
- *ConditionalSubProcess* – Basically like the *SubProcess* actor, but it allows you to impose constraints on when to process the tokens with actors defined in the sub-process.
- *Switch* – Allows an arbitrary number of branches, which get forwarded the token if the corresponding condition evaluates to *true*.
- *UpdateContainerValue* – Applies all defined sub-actors to the specified element of the container that is passing through.
- *UpdateProperties* – Updates multiple properties of an actor wrapping a non-ADAMS object, using current variable values.
- *WhileLoop* – Executes the sub-flow as long as the boolean condition evaluates to *true*.

## 2.3 Global actors

ADAMS uses a tree structure to represent the nested actor structure. This enforces a 1-to-n relationship on how the actors can forward data. In the example flow shown in Figure 2.26, two separate *Display* actors get displayed. The more branches, the more windows will pop up. This gets very confusing rather quickly. ADAMS offers a remedy for this: **global actors**. With this mechanism, multiple data streams can once again be channeled into a single actor again, simulating a n-to-1 relationship. And here is what to do:

- Add the *GlobalActors* standalone actor at the start of the flow.
- Add the actor that you want to channel the data into below the *GlobalActors* actor that you just added. In our example, this is the *Display* actor.
- Replace each occurrence of the actor that you just added below the *GlobalActors* actor with a *GlobalSink* sink actor. Enter as value for the parameter *globalName* the name of the actor that you added below the *GlobalActors* actor. In this example this is simply *Display*.

Figure 2.27 shows the modified flow <sup>9</sup>, using a single global *Display* actor and multiple *GlobalSink* actors. In addition to the *GlobalSink* actor, there are also

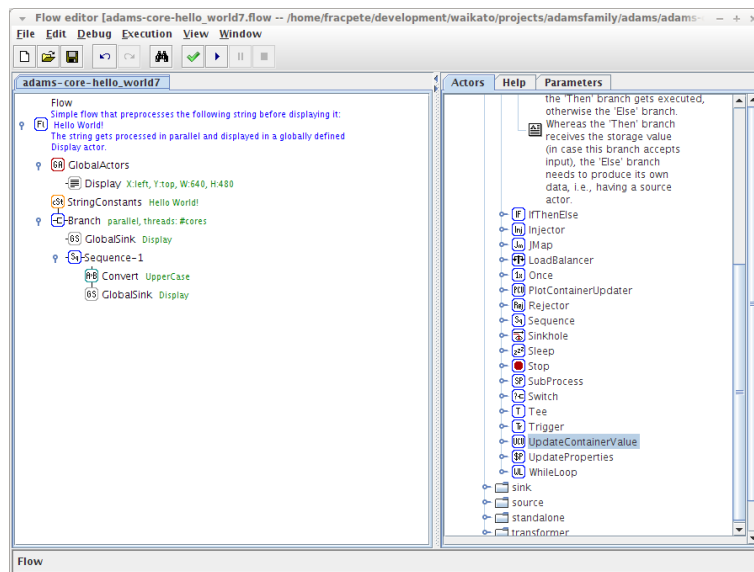


Figure 2.27: Outputting parallel processed strings in a single global *Display* actor.

the *GlobalSource* actor (for using the same source multiple times in a flow) and the *GlobalTransformer* actor (for instance for using the same preprocessing multiple times). They are used in the same fashion as the *GlobalSink* actor that we just introduced.

<sup>9</sup>adams-core-hello.world7.flow

## 2.4 External actors

Flows can quickly become large and complex, with lots of preprocessing happening in multiple locations. Pretty soon you will realize that certain preprocessing steps are always the same. The same applies to loading data (e.g., various benchmark data sets) or writing results back to disk.

To avoid unnecessary duplication of functionality, ADAMS allows you to *externalize* parts of your flow to be externalized, i.e., stored on disk. Externalizing an existing sub-flow is very easy, you merely have to right-click on the actor that you want to save to disk and select *Externalize...* from the popup menu. A new Flow editor window will pop up with the currently selected sub-flow copied into, ready to be saved to disk<sup>10</sup>. Once you saved the sub-flow to a file, you have to go back into the original flow and update the file name of the flow in the external *meta* actor that replaced the sub-flow.

Here are the available meta actors:

- *ExternalStandalone* – for using externalized standalones.
- *ExternalSource* – for incorporating an external source.
- *ExternalTransformer* – for applying an external transformer.
- *ExternalSink* – for processing data in an external sink.

---

<sup>10</sup>Only actors that implement the *InstantiatableActor* interface can be externalized directly. All others need to be enclosed in the appropriate *InstantiatableXYZ* wrapper. Using the *Externalize...* menu item automatically wraps the actor if required,



## 2.5 Templates

ADAMS comes with a powerful templating mechanism, that either speeds up the inception of new flows or dynamic parametrization at runtime. The following two sections explain these two approaches in detail.

### 2.5.1 Static use

The *static use* of templates occurs at design time. Here, templates can speed up the generation of flows by allowing you to insert complete sub-flows that are generated by a template class. Therefore, commonly occurring sub-flows can be encapsulated in a template class with optional parameters. A fairly common sub-flow, encapsulated by a *Trigger* control actor, is the updating of a variable. The *UpdateVariable* template inserts such a sub-flow, consisting of a *Variable* source and a *SetVariable* transformer, enclosed by a *Trigger*. You only need to supply the variable name that needs updating to generate the sub-flow and then add the required transformers that take the current variable value and process is some way or the other. Figures 2.28 to 2.30 show the use of this mechanism.

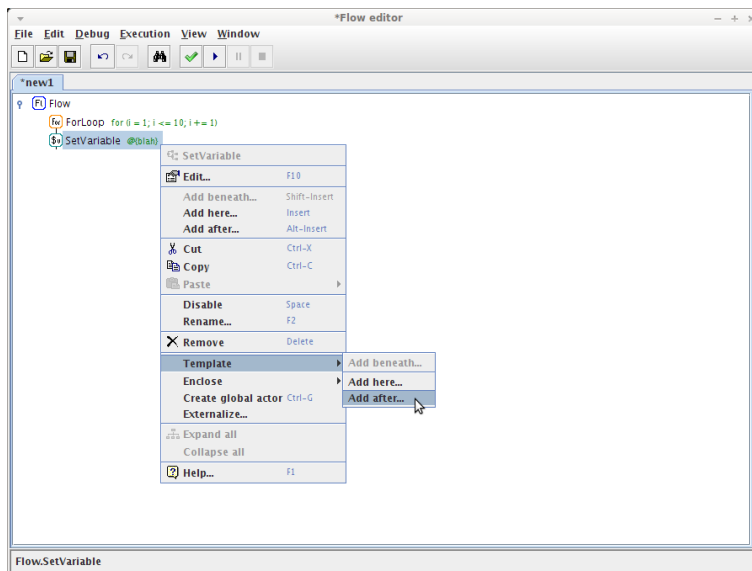


Figure 2.28: Adding a sub-flow generated from a template to an existing flow.

### 2.5.2 Dynamic use

The templating mechanism can be used at runtime as well, using one or more of the following actors:

- *TemplateStandalone* – for templates that generate standalones
- *TemplateSource* – for templates that generate sources
- *TemplateTransformer* – for templates that generate transforming sub-flows
- *TemplateSink* – for templates that generate sinks

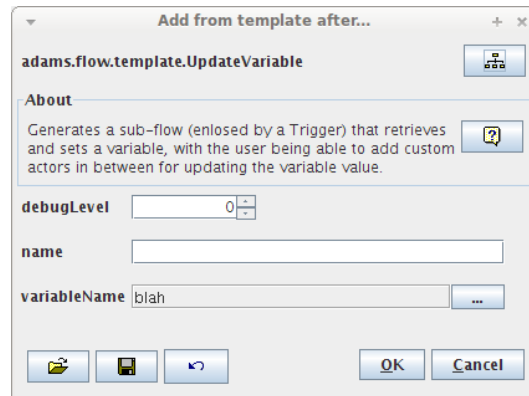


Figure 2.29: The options of the *UpdateVariable* template.

The sub-flow generation is done in a lazy way, i.e., only when the aforementioned template actor is executed, the template is generated. The sub-flow is used till either the end of the flow execution or if a variable changes that is attached to the template itself. In the latter case, the sub-flow gets re-generated the next time the template actor gets executed. This dynamic sub-flow generation in conjunction with variable use, allows to adapt and change the flow at runtime. The example *adams-core-template.flow* demonstrates this.

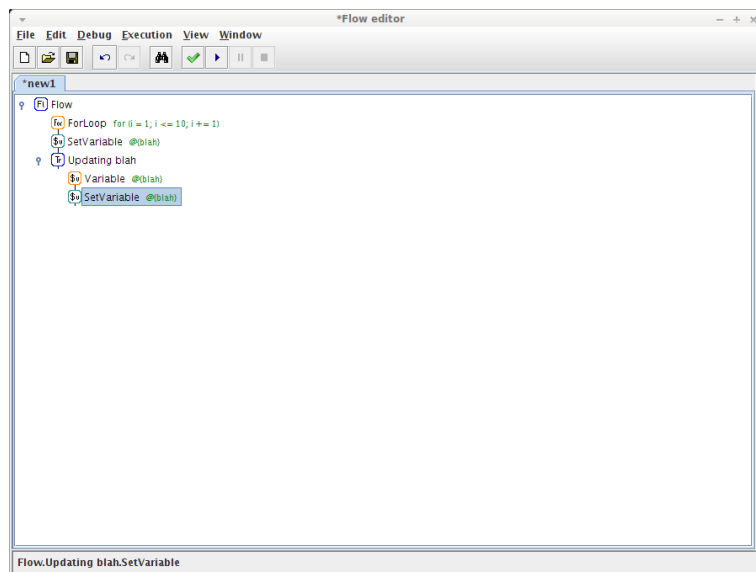


Figure 2.30: The added sub-flow.

## 2.6 Variables

A flow is very useful for documenting all the steps involved in loading, processing and evaluating of data. But setting up a new flow, whenever you are merely varying a parameter is not very efficient. In order to make flows more flexible and dynamic, ADAMS offers the concept of *variables*. The idea of variables is to *attach* them to options of the object that you want to vary. Actors keep track of what variables have been attached to themselves or nested objects. Whenever an actor gets executed, it checks first whether any of the variables that it is monitoring has been modified. If that is the case, the actor re-initializes itself before the execution takes place. This guarantees that the correct set up has been applied. At the time of writing, the scope of variables is restricted to *Flow* actors. Running the same flow in two concurrent Flow editor windows does not result in those two flows interfering with each other.

In the following example <sup>11</sup>, we are using a *ForLoop* source to generate the index of a file to load. In a *Tee* actor we first convert the integer token to a string using the *Convert* transformer and the *AnyToString* conversion scheme. Then we add, first the path and then the file extension, to generate the full file name using *StringReplace* transformers. Finally, we associate the generated file name with the variable *filename* using the *SetVariable* transformer.

In order to display the content of the files, we need to set up a sub-flow that consists of a *SingleFileSupplier* source, the *TextFileReader* transformer for reading in the content and a *HistoryDisplay* sink for displaying the file contents. The sub-flow gets enclosed by a *Trigger* control actor, which will get executed whenever an integer token from the *ForLoop* passes through.

To make use of the variable *filename*, we need to attach it to the *file* option of the *SingleFileSupplier*. You can attach a variable by simply bringing up the

<sup>11</sup>adams-core-variables1.flow

properties editor of an actor (or other ADAMS object), right-click on the name of the option and then entering the name of the variable (without “@{” and “}”). The properties editor indicates whether a variable has been attached to an option by appending an asterisk (“\*”) to the name of the option, as can be seen in Figure 2.31.

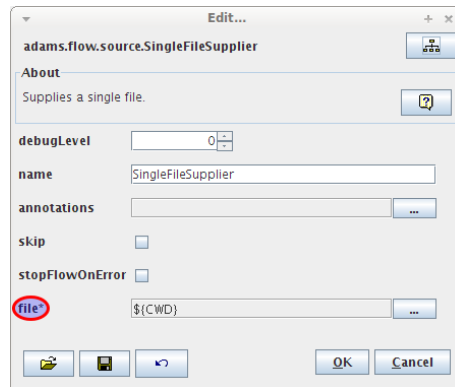


Figure 2.31: The asterisk (“\*”) next to an option indicates that a variable is attached.

The complete flow is displayed in Figure 2.32. With “quick info” enabled, the *SingleFileSupplier* now also hints that the file it is forwarding is variable-based: *@{filename}*.

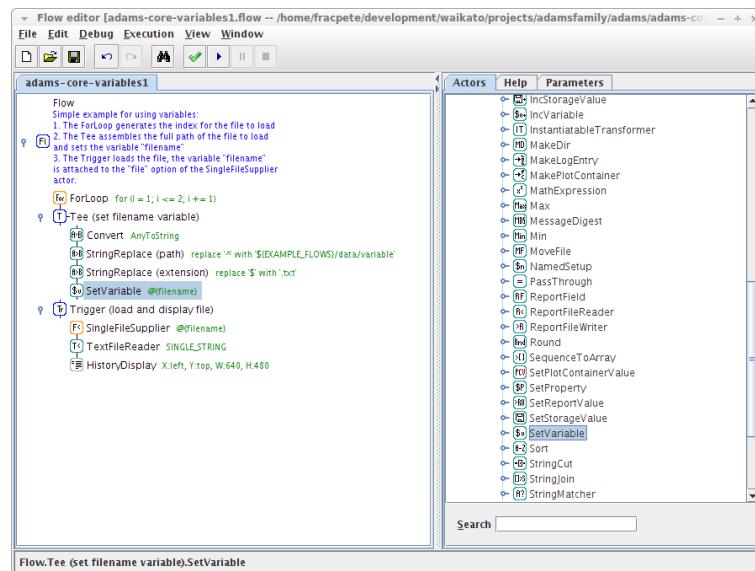


Figure 2.32: Using a variable to control what file to load and display.

The variable mechanism can also be used to dynamically execute another external actor at runtime (see section 2.4 on external actors).

In Figure 2.33 you can see a flow that uses a *ForLoop* to execute three

external flows using a variable attached to the *actorFile* option.

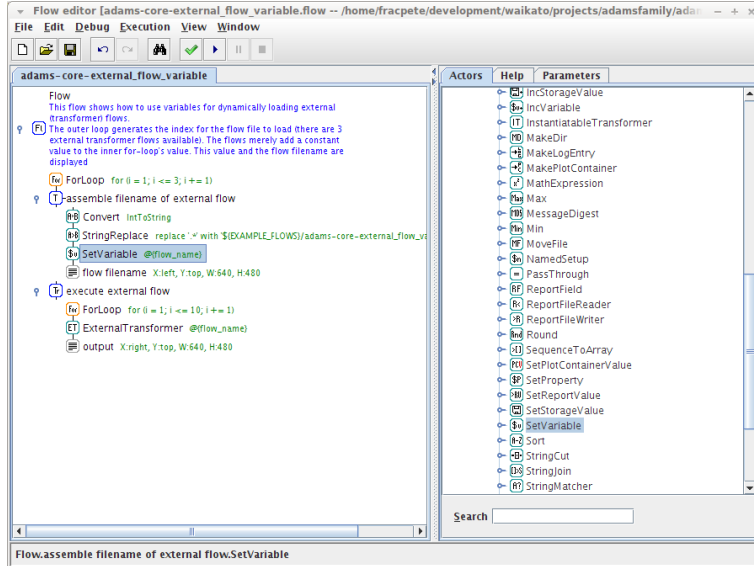


Figure 2.33: Using a variable to control what external flow to execute (flow).

The output generated by the three sub-flows is shown on screen in the same *Display* actor. A screenshot of the output is displayed in Figure 2.34.

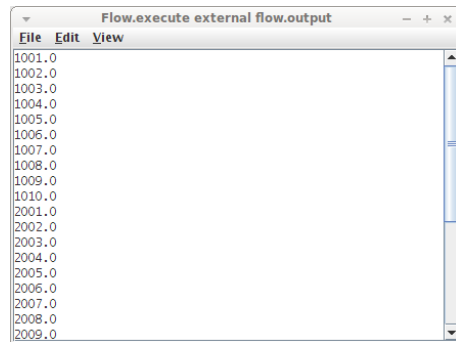


Figure 2.34: Using a variable to control what external flow to execute (output).

### Overview of actors

The following actors are available to handle variables:

- *Variable* – source actor for outputting the value associated with the variable.
- *SetVariable* – updates the value of a variable (transformer).
- *IncVariable* – increments the value of the variable by either an integer or double increment (transformer).
- *DeleteVariable* – removes a variable and its associated value from internal memory (transformer).

**Non-ADAMS objects**

The Variable functionality is only available for objects within the ADAMS framework, as it requires special option handling. 3rd-party libraries do not benefit from this functionality directly. But thanks to Java Introspection<sup>12</sup> you can use *property paths* to access nested properties and update their values. A property path is simply the names of the various properties concatenated and separated by dots (“.”). In case of arrays, you simply have to append “[x]” to the property with “x” being the 0-based index of the array element that you want to access.

The following actors allow the updating of properties:

- *SetProperty* – transformer that modifies a single property of a global actor based on the current value of the specified variable.
- *UpdateProperties* – allows you to update multiple properties (each property is associated with a particular variable) of the actor that this actor manages.

---

<sup>12</sup>See <http://download.oracle.com/javase/tutorial/javabeans/introspection/> for more information on Java Introspection.

## 2.7 Temporary storage

Variable handling within ADAMS is a very convenient way of changing parameters on-the-fly, but it comes at a cost. Values for variables are merely stored as strings internally and each time an options gets updated this string needs to get parsed and interpreted. Furthermore, each time the whole actor gets reinitialized if one its own options or an option of its dependent objects gets updated. It is strongly advised against using the variables functionality if they are not actually attached to any options, but only used for keeping track of values like loop variables.

Instead, ADAMS offers an alternative framework for managing values at runtime: *temporary storage*. In contrast to variables, values are stored internally as Java objects, referenced by a unique name. Just like with variables, the scope of these objects is restricted to *Flow* actors at the time of writing. Additionally, the values don't need to be parsed again when used, since they are stored as is, resulting in a more efficient storage/retrieval. Finally, arbitrary objects can be stored, not just objects for which a string representation can be generated/parsed. The latter aspect combined with fast storage/retrieval encourages multiple read/write accesses of the same object in various locations of the flow. An example would be accessing a data set or spreadsheet, retrieving, setting or updating values. Figure 2.35 shows a flow that takes the number generated by the random number generator and stores it, before re-using it in the sub-flow below the *Trigger* actor. The resulting output is displayed in Figure 2.36.

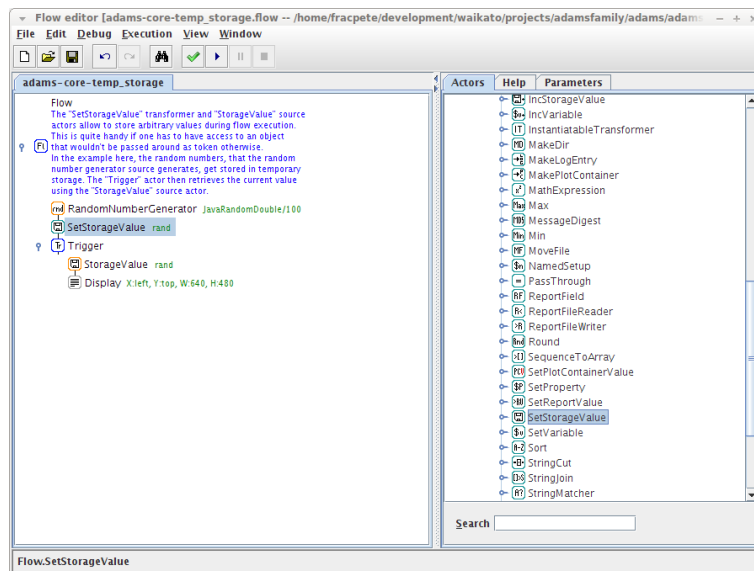


Figure 2.35: Flow demonstrating the temporary storage functionality.

By default, the storage system is unlimited which can quickly result in memory problems when not used wisely. In order to restrict memory usage and encourage re-generation of values on demand, the storage system also offers

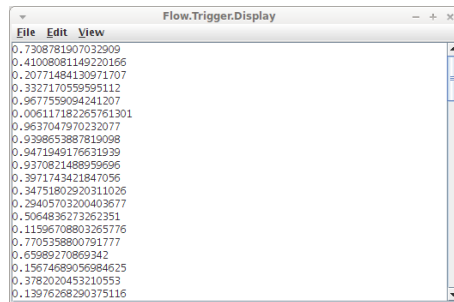


Figure 2.36: Output of flow demonstrating the temporary storage functionality.

least-recently-used (LRU) caches<sup>13</sup>. Instead of simply setting a value with a name, you can specify the name of a particular LRU cache as well. The cache needs to be initialized first, of course, using the *InitStorageCache* standalone actor. Figure 2.37 shows the use of the LRU cache functionality, with Figure 2.38 displaying a snapshot in time of the storage inspection panel available through the *Breakpoint* control actor. Finally, Figure 2.39 shows the final output of the flow.

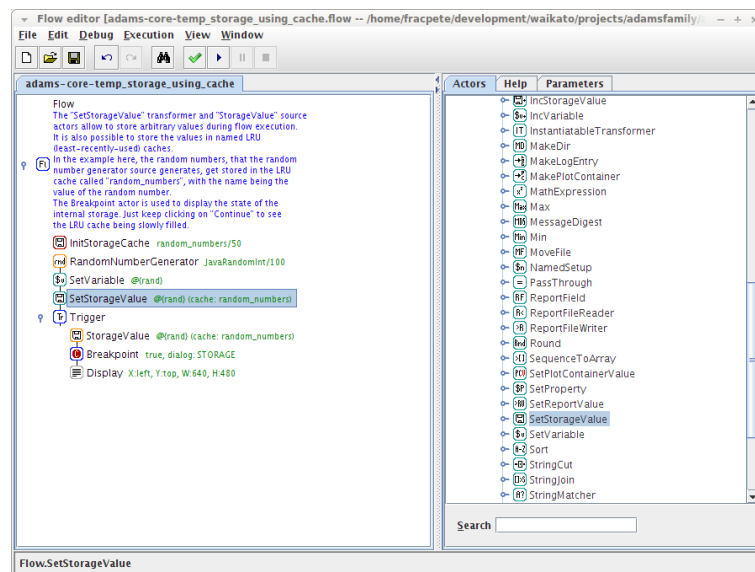


Figure 2.37: Flow demonstrating the LRU cache storage functionality.

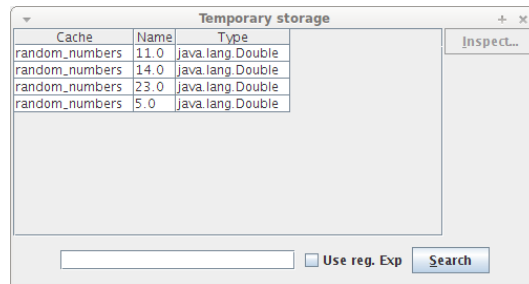
### Overview of actors

The following actors are available to handle variables:

- *InitStorageCache* – standalone actor for initializing a named LRU cache with a specific size.

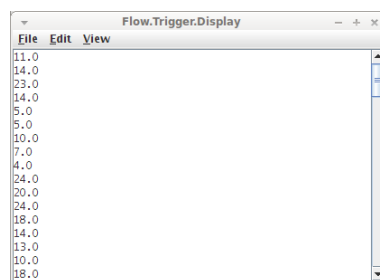
<sup>13</sup>See [http://en.wikipedia.org/wiki/Cache\\_algorithms#Least\\_Recently\\_Used](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used) for more information on LRU caches.





Cache	Name	Type
random_numbers	11.0	java.lang.Double
random_numbers	14.0	java.lang.Double
random_numbers	23.0	java.lang.Double
random_numbers	5.0	java.lang.Double

Figure 2.38: Display of the temporary storage during execution.



File	Edit	View
11.0		
14.0		
23.0		
14.0		
5.0		
5.0		
10.0		
7.0		
4.0		
24.0		
20.0		
24.0		
18.0		
14.0		
13.0		
10.0		
18.0		

Figure 2.39: Output of flow demonstrating the LRU cache storage functionality.

- *StorageValue* – source actor for outputting the storage value associated with the specified name.
- *SetStorageValue* – updates the specified storage value (transformer).
- *IncStorageValue* – increments the value of the stored integer or double object by either an integer or double increment (transformer).
- *DeleteStorageValue* – removes a storage value from internal memory (transformer), freeing up memory.

## 2.8 Debugging your flow

The more complex a flow gets, the harder it becomes to track down problems. With all its general purpose actors and control actors (loops, switch, if-then-else, ...), ADAMS is basically a basic graphical programming language. A programming language without at least some basic debugging support is very inconvenient. Therefore, ADAMS allows you to set *breakpoints* in your flow. These breakpoints are merely instances of the *Breakpoint* control actor. This actor allows you to specify a breakpoint condition on when to stop. The default condition is *true*, i.e., the execution gets paused whenever the actor gets reached. This boolean condition can also evaluate the value of variables. Just surround the name of the variable with “@{” and “}” in order to use its value within the expression. For more information on what the expression can comprise of, check the online help of the *Breakpoint* actor.

Whenever the *Breakpoint* actor is reached and the condition evaluates to *true*, the control panel of the actor will get displayed (see Figure 2.40).

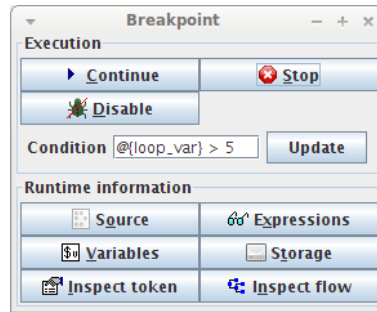


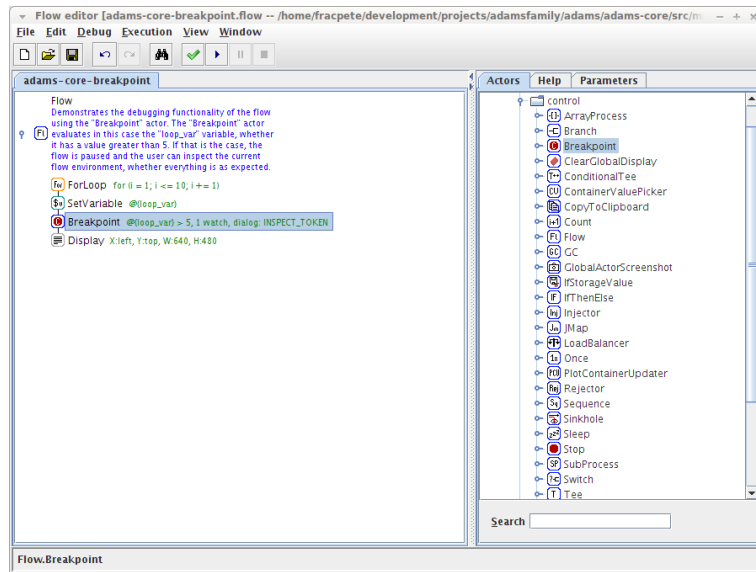
Figure 2.40: The control panel of the *Breakpoint* actor.

The functionality of the control panel is best explained with an example. The flow <sup>14</sup> in Figure 2.41 simply outputs integer values, ranging from 1 to 10. This value gets stored in the variable *loop\_var* which is also part of the condition of the *Breakpoint* actor. Finally, these values get displayed in a *Display* sink.

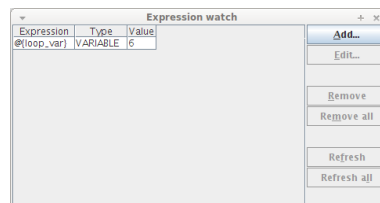
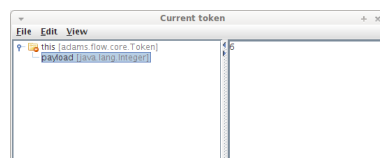
When the breakpoint gets triggered, the flow gets paused and the aforementioned control panel is displayed.

- The buttons in the *Execution* group allow you to continue with the flow execution, you can stop the flow or you can simply disable the breakpoint (which resumes the execution immediately).
- It is possible to update the breakpoint condition whenever the breakpoint is reached, by simply changing the condition string and clicking on the *Update* button.
- In the *Runtime information* group you can view the source code of the fully expanded flow (i.e., all external actors are inserted completely and variables are expanded to their current value), you can define watch expressions (variables, boolean and numeric expressions; Figure 2.42), display an overview of all the variables and their current values, inspect the current storage items, you can inspect the current token that is being

<sup>14</sup>adams-core-breakpoint.flow

Figure 2.41: Example flow with *Breakpoint* actor.

passed through the breakpoint (Figure 2.43) and also inspect the current flow object (Figure 2.44).

Figure 2.42: The *Expression watch* dialog of the *Breakpoint* actor.Figure 2.43: The *Inspection* dialog of the *Breakpoint* actor for the current token.

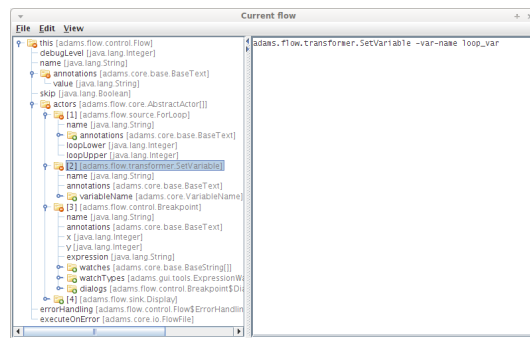


Figure 2.44: The *Inspection* dialog of the *Breakpoint* actor for the current flow.

## Chapter 3

# Visualization

Visualization is very important in data analysis. The core module of ADAMS comes with some basic support.

- **Image viewer** – For displaying images of type PNG, JPEG, BMP, GIF.
- **Preview browser** – Generic preview browser, any ADAMS module can register new preview handlers for various file types.

### 3.1 Image viewer

The Image viewer is a basic viewer for graphic files (PNG, JPEG, BMP, GIF). Figure 3.1 shows the viewer with a single image loaded. It is possible to copy images to the system's clipboard, export or save them in a different file format or print them.

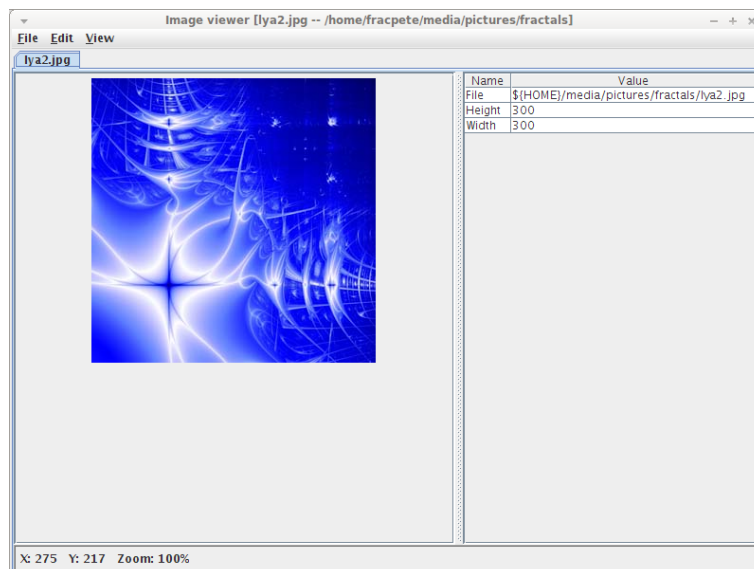


Figure 3.1: Displaying a fractal in the Image viewer.

## 3.2 Preview browser

The preview browser is a generic preview framework within in ADAMS and each module can register new handlers for various file or archive types. In its basic functionality, the preview browser can view images (see 3.2), properties files, flows (see 3.3) and plain text files (see 3.4). If no handler is registered for a file type, i.e., a certain file extension, then the plain text handler is used by default. If more than one handler is registered for a file type, then you can select from the combobox at the bottom of the dialog, which handler is the preferred for this type of file.

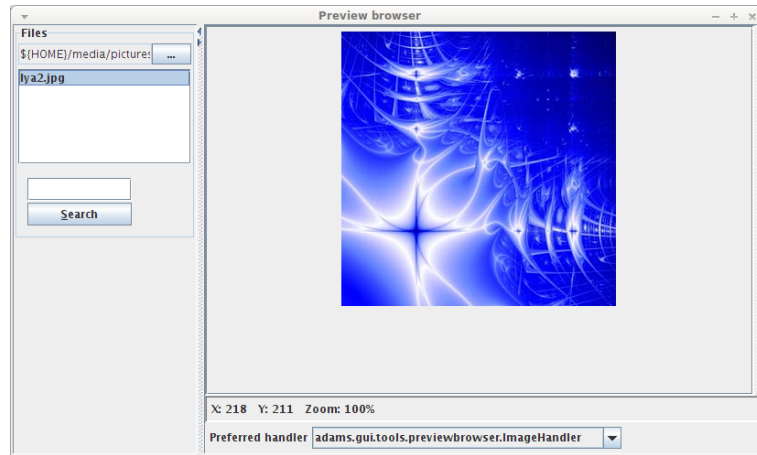


Figure 3.2: Preview browser displaying an image.

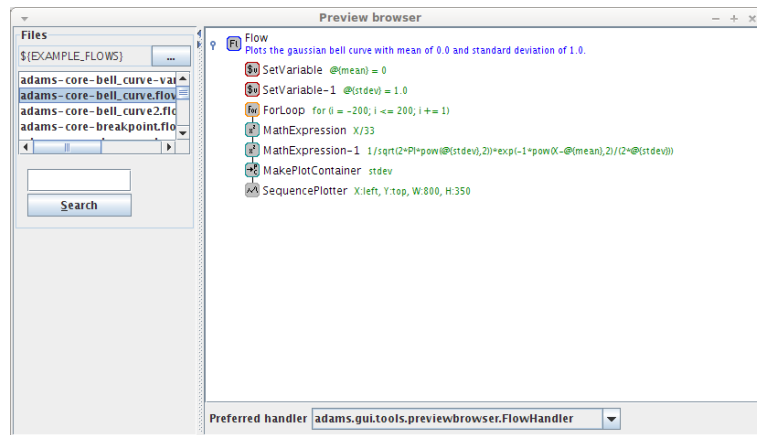


Figure 3.3: Preview browser displaying a flow.

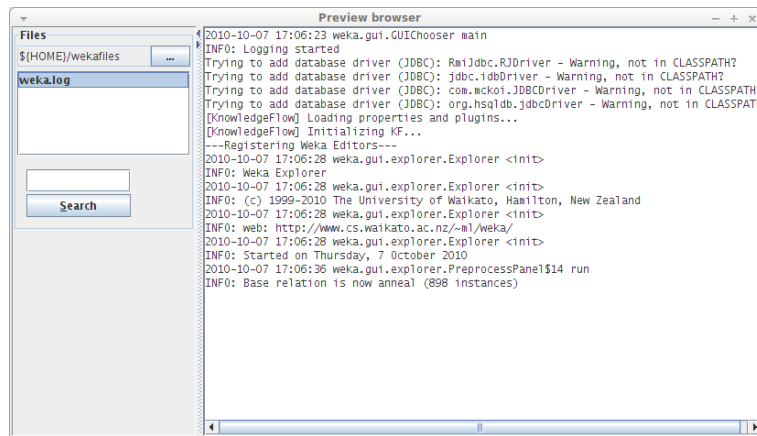


Figure 3.4: Preview browser displaying an plain text file.





## Chapter 4

# Tools

### 4.1 Flow editor

The Flow editor is the central tool in ADAMS, allowing you the definition of powerful workflows for a multitude of purposes. See chapter 2 for a comprehensive introduction.

### 4.2 Flow runner

TODO

### 4.3 Flow control center

TODO

### 4.4 Text editor

TODO



## Chapter 5

# Maintenance

The *Maintenance* menu is only available, if the application has been started as a user labeled as *expert* or *developer*. By default, the user is assumed to be a *basic* user, not needing the more advanced features, requiring more care and consideration. If access to maintenance tools is required, you can add the following to the command-line for starting up ADAMS:

```
-user-mode EXPERT
```

or

```
-user-mode DEVELOPER
```

### 5.1 Placeholder management

TODO

### 5.2 Named setup management

TODO

### 5.3 Variable management

TODO

### 5.4 Favorites management

TODO



## Chapter 6

# Customizing ADAMS

TODO

### 6.1 Main menu

TODO

### 6.2 Properties files

TODO



## Part II

# Developing with ADAMS





# Chapter 7

## Tools

ADAMS, like any other complex project, is using a revision control system to keep track of changes in the code and a build system to turn the source code into executable code.

The following sections cover the various tools and environments that are used when developing for/with ADAMS.

### 7.1 Subversion

The revision control system that ADAMS uses as backend is Apache Subversion [4]. The ADAMS repository is accessible via the following URL:

`https://svn.scms.waikato.ac.nz/svn/adams/`

You can check out the code in the console using the following command, provided you have subversion command-line tools installed:

```
svn checkout https://svn.scms.waikato.ac.nz/svn/adams/trunk adams
```

There are lots of graphical clients for subversion available, open-source and closed-source ones alike. A good overview is accesible through Wikipedia, *Comparison of Subversion clients*:

`http://en.wikipedia.org/wiki/Comparison\_of\_Subversion\_clients`

### 7.2 Maven

ADAMS was designed to be a modular framework, but not only *multi-module* but *multi-project* and each of the projects consisting of multiple modules. In order to manage such a complex setup, a build system that can handle all this was necessary. Apache Maven [5] fits the bill quite well, coming with a huge variety of available plug-ins that perform many of the tasks that are necessary for build management, e.g., generating binary and source code archives, automatic generation of documentation.

#### 7.2.1 Nexus repository manager

By default, maven merely uses a remote site that one copies archives via `scp` or `sftp`. This approach does not offer a fine-grained access control, you either

have access or you don't. Also, if you are deploying snapshots on a constant basis, these will start to clutter your server hosting the archives, since none of them will ever get removed - even if they are completely obsolete. For better management of the maven repository, Sonatype's Nexus repository manager [6] is used.

In addition to hosting the ADAMS artifacts, Nexus also functions as a proxy to common maven repositories like Maven Central, JBoss Public, java.net, Codehaus, Apache and Google Code.

The manager instance for ADAMS is accessible under the following URL: <http://adams.cms.waikato.ac.nz:8081/nexus/>

### 7.2.2 Configuring Maven

In order to gain access to the repositories hosted by the Nexus repository manager, maven needs to be configured properly. The following steps guide you through the process:

#### Create maven home directory

First, you need to create maven's home directory, if it doesn't exist already. The directory is usually located in your home directory and is called `.m2`. The full path, on \*nix systems, is as follows:

```
$HOME/.m2
```

#### Set up authentication

In your maven home directory, you have to configure the following two files:

- `settings-security.xml` – stores the master password
- `settings.xml` – stores user/password credentials and URLs for the ADAMS repositories.

Next, download the templates of these files from the following web pages and place them in your maven home directory:

- <http://adams.cms.waikato.ac.nz/pg/file/fracpete/read/156/maven-settingssecurityxml>
- <http://adams.cms.waikato.ac.nz/pg/file/fracpete/read/157/maven-settingsxml>

The *Password Encryption* page <sup>1</sup> on the maven homepage explains how to generate the master password and the passwords for the repositories using the maven command-line tools. Since you will be entering your password in plain text in the console, it is recommended that you turn off command history temporarily. For \*nix systems that use *bash* as their interpreter, see the following link as how to disable the history:

<http://forums.fedoraforum.org/showpost.php?p=321365&postcount=2>

In each of the template files that you downloaded, you have to replace the *PASSWORD* string with the actual encrypted password that was generated, including the curly brackets. In the *settings.xml* template, you also have to replace the *USERNAME* string with your user name that you use for accessing the Nexus repository manager.

**NB:** replacements have to be performed case-sensitive, otherwise you will corrupt the tags in the XML file.

<sup>1</sup><http://maven.apache.org/guides/mini/guide-encryption.html>

### 7.2.3 Common commands

Here are a few common maven commands, if you obtained ADAMS from subversion:

- Removing all previously generated output:  
`mvn clean`
- Compiling the code:  
`mvn compile`
- Executing the junit tests:  
`mvn test`
- Executing a specific junit test:  
`mvn test -Dtest=<class.name.of.test>`
- Packaging up everything:  
`mvn package`
- Installing the ADAMS jars in your local maven repository (that will also run the tests):  
`mvn install`
- You can skip the junit test execution (when packaging or installing) by adding the following option to the maven command-line:  
`-DskipTests=true`

### 7.2.4 3rd-party libraries

Sometimes, required libraries are not available through public maven repositories (<http://mvnrepository.com/> is a good place to search for them).

The following sections explain how to install new libraries, either in your local Maven repository (does not require write access to the Nexus repository manager) or uploading them to the Nexus repository manager.

#### 7.2.4.1 Installing locally

Installing a library in your local Maven repository is very simple, using the Maven *install* plugin.

The following example installs version *1.2.3* of the *funky-lib.jar* in the group *adams.thirdparty.misc* and as artifact *funky-lib* under Linux:

```
mvn install:install-file \  
-DgroupId=adams.thirdparty.misc \  
-DartifactId=funky-lib \  
-Dversion=1.2.3 \  
-Dpackaging=jar \  
-Dfile=/some/where/funky-lib.jar
```

And the same under Windows:

```
mvn install:install-file \  
-DgroupId=adams.thirdparty.misc ^  
-DartifactId=funky-lib ^  
-Dversion=1.2.3 ^  
-Dpackaging=jar ^  
-Dfile=C:\some\where\funky-lib.jar
```

You can then reference this library in your *pom.xml* file as follows:

```
<dependency>
  <groupId>adams.thirdparty.misc</groupId>
  <artifactId>funky-lib</artifactId>
  <version>1.2.3</version>
</dependency>
```

#### 7.2.4.2 Uploading to Nexus

If your Nexus user has the appropriate rights, you can easily upload 3rd-party libraries to Nexus and make them available to ADAMS. Maven allows the deployment of files with the following command:

```
mvn deploy:deploy-file <parameters>
```

But in order to keep track on the libraries being added to Nexus - and being able to restore them in case of a Nexus problem - this command is not issued directly by users. Instead the *bash* script **deploy.sh** in the sub-directory *adams-thirdparty* is used. This script and the associated libraries are kept under subversion control to keep track of them.

#### Updating a library

The most common case will be updating an existing library to a newer version. This is the easiest and involves the following steps:

1. Overwrite existing library with the new version in the *adams-thirdparty* directory.
2. Update the version in the **deploy.sh** script for the library (the **-Dversion=...** parameter).
3. Execute the **deploy.sh** script (don't worry about error messages for other libraries, only new libraries or updated versions will succeed).
4. Commit the changes in *adams-thirdparty* to subversion.

Now you can update the dependency in the appropriate *pom.xml* files.

#### Adding a library

Adding a new library involves a little bit more work than merely updating an existing one. Here are the necessary steps:

1. Copy the new library in the *adams-thirdparty* directory.
2. Either find an existing group (e.g., *adams.thirdparty.graphics*) that the new library fits in, or create a new group by adding a block at the end, starting with the **GROUP=adams.thirdparty.yourgroup** statement.
3. Copy another **mvn deploy:deploy-file ...** block and update the following parameters:
  - **-DartifactId=...**
  - **-Dversion=...**
  - **-Dfile=...**

- `-DgeneratePom.description=...`
4. Execute the `deploy.sh` script (don't worry about error messages for other libraries, only new libraries or updated versions will succeed).
  5. Put the new library under subversion control and commit the changes in *adams-thirdparty*.

Now you can add the dependency in the appropriate `pom.xml` files.

## 7.3 Eclipse

The choice of integrated development environment (IDE) is Eclipse [7]. It is not only a very good IDE for Java development, but also for maven and LaTeX - provided you install the proper plug-ins.

### 7.3.1 Plug-ins

In order to get the most out of developing with Eclipse, it is recommended to install the following plug-ins:

- m2eclipse – adds proper maven support  
<http://m2eclipse.sonatype.org/>
- texlipse – turns Eclipse into a type-setting environment with syntax highlighting, previewing, etc. This allows you to program and document with the same application.  
<http://texlipse.sourceforge.net/>

### 7.3.2 Setting up ADAMS

After installing the recommended plug-ins, you can proceed to import the ADAMS source code that you checked out earlier using subversion. Importing maven projects is extremely easy:

- right-click in the *Navigator* or *Project Explorer* and select *Import...*
- select *Maven* → *Existing Maven projects*
- choose the top-level directory of your ADAMS source code tree (the one that contains all the modules and the system-wide *pom.xml*)
- select all the projects that you want to work with and hit *Finish*

For projects that have LaTeX documentation, you have to make sure that the texlipse plugin is configured correctly, otherwise you might end up losing files. Figure 7.1 shows an example setup for the manual of the *adams-core* module. This module has the *adams-core-manual* sub-directory below the *latex* directory, with a LaTeX file of the same name, i.e., *adams-core-manual.tex*. This LaTeX file is listed as the main TeX file in the setup. Since the documentation is generated using *pdflatex*, the output format is *pdf* and the build command *pdflatex*. It is very important not to place any temporary files in the source directory, as they might get deleted during an Eclipse *clean project* operation. Instead, the output directory should be *target/latex/<documentation sub-dir>* (e.g., *target/latex/adams-core-manual*), and the output file *target/latex/<documentation sub-dir>/<documentation sub-dir>.pdf* (e.g., *target/latex/adams-core-manual/adams-core-manual.pdf*).

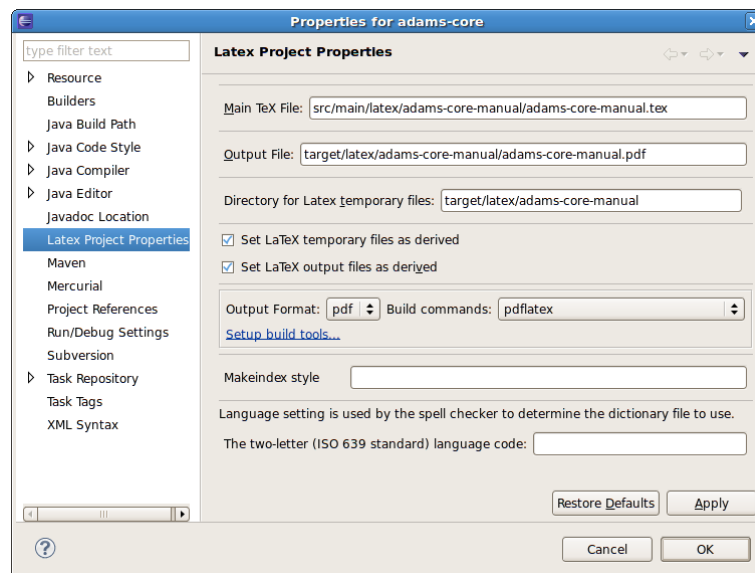


Figure 7.1: texlipse configuration for the *adams-core* module.

## Chapter 8

# Using the API

Using the graphical user interface may be sufficient for most users, but as soon as you want to embed one framework in another, you need to get down and dirty with the API. This chapter addresses some core elements of the ADAMS API, mainly the flow related APIs.

### 8.1 Flow

TODO

#### 8.1.1 Life-cycle of an actor

Any actor, whether a simple one like the Display actor or a control actor like *Branch*, has the following lifecycle of method calls:

- **setUp()** – performs initializations and checks, ensuring that the actor can be executed
- **execute()** – executes the actor, i.e., transformers process the input data and generate output data
- **wrapUp()** – finishes the execution, frees up some memory that was allocated during execution
- **cleanUp()** – removal of graphical output, like dialogs/frame and destruction of internal data structures

The *setUp()* and *execute()* methods return *null* if everything was OK, otherwise the reason (or error message) why the method didn't succeed. The *execute()* method is executed as long as *finished()* returns *false*.

#### OutputProducer

As long as the *hasPendingOutput()* method of an actor implementing *OutputProducer* returns *true*, output tokens will get collected and passed on to the next *InputConsumer*.





## Chapter 9

# Extending ADAMS

The overarching goal of ADAMS was to develop a plug-in framework, which makes extending it very easy. The built-in dynamic class discovery is at the heart of it. The following sections cover various aspects of extending ADAMS, from merely adding a subclass to creating a new project built on top of ADAMS.

### 9.1 Dynamic class discovery

ADAMS is a flexible plug-in framework thanks to the dynamic class discovery that is offered through the `adams.core.ClassLocator` class. But merely locating classes is just half of the story, you also have to organize them. This is where the `adams.core.ClassLister` class and its properties file `ClassLister.props` (located in the `adams.core` package) come into play. The `ClassLister` class iterates through the keys in the properties file, which are names of superclasses, and locates all the derived classes in the listed packages, the comma-separated list which represents the value of the property.

Here is an example for the conversion schemes that can be used with the *Convert* transformer:

```
adams.data.conversion.AbstractConversion=\
adams.data.conversion
```

The superclass in this case is `adams.data.conversion.AbstractConversion` and only one package is listed for exploration, `adams.data.conversion`.

Instead of adding new keys and packages to this central properties file, whenever a new module requires additional class discovery, the developer can just simply add an extra file in their module. The only restriction is that it has to be located in the `adams.core` package.

This works for adding new keys, i.e., new superclasses, as well as for merely adding additional packages to existing superclasses. In the latter case, only the additional packages have to be specified, since ADAMS will automatically merge keys across multiple properties files.

#### 9.1.1 Additional package

Coming back to the previous example of the conversion schemes, module *funky-module*, package `org.funky.conversion` contains additional conversion schemes.

These are all derived from `AbstractConversion`. In that case, the `ClassList.props` file would contain the following entry:

```
adams.data.conversion.AbstractConversion=\
    org.funky.conversion
```

When starting up, ADAMS will merge the two props files and the key will look like this, listing both packages:

```
adams.data.conversion.AbstractConversion=\
    adams.data.conversion,\
    org.funky.conversion
```

### 9.1.2 Additional class hierarchy

Adding a new class hierarchy works just the same. You merely have to use the superclass that all other classes are derived from as key in the props file and list all the packages to look for derived classes. Here is an example for a class hierarchy derived from `org.funky.AbstractFunkiness`, which has derived classes in the packages `org.funky` and `org.notsofunky`:

```
org.funky.AbstractFunkiness=\
    org.funky,\
    org.notsofunky
```

### 9.1.3 Blacklisting classes

In production environments it might not always be wise to list all the classes that are available, e.g., experimental classes. ADAMS provides a mechanism to exclude certain classes, using pattern matching (using regular expressions). These patterns are listed in the `ClassList.blacklist` properties file. The format for this file is similar to the `ClassList.props` file, with the *key* being the superclass and the *value* a comma-separated list of patterns. In the following an example that excludes a specified data conversion class called `SuperExperimentalConversion` from being listed:

```
adams.data.conversion.AbstractConversion=\
    org\.funky\.conversion\.SuperExperimentalConversion
```

If you want to exclude all conversions of the `org.funk.conversion` package that contain the word *Experimental*, then use the following pattern:

```
adams.data.conversion.AbstractConversion=\
    org\.funky\.conversion\..*Experimental.*
```

## 9.2 Creating a new actor

Being a workflow-centric application, it is most likely the case that a new module will contain new actors and not just newly derived subclasses of already existing superclasses. For this reason, the development of new actors is explained in detail.

Developing a new actor is fairly easy, you only need to do the following:

- create a new class
- create an icon, which is displayed in the flow editor
- *[optional, but recommended]* create a JUnit test for the actor

### 9.2.1 Creating a new class

Any actor has to be derived from *adams.flow.core.AbstractActor*. Depending on whether the actor consumes or produces data, there are two more interfaces available:

- *adams.flow.core.InputConsumer* – for actors that process data that they receive at their input
- *adams.flow.core.OutputProducer* – for actors that generate data of some form

In general, four types of actors can be distinguished, based on the combinations of these two interfaces:

- *standalone* – no input, no output
- *source* – only output
- *transformer* – input and output
- *sink* – only input

In order to make development of new actors easier and to avoid duplicate code as much as possible, there are already a bunch of abstract classes in ADAMS that implement these interfaces:

- *adams.flow.standalone.AbstractStandalone* – for standalones
- *adams.flow.source.AbstractSource* – for data producing source actors
- *adams.flow.transformer.AbstractTransformer* – for simple transformers that take one input token and generate at most one output token.
- *adams.flow.sink.AbstractSink* – the ancestor of all sinks, actors that only consume data

There are plenty more abstract super classes, since there are actors that perform similar tasks. Some of them are listed below:

- *adams.flow.sink.AbstractDisplay* – for actors displaying data in a frame or dialog
- *adams.flow.sink.AbstractGraphicalDisplay* – for actors that display graphical data, e.g., a graph, which can be saved to an image file automatically
- *adams.flow.sink.AbstractTextualDisplay* – for actors that display text

A special interface, *adams.flow.core.ControlActor*, is an indicator interface for actors that control the flow or the flow of data somehow. For instance, a *Branch* actor controls the flow of data, since it provides each sub-branch with the same data token that it received.

Actors that manage sub-actors, need to implement the *adams.flow.core.ActorHandler* interface.

The special superclass *adams.flow.control.AbstractControlActor* already implements the *ActorHandler* and *ControlActor* interfaces and implements some of

the functionality. The *AbstractConnectorControlActor* class in the same package, is used for control actors which sub-actors are connected, like the *Sequence* actor. The sub-actors in the *Branch* actor, on the other hand, are not connected, but treated individually.

The following methods you will usually have to implement:

- `globalInfo()` – The general help text for the actor.
- `doExecute()` – Here the actual execution code is located, the `pre-` and `post-` methods, you usually won't have to touch. All three methods are called in the `execute()` method.

## 9.2.2 Option handling

Option handling in ADAMS is available through classes implementing the `OptionHandler` interface (package `adams.core.option`). Most classes or class hierarchies, that includes the actors, are simply derived from `AbstractOptionHandler`, which implements this interface and all the required methods. For adding a new option, there are usually only three things to do:

1. add the (protected) **field**
2. add the **get-**, **set-** and **tiptext-methods** that make up the new property of this class
3. add an option **definition**

### 9.2.2.1 Example

The following shows how to implement a new option for an integer field *volume* that only allows values from 1 to 11. For clarity's sake, Javadoc comments have been left out.

First of all, we define the (serializable) field:

```
protected int m_Volume;
```

Then we add the required methods<sup>1</sup>:

```
public void setVolume(int value) {
    if ((value >= 1) && (value <= 11)) {
        m_Volume = value;
        reset(); // notify object that the settings have changed
    }
}

public int getVolume() {
    return m_Volume;
}

public String volumeTipText() {
    return "The volume to crank up the speakers to.";
}
```

And finally, we define the option, by overriding the `defineOptions()` method. Otherwise, the option won't show up in the GUI and you won't be able to set the value via a command-line string.

---

<sup>1</sup>The `tiptext` method generates the help text in the GUI and command-line, so you should never omit this.

```

public void defineOptions() {
    super.defineOptions();
    m_OptionManager.add(
        "volume",    // flag on the command-line without the leading "-"
        "volume",    // the Java Bean property for getting/setting the value
        1,           // the default volume
        1,           // the minimum value
        11);         // the maximum value
}

```

For numeric values, like integers and doubles, you can specify the lower and upper bounds, if that makes sense, like in our example here. If one of them is to be unbounded, simply use `null`. If both are unbounded, then simply omit the last two parameters.

### 9.2.3 Variable side-effects

TODO

### 9.2.4 Graphical output

TODO

### 9.2.5 Textual output

TODO

### 9.2.6 Creating an icon

The icon has to be placed in the *adams.gui.images* package with the same name as the class, but with a GIF or PNG extension. E.g., the *Display* actor's full class name is *adams.flow.sink.Display*. This means that ADAMS expects an image called *adams.flow.sink.Display.gif* or *adams.flow.sink.Display.png* in the *adams.gui.images* package.

There are already some templates available for new icons:

- *adams.flow.standalone.Unknown.gif* – red outline
- *adams.flow.source.Unknown.gif* – orange outline
- *adams.flow.transformer.Unknown.gif* – green outline
- *adams.flow.sink.Unknown.gif* – grey outline
- *adams.flow.control.Unknown.gif* – blue outline

Just create a copy of one of these icons and modify it to make your actor distinguishable from all the others in the flow editor.

### 9.2.7 Creating a JUnit test

JUnit 3.8.x [8] is used as basis for the unit tests. Test classes are placed in *src/test/java* and have to be suffixed with *Test*. E.g., the *Display* actor has a test class called *DisplayTest* in package *adams.flow.sink*.

A flow unit test needs to be derived from *adams.flow.AbstractFlowTest* and only the *getActor()* method needs to be implemented by default. This method

typically returns a Flow actor which is set up and executed. If any step in the lifecycle of the actor returns an error, the unit test will fail.

If required, a regression test can be performed. For this, you merely need to implement a method called *testRegression()*, which calls the *performRegressionTest(File)* or *performRegressionTest(File[])* method. These methods record the content of the specified files in a special reference file (found below *src/test/resources/regression*) and the next time the test is run the newly generated output is compared against the stored reference data. If the data differs, the regression test will fail. Please note, that you should remove temporary files that you use for regression tests in the *setUp()* and *tearDown()* methods of the unit test, to provide a clean environment to this and other tests.

## 9.3 Creating a new module or project

### 9.3.1 Module

First, you have to make sure that your local repository catalog is up-to-date:

```
mvn archetype:update-local-catalog
```

Second, run the following command to create a new module called *adams-funky*:

```
mvn archetype:generate \
    -DarchetypeCatalog=local \
    -DinteractiveMode=false \
    -DarchetypeGroupId=adams \
    -DarchetypeArtifactId=adams-archetype-module \
    -DarchetypeVersion=0.3.8 \
    -DgroupId=adams \
    -DartifactId=adams-funky \
    -Dversion=0.0.1-SNAPSHOT
```

This command will base the module on the latest ADAMS release, using the 0.3.8 release of the template (or *archetype*, to use the correct maven term). The version number of the newly created module will be 0.0.1-SNAPSHOT.

After the command has finished, you have to update the *Module.props* file in the *src/main/java/adams/env* directory. The minimal change that you have to perform is to set the correct module name, specified under the *Name* key. Apart from the name, this properties file contains information about your module, which gets displayed automatically in the *About* dialog in the GUI.

#### Official modules

If you run the above command within the top-level directory that hosts all the other ADAMS modules, then it will automatically add this module to the *pom.xml* configuration file as a new dependent module. This means that each time you issue a command in this directory (e.g., **mvn package**), your module will be processed accordingly. This is the preferred approach when adding a new module to be added to the ADAMS subversion repository.

#### Other modules

Otherwise, if you created that module outside the ADAMS module hierarchy, it will use the artifacts that you have installed in your local repository (of

course, maven will occasionally check the Nexus repository manager for updates). Use this approach if there is no intention on adding the module to the official ADAMS subversion repository.

### 9.3.2 Project

TODO

## 9.4 Main menu

The main menu of ADAMS can use a pre-defined menu structure, as defined in the `adams/gui/Main.props` properties file. But it also offers dynamic addition of other menu items at runtime.

In order for new menu items being picked up at runtime, you need to derive a new menu item definition from the following class (or one of the appropriate abstract classes derived from it):

`adams.gui.application.AbstractMenuItemDefinition`

For instance, if you merely want the menu item to open a browser with a specific URL (displaying the homepage or some help page), then you can derive the menu item from the following class:

`adams.gui.application.AbstractURLMenuItemDefinition`

If you don't want to modify the dynamic class discovery (`ClassListner.props`), then you have to place your newly created menu item definition in the following package:

`adams.gui.menu`

In order to get implement a menu item, derived from `AbstractMenuItemDefinition`, you need to implement or override the following methods:

- `getTitle()` – The text of the menu item.
- `getIconName()` – By default, the menu item won't have an icon, specify the filename (without path) of the icon that you would like to use. The icon is expected to reside in the `adams/gui/images` directory.
- `getCategory()` – This string defines the menu the item will get added to. Existing ones are, e.g., *Tools* or *Maintenance*. You don't have to use an existing one, new categories get automatically added as new menus.
- `isSingleton()` – If your menu item can be launched multiple times, then return *false*, otherwise *true*.
- `getUserMode()` – This defines the visibility of your menu item. Whether it is intended for regular users, experts or developers. What level is being displayed is defined – normally – by the application's `-user-mode <mode>` command-line option when starting the application.
- `launch()` – This method finally launches your custom code. More details below.

### The `launch()` method

For the best integration within ADAMS, the `launch()` will create a `java.swingx.JPanel` derived panel and create an internal frame using the following call:

```
JPanel panel = new MyFunkyPanel();  
ChildFrame frame = createChildFrame(panel);
```

Using this approach, your panel will show up in the *Windows* menu of the main menu of ADAMS.

Menu items derived from *AbstractURLMenuItemDefinition* don't need to implement this method, they merely need to supply a URL string with the *getURL()* method. Their *launch()* method uses this URL to open a browser with.



# Bibliography

- [1] *Kepler* – A free and open source, scientific workflow application.  
<https://kepler-project.org/>
- [2] *KeplerWeka* – A module for the Kepler workflow engine, which adds WEKA functionality.  
<http://keplerweka.sourceforge.net/>
- [3] *ADAMS* – Advanced Data mining and Machine learning System. The community homepage is available at the following URL:  
<http://adams.cms.waikato.ac.nz/>
- [4] *Apache Subversion* – An open-source, centralized version control system.  
<http://subversion.apache.org/>
- [5] *Apache Maven* – Software project management and comprehension tool.  
<http://maven.apache.org/>
- [6] *Nexus* – Repository manager for Apache Maven.  
<http://nexus.sonatype.org/>
- [7] *Eclipse* – An open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle.  
<http://eclipse.org/>
- [8] *JUnit* – JUnit is a unit testing framework for the Java programming language.  
<http://junit.org/>