ADAMS

Advanced **D**ata mining **A**nd **M**achine learning **S**ystem

Module: adams-core



Peter Reutemann

June 23, 2015

DRAFT

©2009-2013



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

I Using ADAMS **7**

1 Introduction **9**

2 Flows **11**

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 *Tee* . . . 26
2.2.3.2 Pull the *Trigger* . . . 27
2.2.3.3 Branching – or how to grow your flow . . . 28
2.2.3.4 Further control actors . . . 29
2.2.4 Protecting sub-flows . . . 31
2.3 Running flows . . . 33
2.3.1 Flow runner - GUI . . . 33
2.3.2 Flow runner - command-line . . . 33
2.4 Arrays and collections . . . 35
2.5 Converting objects . . . 36
2.6 String handling . . . 38
2.7 File handling . . . 39
2.8 Numeric operations . . . 41
2.9 JSON . . . 42
2.10 Properties . . . 43
2.11 XML . . . 44
2.12 Databases . . . 45
2.13 Callable actors . . . 46
2.14 External actors . . . 48
2.15 Interactive actors . . . 50
2.16 Templates . . . 51
2.17 Variables . . . 52
2.18 Temporary storage . . . 58
2.19 Debugging your flow . . . 62
2.19.1 Breakpoints . . . 62
2.19.2 Monitoring . . . 64
2.20 Passwords . . . 66
2.21 External processes and classes . . . 67

3 Visualization **69**
3.1 Image viewer . . . 69
3.2 Preview browser . . . 70

4 Tools **73**
4.1 Flow editor . . . 73
4.2 Flow runner . . . 73
4.3 Actor usage . . . 73
4.4 Text editor . . . 74
4.5 Comparing text . . . 76

5 Maintenance **77**
5.1 Placeholder management . . . 78
5.2 Named setup management . . . 82
5.3 Favorites management . . . 84

6 Customizing ADAMS **95**
6.1 Environment variables . . . 95
6.2 Properties files . . . 95
6.3 Main menu . . . 96
6.4 Flow editor . . . 97
6.5 Proxy . . . 97
6.6 Time zone . . . 97
6.7 Locale . . . 98
6.8 Database access . . . 98
6.9 Browser . . . 99

II Developing with ADAMS **101**

7 Tools **103**
7.1 Subversion . . . 103
7.2 Maven . . . 104
7.2.1 Nexus repository manager . . . 104
7.2.2 Configuring Maven . . . 104
7.2.3 Common commands . . . 105
7.2.4 3rd-party libraries . . . 106
7.2.5 Troubleshooting . . . 106
7.3 IntelliJ IDEA . . . 106
7.4 Eclipse . . . 106
7.4.1 Plug-ins . . . 106
7.4.2 Setting up ADAMS . . . 107
7.5 Custom Maven project . . . 107
7.6 Non-maven approach . . . 108

8 Using the API **111**
8.1 Flow . . . 111
8.1.1 Life-cycle of an actor . . . 111
8.1.2 Setting up a flow . . . 112

9 Extending ADAMS **115**
9.1 Dynamic class discovery . . . 115
9.1.1 Additional package . . . 115
9.1.2 Additional class hierarchy . . . 116
9.1.3 Blacklisting classes . . . 116
9.2 Creating a new actor . . . 116
9.2.1 Creating a new class . . . 117
9.2.2 Option handling . . . 118
9.2.2.1 Example . . . 118
9.2.3 Variable side-effects . . . 119
9.2.4 Graphical output . . . 120
9.2.5 Textual output . . . 120
9.2.6 Creating an icon . . . 121
9.2.7 Creating a JUnit test . . . 121
9.3 Creating a new module . . . 121
9.4 Main menu . . . 122
9.5 Flow editor . . . 123
9.5.1 Main menu . . . 123
9.5.2 Popup menu . . . 124
9.6 Image viewer . . . 125
9.7 Database access . . . 125

10 JUnit tests **127**

11 Temporary files **129**

12 Parser plugins **131**
12.1 Programmatic hooks . . . 132

Bibliography **133**

List of Figures

2.1 Flow editor with an empty new flow (*File* → *New* → *Flow)* . . . 13
2.2 Popup menu for adding a new actor . . . 14
2.3 Selecting a different actor . . . 15
2.4 Searching for *StringConstants* actor . . . 16
2.5 Help dialog for the *StringConstants* actor . . . 17
2.6 Adding the *Hello World!* string . . . 18
2.7 Flow after adding the *StringConstants* actor . . . 18
2.8 Tab displaying help for the *StringConstants* actor . . . 19
2.9 Tab displaying the non-default options for the *StringConstants* actor . . . 19
2.10 Adding another actor *after* the current one . . . 20
2.11 Searching for the *Display* actor . . . 20
2.12 The complete *Hello World* flow . . . 21
2.13 The output of *Hello World* flow . . . 21
2.14 Adding an additional actor . . . 22
2.15 Adding the *Convert* transformer . . . 23
2.16 Configuring the *Convert* transformer . . . 23
2.17 Extended *Hello World* flow . . . 24
2.18 Output of the extended *Hello World* flow . . . 24
2.19 Customizing the *StringReplace* transformer . . . 24
2.20 Output of further extended *Hello World* flow . . . 25
2.21 The *Hello World* flow with *Tee* actors. . . . 26
2.22 The log file generated by the *Tee* actors. . . . 27
2.23 A customized *ConditionalTee* actor. . . . 27
2.24 The *Hello World* flow with an additional *Trigger* actor. . . . 28
2.25 The modified log file generated with the additional *Trigger* actor. 28
2.26 The *Hello World* flow using a *Branch* actor. . . . 29
2.27 Flow runner interface with a flow for generating the Mandelbrot set. . . . 33
2.28 Flow editor interface with a flow for generating the Mandelbrot set. . . . 34
2.29 Outputting parallel processed strings in a single callable *Display* actor. . . . 46
2.30 Editing an external flow directly. . . . 48
2.31 An *inlined* or *expanded* external flow. . . . 49
2.32 Simple flow that prompts the user to enter a value, using a default value of "42" and a custom message. . . . 51
2.33 Adding a sub-flow generated from a template to an existing flow. 52

2.34 The options of the *UpdateVariable* template. 52
2.35 The added sub-flow. 53
2.36 The asterisk ("*") next to an option indicates that a variable is attached. 54
2.37 Using a variable to control what file to load and display. 54
2.38 Using a variable to control what external flow to execute (flow). . 55
2.39 Using a variable to control what external flow to execute (output). 55
2.40 Flow demonstrating the temporary storage functionality. 58
2.41 Output of flow demonstrating the temporary storage functionality. 59
2.42 Flow demonstrating the LRU cache storage functionality. 59
2.43 Display of the temporary storage during execution. 60
2.44 Output of flow demonstrating the LRU cache storage functionality. 60
2.45 The debugging control panel. 62
2.46 Example flow with *Breakpoint* actor. 63
2.47 The *Inspection* tab of the debugging control panel for the current token. 64
2.48 The *Breakpoints* tab of the debugging control panel. 65

3.1 Displaying a fractal in the Image viewer. 69
3.2 Preview browser displaying an image. 70
3.3 Preview browser displaying a flow. 71
3.4 Preview browser displaying an plain text file. 71

4.1 Overview of actor usage in flow files. 74
4.2 Editor for viewing/editing plain text files. 75
4.3 Comparing two text files. 76

5.1 Viewing the currently defined placeholders. 78
5.2 Entering the name for a new placeholder. 79
5.3 Selecting the directory that the new placeholder represents. . . . 79
5.4 The updated view of the placeholders. 80
5.5 Making the placeholder changes persistent. 80
5.6 Editing the path of a placeholder. 81
5.7 Selecting the new directory that the placeholder should represent instead. 81
5.8 Viewing the currently defined named setups. 82
5.9 The class hierarchy for the named setup. 82
5.10 Selecting the configuration that the new named setup represents. 83
5.11 The *nickname* for the setup. 83
5.12 The updated view of the named setups. 83
5.13 Making use of a favorite in the Flow editor. 84
5.14 Adding a setup in the object editor to the favorites. 85
5.15 Adding a favorite for new superclass. 86
5.16 Configuring the new favorite. 87
5.17 Naming the favorite. 87
5.18 The updated favorites view. 88
5.19 Changing a different setup for a favorite. 89
5.20 The view with the updated favorite. 90
5.21 Choosing a new name for the favorite. 91
5.22 The view with the renamed favorite. 92

5.23 Saving the modified favorites. 93

6.1 Proxy preferences 98
6.2 Time zone preferences 98
6.3 Locale preferences 99

7.1 texlipse configuration for the *adams-core* module. 108
7.2 Screenshot of the "Roll your own" section of the ADAMS website. 109

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 callable 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 – *adams.flow.provenance.ProvenanceSupporter* – 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.

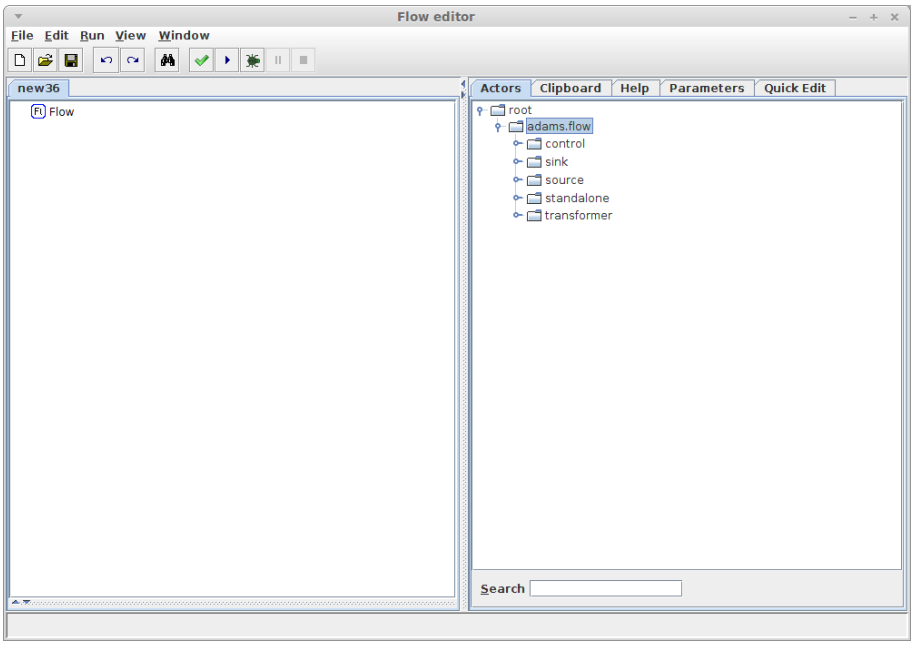


Figure 2.1: Flow editor with an empty new flow (*File* → *New* → *Flow)*

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 [1] 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.

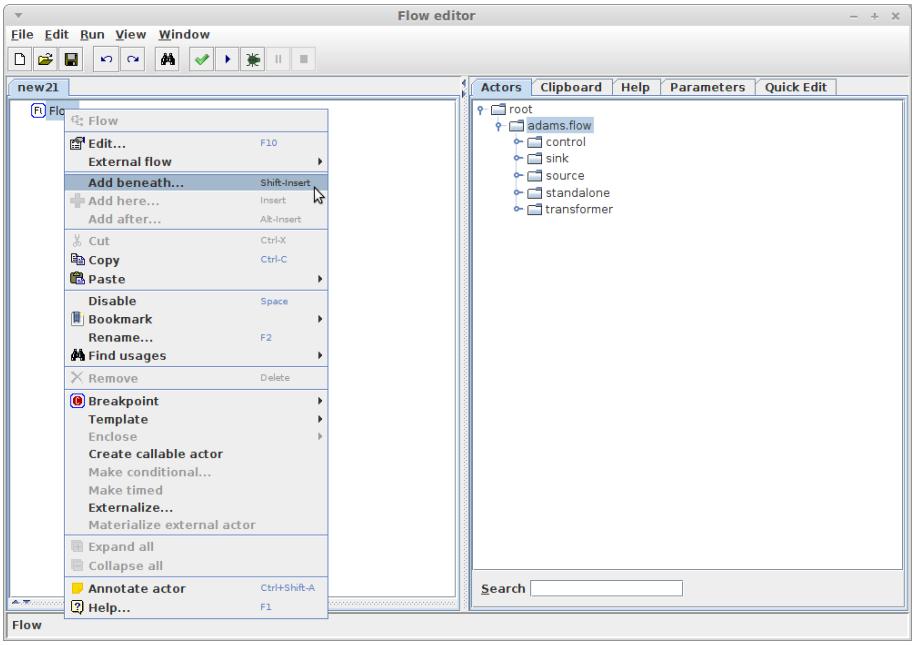


Figure 2.2: Popup menu for adding a new actor

ADAMS tries to suggest an actor depending on pre-defined rules and the context where the actor will get placed. The ones pre-defined by rules will be automatically available through the combobox at the top. But due to the large amount of actors, quite often 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 [2], most of the time it is quicker

[1] adams-core-hello_world1.flow

[2] ADAMS automatically filters actors that won't fit where you want to place a new actor.

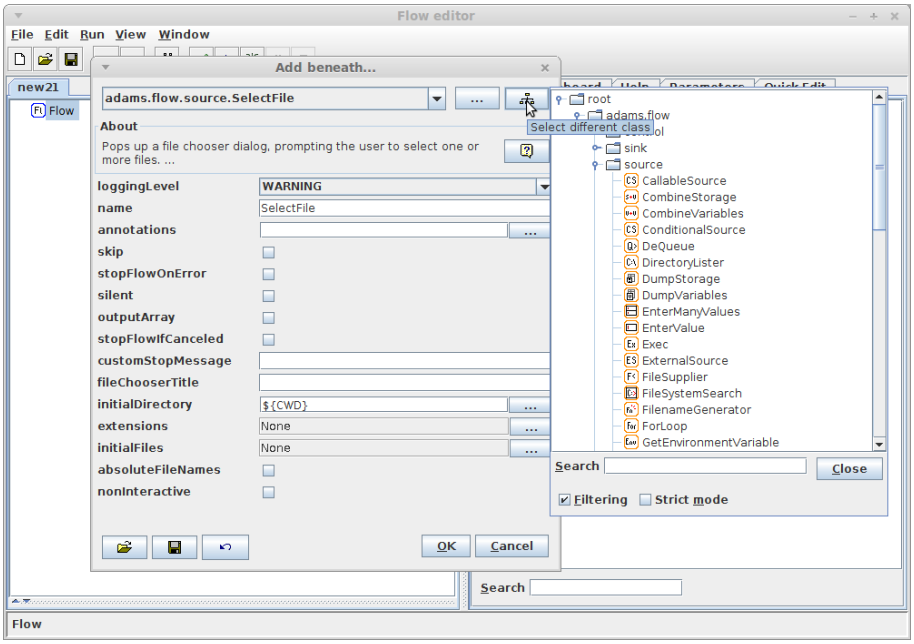


Figure 2.3: Selecting a different actor

to use the search facility of the popup displaying the class tree. Either click in the search box or use the shortcut `Alt+S` to jump there. As soon as you type, the display filters out all the class names that don't match the entered string. After entering *str* you will see a result similar to Figure 2.4.

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 *strings* property holds all the user-specified strings that this source actor will output. In our case, we just want to output `Hello World!`. Open up the array editor for the *strings* property, by clicking on the `...` button for this property. In order to enter a string value in this dialog, just click on the ... button again and enter the value as shown in Figure 2.6 and click on *OK*.

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 inidicates a recent 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

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 the GenericObjectEditor instead of a simple label.

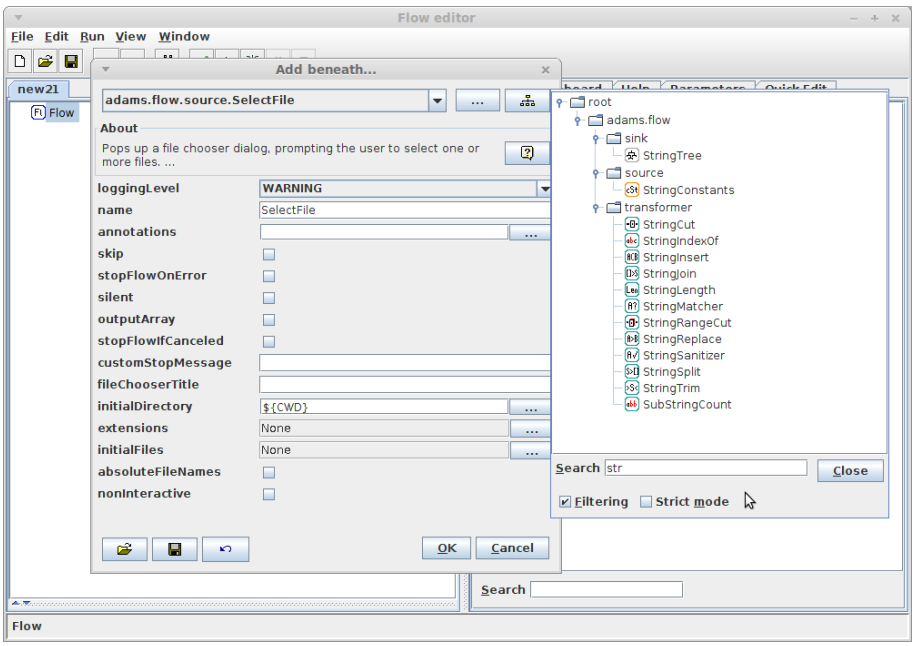


Figure 2.4: Searching for *StringConstants* actor

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 2.7 shows the resulting flow.

The Flow editor offers help also through the tabs on the right hand side. The *Help* tab (Figure 2.8) 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 (2.9). This can be quite handy when quickly going through multiple actors, checking their values. Especially actors with lots of options.

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.

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

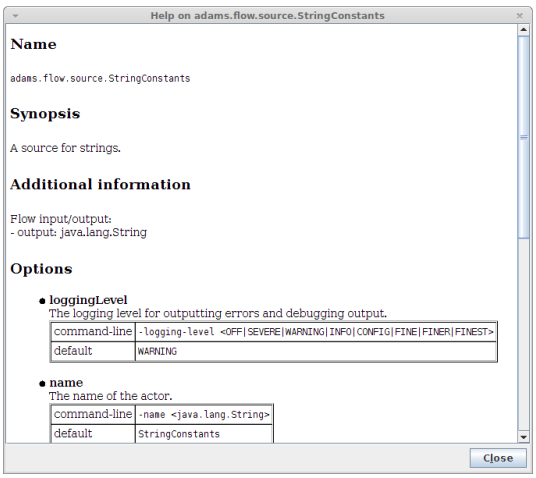


Figure 2.5: Help dialog for the *StringConstants* actor

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 *Run* → *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!

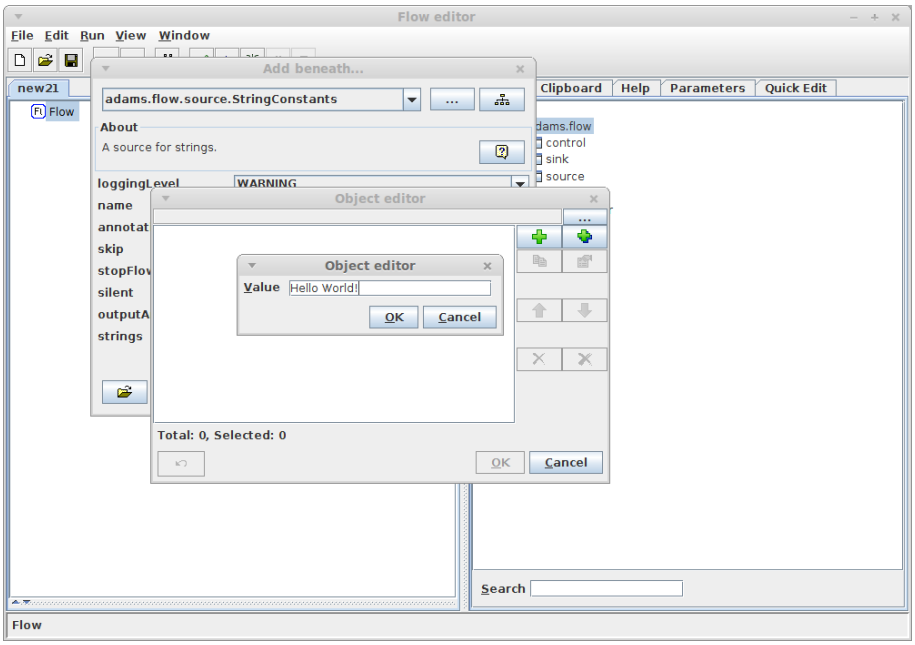


Figure 2.6: Adding the *Hello World!* string

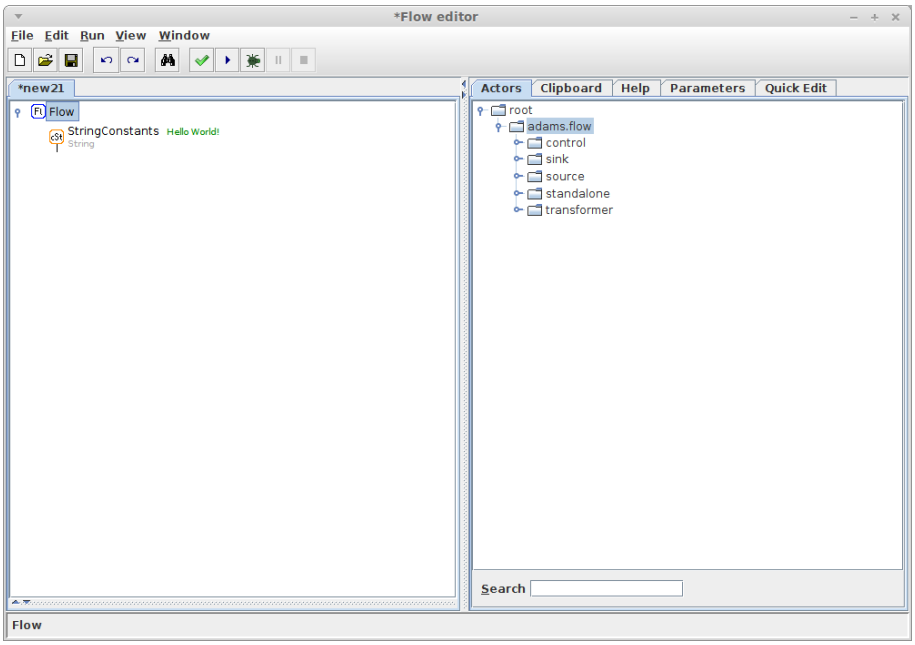


Figure 2.7: Flow after adding the *StringConstants* actor

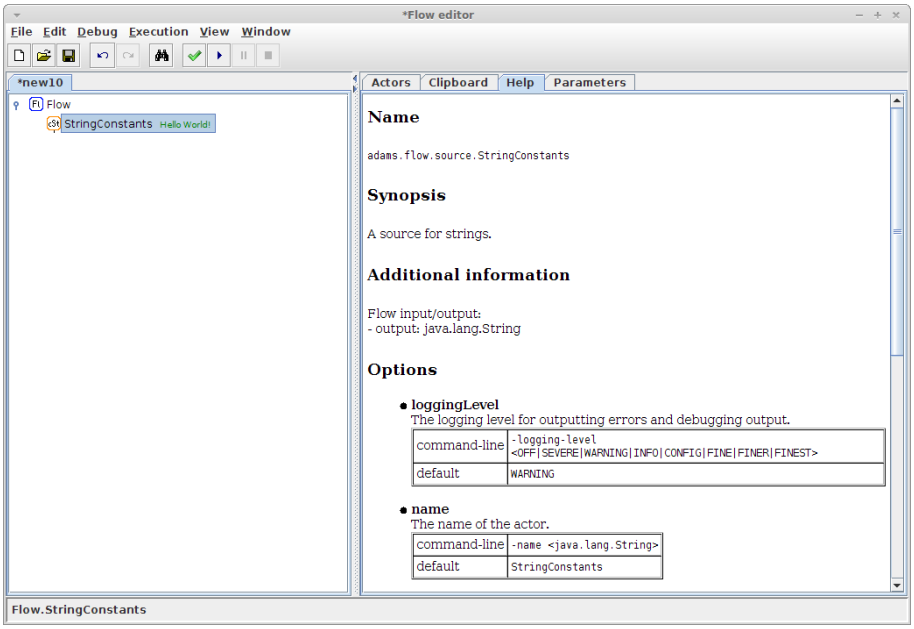


Figure 2.8: Tab displaying help for the *StringConstants* actor

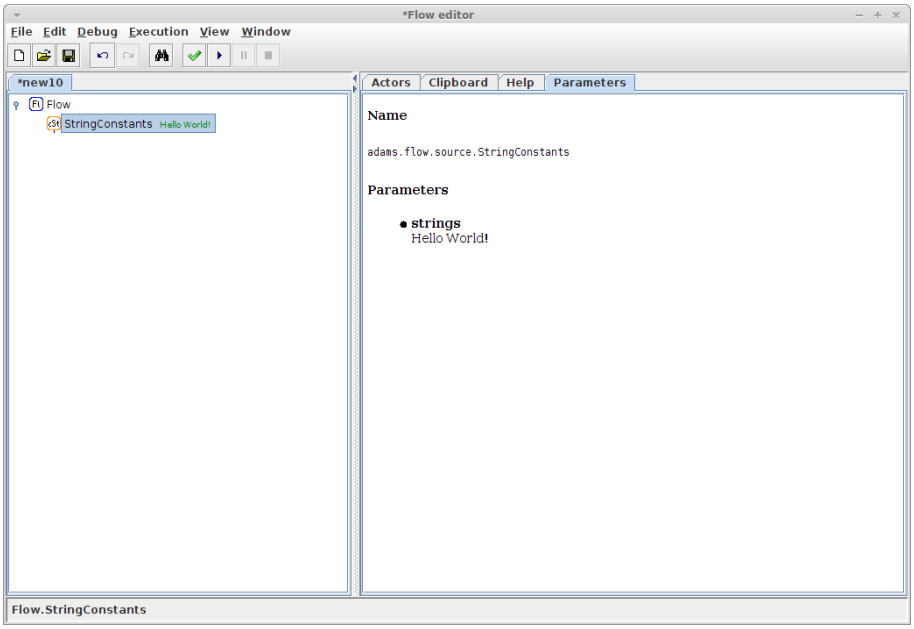


Figure 2.9: Tab displaying the non-default options for the *StringConstants* actor

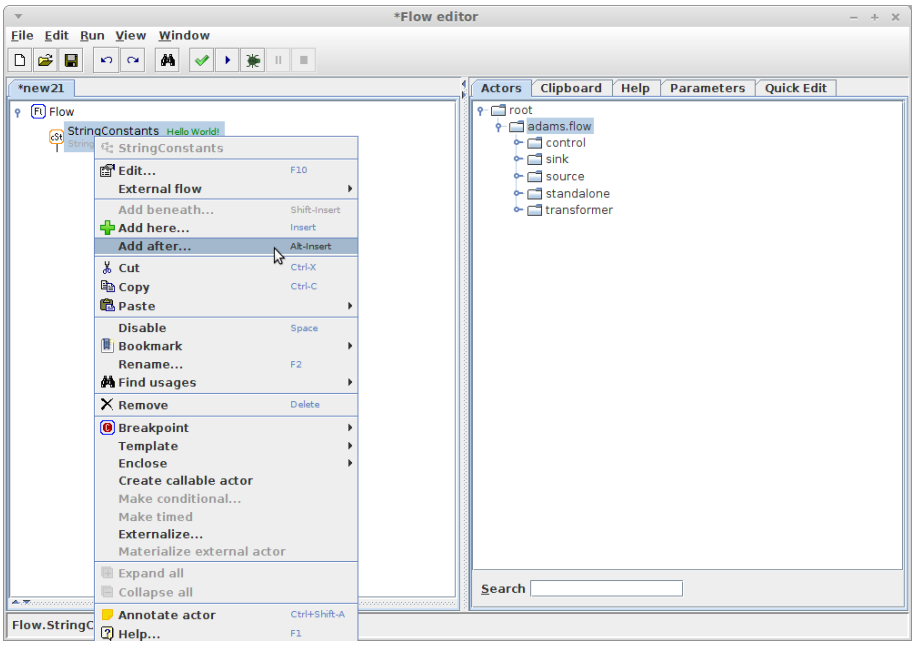


Figure 2.10: Adding another actor *after* the current one

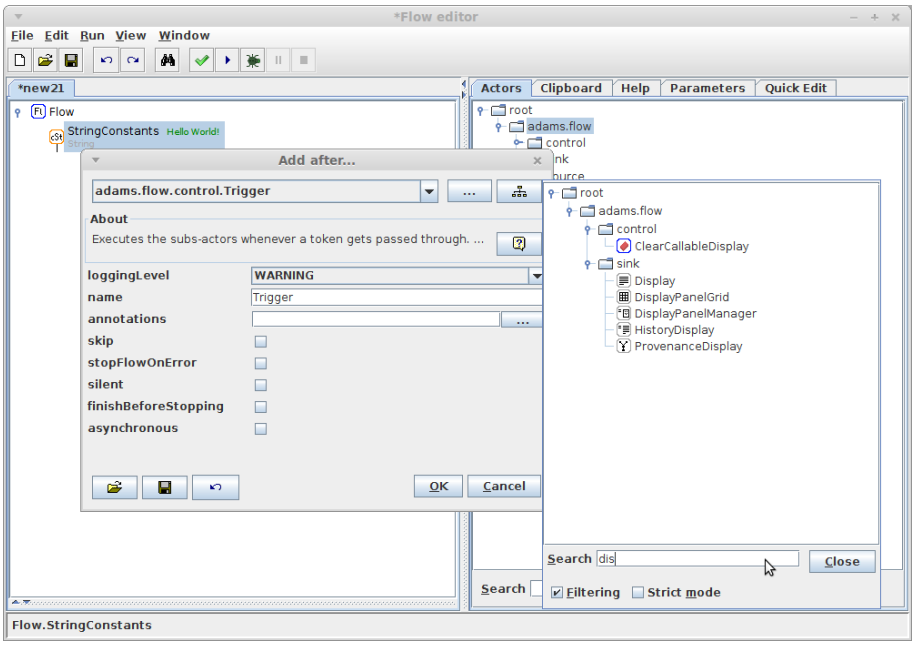


Figure 2.11: Searching for the *Display* actor

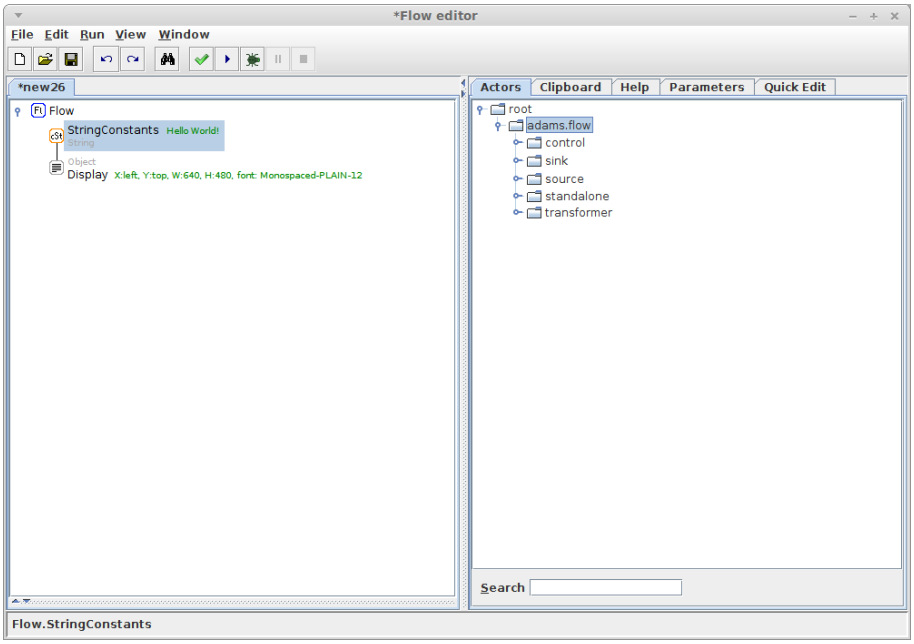


Figure 2.12: The complete *Hello World* flow

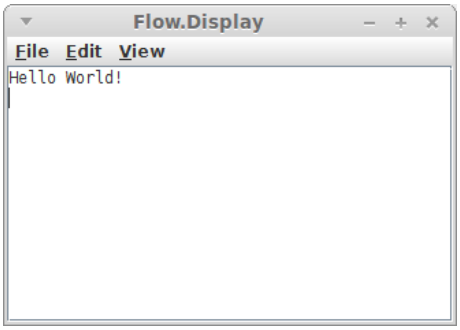


Figure 2.13: The output of *Hello World* flow

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 [3] 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.

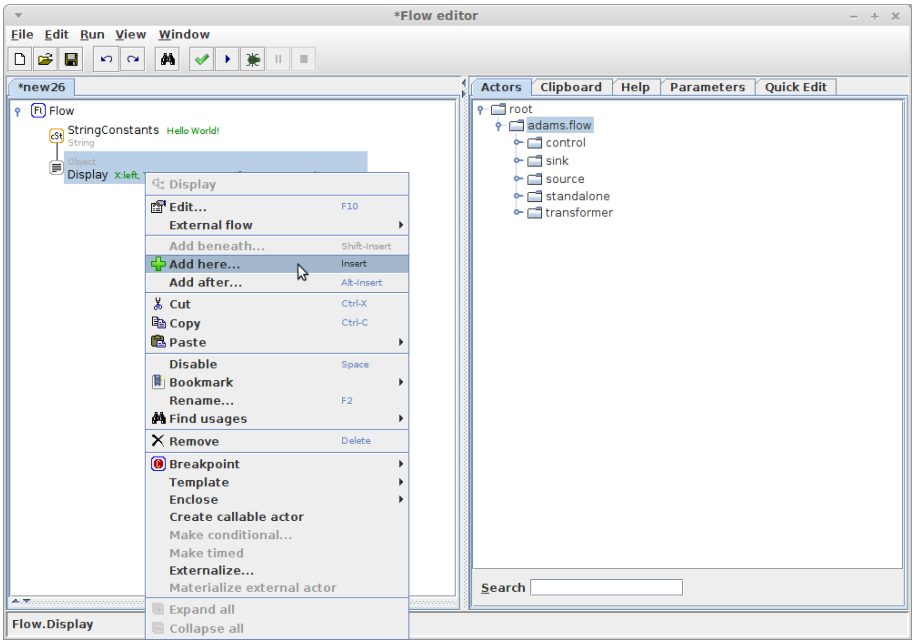


Figure 2.14: Adding an additional actor

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 [4] requires adding a custom string at the

[3] adams-core-hello_world2.flow
[4] adams-core-hello_world3.flow

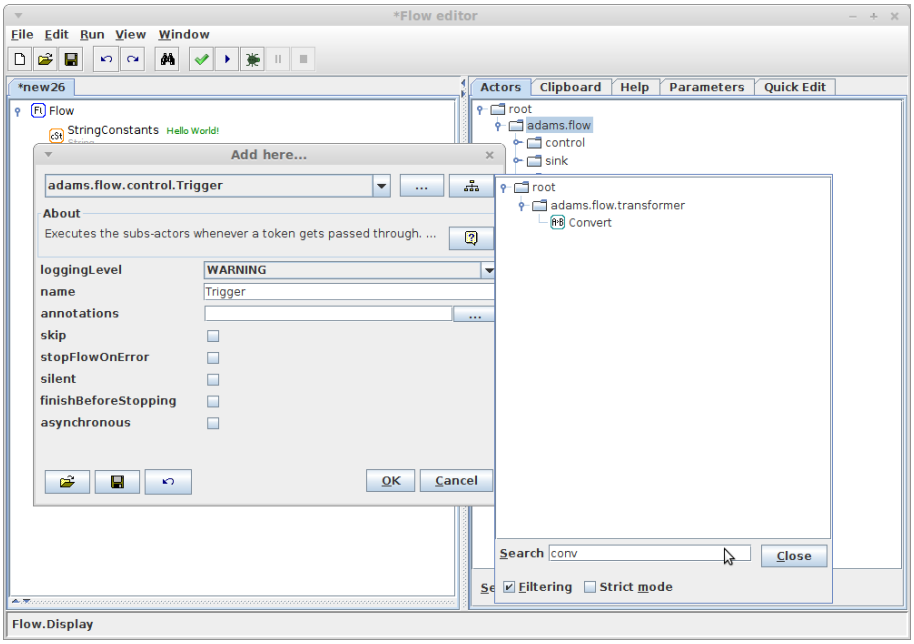


Figure 2.15: Adding the *Convert* transformer

end of the actor outputting *HELLO WORLD!*. We can achieve this by using the *StringReplace* actor, which allows us to perform string replacements using regular expressions [5]. 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).

In a lot of cases, regular expressions can be overkill for manipulating strings. E.g., when prepending or appending a string. In such simple cases, you also just use the simpler *StringInsert* transformer.

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

[5]For more information see `http://en.wikipedia.org/wiki/Regular_expressions`.

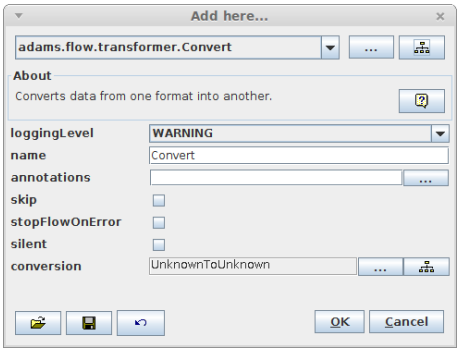


Figure 2.16: Configuring the *Convert* transformer

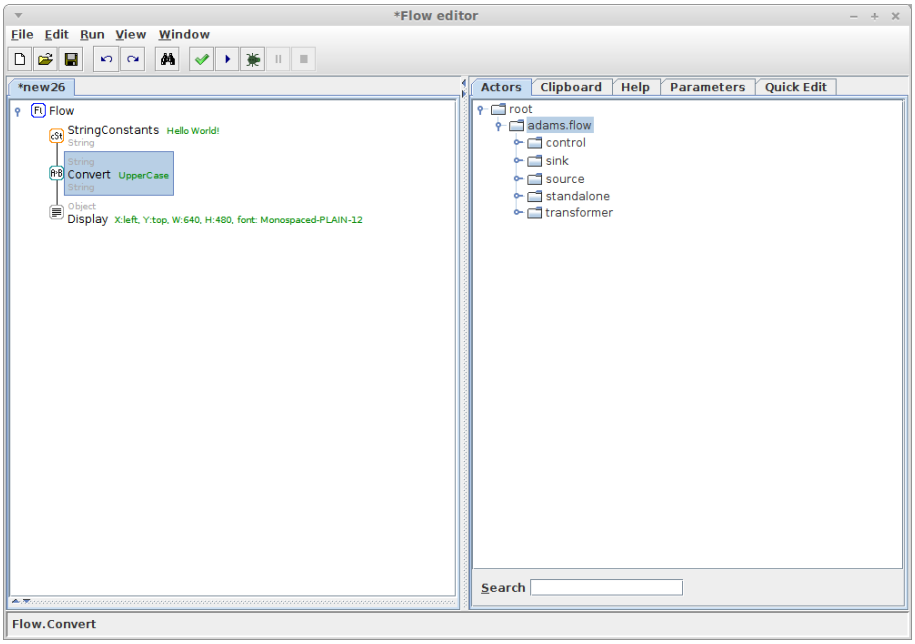


Figure 2.17: Extended *Hello World* flow

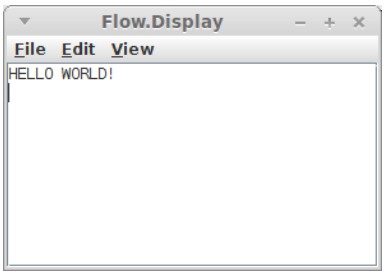


Figure 2.18: Output of the extended *Hello World* flow

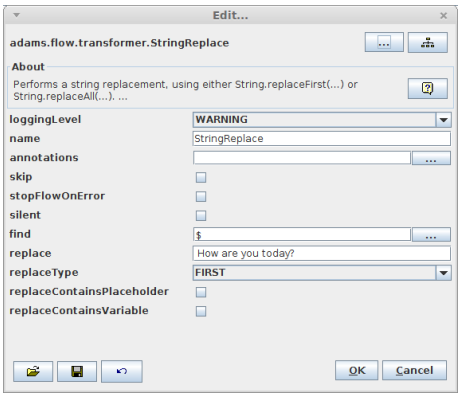


Figure 2.19: Customizing the *StringReplace* transformer

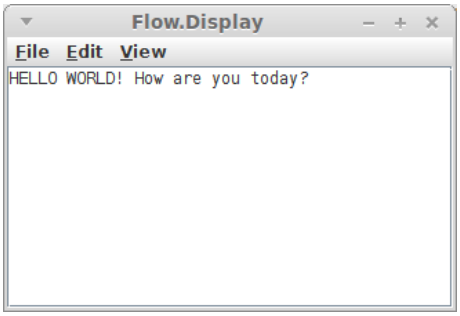


Figure 2.20: Output of further extended *Hello World* flow

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 introduced *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 [6] 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.

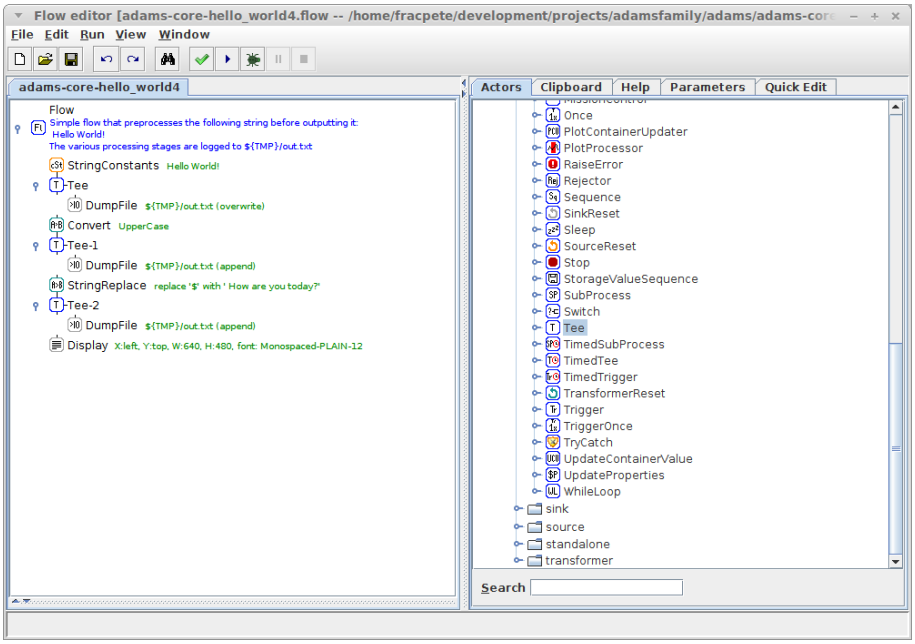


Figure 2.21: The *Hello World* flow with *Tee* actors.

[6] adams-core-hello_world4.flow

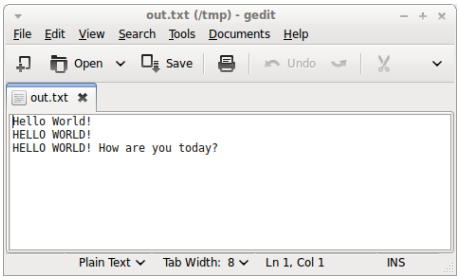


Figure 2.22: The log file generated by the *Tee* actors.

The *ConditionalTee* control actor is an extended version of the simple *Tee* actor. This actor only tees off the token if its boolean condition returns *true*. For instance, using the *Counting* condition, this actor will keep 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, 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.

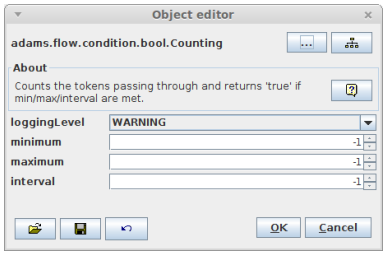


Figure 2.23: A customized *ConditionalTee* actor.

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 [7] 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 actor, besides the *Flow* control actor, that allows *standalone* actors to be added to it.

[7] adams-core-hello_world5.flow

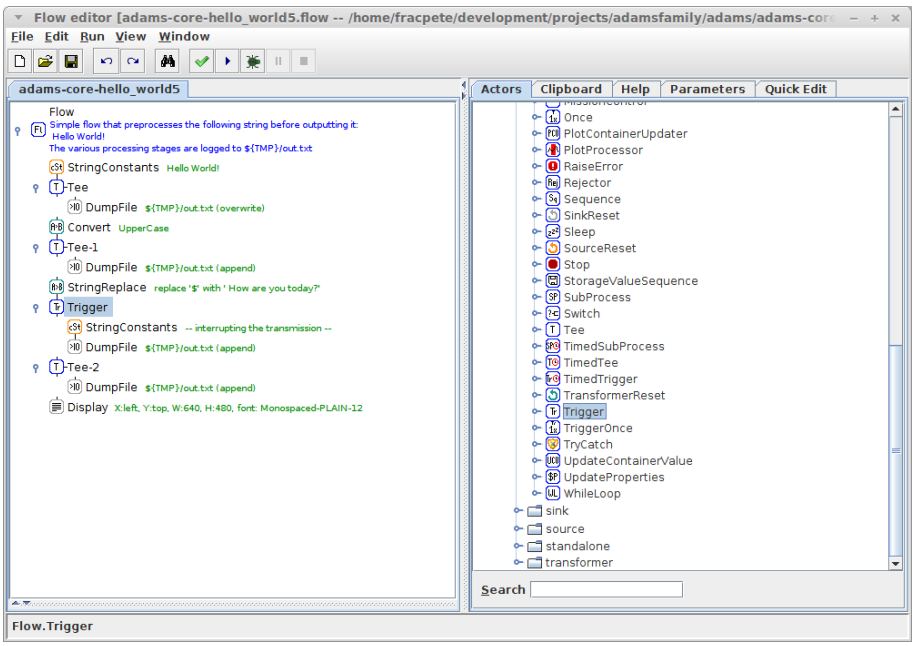


Figure 2.24: The *Hello World* flow with an additional *Trigger* actor.

A variant of the *Trigger* actor is the *ConditionalTrigger* actor. This actor only executes the sub-flow if its boolean condition returns *true*. *TriggerOnce*, another variant, triggers the sub-flow exactly once, useful for initializations.

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 [8]. 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

[8] adams-core-hello_world6.flow

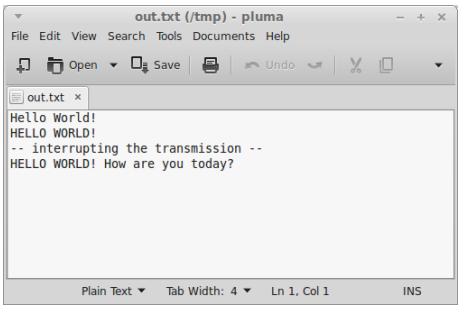


Figure 2.25: The modified log file generated with the additional *Trigger* actor.

to encapsulate the actors in a *Sequence* control actor. The default setting of

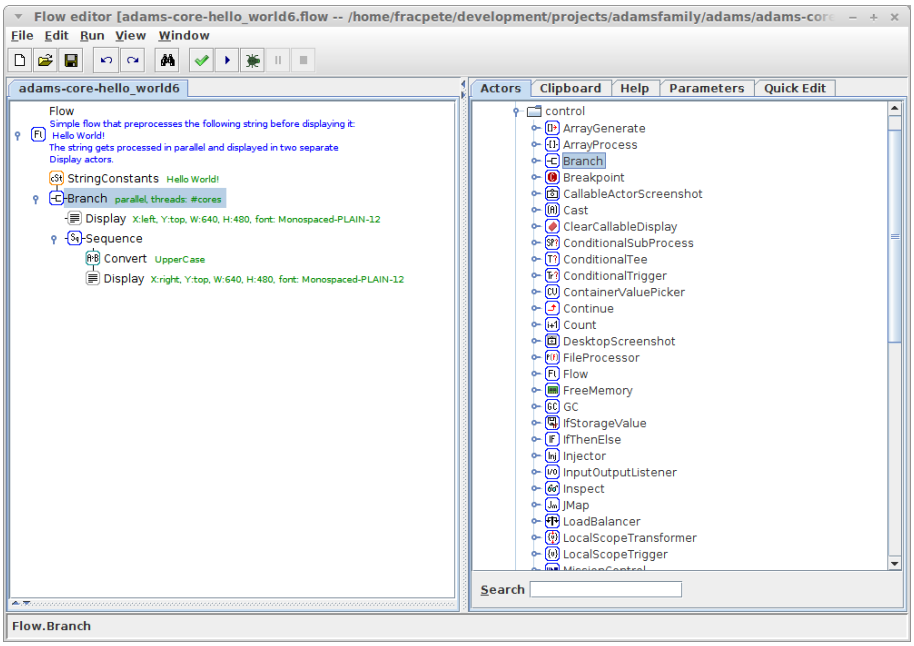


Figure 2.26: The *Hello World* flow using a *Branch* actor.

the *Branch* actor is to process the branches in separate threads, taking the maximum number of cores/CPUs 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 fro 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 system 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:

- *ArrayGenerate* – generates an array of the output generated by its sub-actors, using the same input token.
- *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 2.19 for more details.
- *ClearCallableDisplay* – Can be used to clear callable graphical actors, e.g., a *SequencePlotter*. See 2.13 for more information on callable 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.
- *Continue* – Does not pass on tokens if the specified boolean expression evaluates to *true*, i.e., acts like the "continue" control statement.
- *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.
- *DesktopScreenshot* – allows you to take a screenshot of the complete desktop.
- *FreeMemory* – Invokes the parent to wrap up all its sub-actors, effectively freeing up memory. Useful if branches of a many-branched *Branch* actor only get executed once, but still keep their state and hog memory.
- *GC* – For explicitly executing the Java garbage collection.
- *CallableActorScreenshot* – For taking screenshots of a callable (graphical) actor, whenever a token passes through this control actor. See 2.13 for more information on callable 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.
- *InputOutputListener* – Enables forwarding incoming tokens and tokens generated by sub-flow to callable actors.
- *Inspect* – A more specialized actor for *visualizing* data that is passing through the flow, interactively or not.
- *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.
- *LocalScopeTransformer* – Provides "local" variables and internal storage; useful when things run in parallel.
- *LocalScopeTrigger* – Provides "local" variables and internal storage; useful when things run in parallel.
- *Once* – A tee actor that only tees off the first token it encounters. A simplified *ConditionalTee* so to speak.
- *MissionControl* – shows a minimalistic control panel for pausing, resuming and stopping the flow.
- *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.

- *RaiseError* – Raises an error if its condition evaluates to *true* using the specified error message (see *TryCatch*).
- *Rejector* – Rejects tokens container data containers that have error messages attached.
- *Sequence* – Allows to specify multiple actors that get exectued one after the other, with the output of one actor being the input of the next.
- *SinkReset* – resets all its sub-actors in case the monitored variable changed value.
- *Sleep* – Suspends the flow execution for the specified number of milliseconds.
- *SourceReset* – resets all its sub-actors in case the monitored variable changed value.
- *Stop* – If executed, stops the flow execution.
- *StorageValueSequence* – For processing the same storage value multiple times in Triggers and/or Tees, but still outputting and forwarding it in the flow.
- *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*.
- *TransformerReset* – resets all its sub-actors in case the monitored variable changed value.
- *TryCatch* – Allows you to protect a sub-flow in a "try" block. If the execution fails for some reason the "catch" sub-flow gets executed to ensure that flow execution continues (see *RaiseError*).
- *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*.

2.2.4 Protecting sub-flows

By default, ADAMS tries to stop the flow execution as fast as possible. However, this behavior might not be desired in case of mission critical steps that should never get interrupted. For instance, when reading (and in the same step, removing) data from the database, that the output of said data on disk should get interrupted.

In order to allow the user to protect the execution of certain sub-flows, a fair amount of actors offer a flag for *atomic execution*. This flag is called *finishBeforeStopping* in the option dialog. When enabled, this actor will wait with stopping its sub-actors until the sub-actors have finished processing all their data. Actors that support this, are for example, *Sequence*, *Branch*, *Tee*,

Trigger (and derived classes).

Be careful how and where you use this flag, as it can have undesired side-effects: if you enable this flag in the *Flow* control actor, then the flow cannot be stopped before all processing has finished.

2.3 Running flows

Executing flows from the *Flow editor* is just one of the options of how to execute a flow. Unless you want the ability to edit the flow, you could use one of the following options.

2.3.1 Flow runner - GUI

The *Flow runner* is an interface to execute flows without the user being able to modify them. This interface is used for merely executing flow. This can be useful for users that only *run* flows, but never modify them. Depending on the flow, the user is still able to influence its execution. The Flow runner interface analyzes the flow and displays the topmost `SetVariable` standalones as parameters that the user then can modify. Figure 2.28 shows the flow in the editor interface and Figure 2.27 in the runner interface. Annotations that are attached to the `SetVariable` actors are available as help through a button next to the edit field with the variable value.

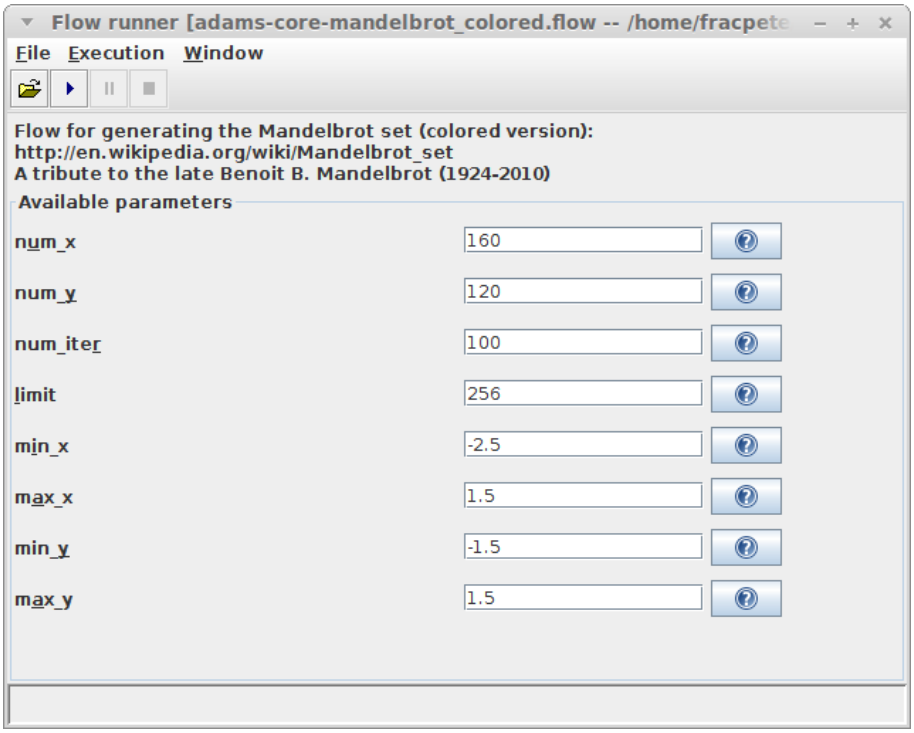


Figure 2.27: Flow runner interface with a flow for generating the Mandelbrot set.

2.3.2 Flow runner - command-line

The ability to run flows on a server, in a headless environment, was one the requirements when desinging ADAMS. This can be either for data processing

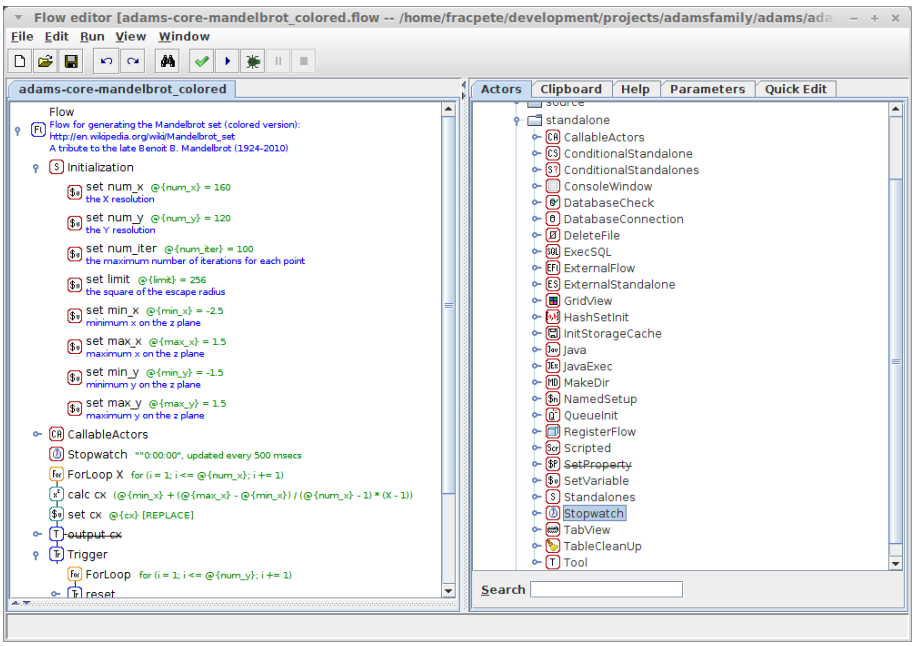


Figure 2.28: Flow editor interface with a flow for generating the Mandelbrot set.

flows that poll directories or databases or scheduled executions. The following class allows you to run a flow from command-line:

```
adams.flow.FlowRunner
```

Available options:

- *-help* – lists all the available options.
- *-input* – either a specific flow file or a directory containing flows to traverse (use *-include* regular expression to limit flows).
- *-headless* – whether to suppress all graphical output.
- *-clean-up* – remove any graphical output, like dialogs and plots when the flow finishes.
- *-no-execute* – if you only want to test flows whether they still load and pass the tests in the *setUp* phase, use this option.

2.4 Arrays and collections

ADAMS offers a range of actors for generating and processing arrays and collections.
The following control actors are available:

- *ArrayGenerate* – similar to the Branch actor, forwards the incoming token to all of its branches and constructs an array from the collected output and forwards this as a new token.

The following sources are available:

- *NewArray* – for generating empty arrays of arbitrary type and dimensions.
- *StorageValuesArray* – combines the specified items in storage into an array and outputs the array.

The following transformers are available:

- *ArrayCombination* – generations combinations or permutations of the elements of the array passing through.
- *ArrayLength* – determines the length of an array of any type
- *ArrayStatistic* – calculates statistics for the array(s), e.g., mean, standard deviation.
- *ArraySubset* – generates a new array with the chosen elements from the incoming array.
- *ArrayToSequence* – turns an array into a sequence of tokens.
- *CollectionToSequence* – turns a collection into a sequence of tokens.
- *GetArrayElement* – returns a specific element from the array.
- *Max* – returns index or value of the largest array element (integer or double).
- *Min* – returns index or value of the smallest array element (integer or double).
- *ObjectArrayToPrimitiveArray* – turns an array of objects into an array of is primitive counterparts.
- *PrimitiveArrayToObjectArray* – generates an array of objects from an array of primitives.
- *SequenceToArray* – turns a sequence of tokens back into an array.
- *SequenceToCollection* – turns a sequence of tokens into a collection (type is specified by user).
- *SetArrayElement* – sets the value of a speficied array element, either from a given value (or variable) or from a storage item.

Of special importance is the *ArrayProcess* control actor. This actor allows you to appply a sequence of actors - defined below the control actor - to all the elements of the array passing through. This is a shortcut to storing the length of the array in a variable (*ArrayLength*), turning an array into a sequence (*ArrayToSequence*), processing the individual tokens and then creating a new array from the sequence with known length (*SequenceToArray*).

2.5 Converting objects

Quite often, you will be faced with converting objects from one type into another, due to ADAMS' strong typing. Adding new actors for converting an object from one type to another (e.g., from *String* to *Integer* and vice versa), would just increase the already large number of actors even further. To avoid this, ADAMS offers a *catch-all* transformer for these simple conversions: *Convert*.

All conversion schemes are derived from the super class *AbstractConversion* (package `adams.data.conversion`). In contrast to actors, which allow an arbitrary number of classes, a conversion scheme can only define a single input and a single output class, underlying the *simple* aspect of the conversion.

Here are some conversion for numbers:

- *ByteToHex* – generates a hexadecimal representation of the byte.
- *ByteToInt* – turns a byte into an integer.
- *ByteToString* – generates a string representation of the byte.
- *DoubleToInt* – turns a double into an integer (calling the *intValue()* method of the Double object).
- *DoubleToFloat* – converts a double into a float (calling the *floatValue()* method of the Double object).
- *DoubleToLong* – turns a double into a long (calling the *longValue()* method of the Double object).
- *DoubleToString* – turns the double into a string, with the number of decimals specified by the user.
- *FloatToDouble* – converts a float into a double (calling the *doubleValue()* method of the Float object).
- *HexToByte* – turns hexadecimal strings into bytes.
- *HexToInt* – turns hexadecimal strings into integers.
- *IntToByte* – turns an integer into a byte (data loss may occur!).
- *IntToLong* – turns an integer into a long.
- *IntToDouble* – turns an integer into a double.
- *IntToHex* – generates a hexadecimal representation of the integer.
- *IntToString* – generates a string representation of the integer.
- *LongToInt* – turns a long into an integer (data loss may occur!).
- *LongToDouble* – turns a long into a double.
- *ObjectArrayToPrimitiveArray* – converts an object array to its primitve counterpart (e.g., Integer[] or Character[] to int[] or char[]).
- *PrimitiveArrayToObjectArray* – converts a primitive array to its object counterpart (e.g., int[] or char[] to Integer[] or Character[]).
- *Round* – rounds double values (round, ceiling, floor).
- *StringToByte* – parses the string representing a byte.
- *StringToDouble* – parses the string representing a double.
- *StringToInt* – parses the string representing an integer.
- *StringToLong* – parses the string representing a long.

Some more string conversions:

- *AnyToCommandline* – Generates a commandline string from any object.

- *AnyToString* – Uses the Object's *toString()* method.
- CommandlineToAny – Creates an object from the commandline (class + options).
- *BackQuote* – Escapes special characters like tab, new line, single and double quotes with backslashes. This can be reversed with *UnBackQuote*.
- *FieldToString* – turns the Field object into its string representation, with or without data type. The reverse is possible as well, using *StringToField*.
- *LeftPad* – left pads a string up to a maximum number of characters, e.g.: turning "1" into "001".
- *LowerCase* – turns a string into its lower case representation.
- *Quote* – surrounds a string quite single or double quotes if it contains blanks or special characters like tabs or new lines. Special characters will get backquoted as well. Thereverse operations is done using *UnQuote*.
- *TimeToString* – Turns a number representing milli-seconds since 1970 (Java date) into a String (see [10]).
- *UpperCase* – turns a string into its upper case representation.

Some more date/time conversions:

- *BaseDateTimeToString* – evaluates a date/time format string and it into a string.
- *BaseDateToString* – evaluates a date format string and it into a string.
- *BaseTimeToString* – evaluates a time format string and it into a string.
- *ConvertDateTimeType* – turns one date/time type into another, e.g., milliseconds into Date.
- *DateTimeTypeToString* – turns various date/time types into a string using a format string.
- *ExtractDateTimeField* – extracts various date/time fields (year, hour, day of week, etc) from date/time types.
- *StringToDateTimeType* – parses a date/time string using a specified format string and turns it into various date/time types.

2.6 String handling

Quite often when designing flows, you will be dealing with strings that need tweaking, e.g., for file names. The following set of common string operations is already implemented in ADAMS:

- *BreakUpString* – breaks up the string into multiple lines (separated by line-feed) using word-boundaries if wider than specified number of columns.
- *StringCut* – cuts out a single portion of the string, either based on column (using a specific separator) or character positions.
- *StringIndexOf* – locates a sub-string in the strings passing through.
- *StringInsert* – allows the insertion of a string into other string tokens, using a specified location.
- *StringJoin* – glues the individual elements a string array together into a single string.
- *StringMatcher* – either lets through or blocks strings that match a regular expression (see also [11]).
- *StringRangeCut* – cuts out one or more portions of a string and glues them back together again into a single string.
- *StringReplace* – uses pattern matching to find and replace parts of the string passing through (see also [11])..
- *StringSanitizer* – removes unwanted characters from a string as specified by the user (or vice versa: leaves only accepted characters)
- *StringSplit* – splits a string into sub-strings based on a regular expression (see also [11]).
- *StringTrim* – removes preceding/trailing whitespaces from the string passing through.
- *SubStringCount* – counts the occurrences of a sub-string in a string.

See also section 2.5 for some basic string conversions and section 2.7 on file handling.

2.7 File handling

The beauty of ADAMS lies in its ability to react dynamically to its processing environment and, if necessary, also bootstrap or modify it. File handling is an essential part in this. In the following you will find an overview of some of the actors and conversion that offer file-related actions.

Available conversions:

- *FileToString* – turns a file object into a string, either with a relative or absolute path. The reverse is possible as well, using
- *ReplaceFileExtension* – replaces the file's extension with a user-supplied one (or removes it if no new extension supplied). *StringToFile*.
- *StringToValidFileName* – ensures that the string passing through can be used as filename (excluding the path).

Available source actors:

- *DirectoryLister* – lists directories and/or files in a directory (lots of options: pattern matching, recursive, sorting, ...).
- *FilenameGenerator* – allows to create filenames using various generators.
- *FileSystemSearch* – searches the file system using the specified search algorithm.
- *SingleFileSupplier* – outputs a single file specified by the user.
- *MultiFileSupplier* – outputs multiple files specified by the user.

And transformers:

- *AppendName* – appends a suffix to the file/directory passing through.
- *BaseName* – strips the parent directory and forwards the remainder.
- *BinaryFileReader* – reads a specific range of bytes from a binary file and outputs the bytes one-by-one or as array.
- *CopyFile* – copies the file passing through to a target directory (if pattern matches).
- *DeleteFile* – deletes the file passing through if the pattern matches.
- *Deserialize* – loads a serialized Java object from disk.
- *Diff* – generates a diff[9] between two files.
- *DirName* – extracts the directory part of the file (or parent directory in case of a directory).
- *FileExtension* – extracts the extension of the filename (the part after the ".").
- *FileInfo* – outputs information on a file, like size or last modified timestamp.
- *FilenameGenerator* – allows to create filenames using various generators, with some of them utilizing the token passing through.
- *MakeDir* – creates the directory received on the input port (also available as standalone actor).
- *MoveFile* – moves/renames a file.
- *PrependDir* – prepends a directory ("prefix") to the file passing through.
- *SplitFile* – splits a file into several pieces using the specified split algorithm.

[9] `http://en.wikipedia.org/wiki/Diff`

- *TextFileReader* – loads the content of a text file.

Sink actors:

- *BinaryFileWriter* – writes a byte array of *BlobContainer* to a binary file.
- *FilePreview* – generates a simple preview of the file.
- *MergeFiles* – merges several files back into a single one.
- *OpenFile* – opens the incoming file with the associated native application.
- *PasteFiles* – combines all the files received at the input into a single file, joining them line by line. The user specifies what separator to use for glueing the lines together. Works similar to the Unix *paste* command.
- *Serialize* – saves any (serializable) Java object to a file.
- *SideBySideDiff* – displays a diff[10] between two files visually.

Finally, control actors:

- *FileProcessor* – processes files with a sub-flow, places them in *processed* or *failed* directories, depending on successful execution of sub-flow.

Conditional control actors, like *IfThenElse*, *Switch*, *ConditionalTee* or *ConditionalTrigger* you can use the following boolean conditions:

- *DirectoriesMatch* – scans a directory for sub-directories that must match a regular expression.
- *DirectoryExists* – checks whether a directory exists.
- *FileExists* – checks whether a file exists.
- *FilesMatch* – scans a directory for files that must meet a regular expression.

See also section 2.5 for some basic file conversions. Section 2.15 for interactive actors is also worth looking at, if the user should select a file or directory during flow execution.

[10] `http://en.wikipedia.org/wiki/Diff`

2.8 Numeric operations

Being targeted at the scientific community, ADAMS also offers some general purpose actors for numeric-related conversions:

- *RandomNumberGenerator* – source outputting random numbers (various generator types are available).
- *MathExpression* – calculates the result of a mathematical expression/formula (supports use of variables) – also available as source.
- *ReportMathExpression* – derives a value based on values from a report (or report handler) passing through and stores the result back in the report.
- *Round* – rounds the data passing through (ceiling, floor or plain round).

See also section 2.5 for some basic numeric conversions and 2.4 for processing arrays and calculating statistics.

2.9 JSON

Flows cannot only be saved as JSON files (and read in again), but flows themselves can process JSON data structures as well.
The following transformers are available:

- *GetJsonKeys* – outputs all named elements of a JSON object.
- *GetJsonValue* – outputs the named value from a JSON object, can use simple key or a JSON path[11].
- *JsonFileReader* – reads the specific JSON file and forwards a JSON object/array.

The following sinks are available:

- *JsonDisplay* – displays a JSON object in a browseable tree structure.
- *JsonFileWriter* – writes the JSON object/array to disk.

The following conversion are available:

- *ArrayToJsonArray* – generates a JSON array from any object array.
- *JsonArrayToArray* – turns a JSON array into a regular Java object array.
- *JsonToString* – turns the JSON object/array into a string.
- *StringToJson* – parses the string and generates a JSON object/array.

[11] `http://code.google.com/p/json-path/`

2.10 Properties

ADAMS can also process Java Properties files directly. The following sources are available:

- *NewProperties* – creates an empty properties object.

The following transformers are available:

- *GetPropertyNames* – outputs all property names.
- *GetPropertyValue* – outputs the values of properties which key matches a supplied regular expression.
- *PropertiesFileReader* – reads the specific properties file and forwards a properties object.
- *SetPropertyValue* – sets the value of a specified property.

The following sinks are available:

- *PropertiesDisplay* – displays a properties object in a table.
- *PropertiesFileWriter* – writes the properties object to disk.

The following conversion are available:

- *DOMToProperties* – flattens a DOM document object into a properties object.
- *PropertiesToString* – turns the properties object into a string.
- *StringToProperties* – parses the string and generates a properties object.

2.11 XML

ADAMS offers some basic processing for XML. The following sources are available:

- *NewDOMDocument* – creates an empty DOM document.

The following transformers are available:

- *AddDOMAttribute* – adds an attribute and its value to the node passing through.
- *AddDOMNode* – appends a child new to the node passing through.
- *XMLFileReader* – reads the specific XML file and forwards a DOM document[12].
- *XPath* – applies an XPath expression to the incoming DOM document[13].
- *XSLT* – applies an XSLT stylesheet to the incoming DOM document[14].

The following sinks are available:

- *DOMDisplay* – simple tree view of a DOM node.
- *XMLFileWriter* – writes the DOM document to disk.

The following conversions are available:

- *DOMToString* – turns the DOM object into an XML string.
- *DOMToProperties* – flattens the DOM object into a Java Properties object (key-value pairs).
- *DOMNodeToString* – turns the DOM node into an XML string.
- *DOMNodeListToArray* – turns the list of DOM nodes into an array.
- *XMLToDOM* – parses the XML string and generates a DOM object.

[12] `http://www.w3.org/TR/xml/`
[13] `http://www.w3.org/TR/xpath`
[14] `http://www.w3.org/TR/xslt`

2.12 Databases

Any database that has a JDBC driver can be used within ADAMS. By default, ADAMS comes with support for MySQL and SQLite. See section 6.8 for how to add more databases. The following actors allow you to perform some basic operations::

- *ExecSQL* – standalone for executing any SQL query.
- *ListTables* – source actor that iterates through tables in database.
- *SQLIdSupplier* – source actor that either outputs integer or strings, obtained from an SQL *SELECT* query.

For more functionality, see the documentation on the *adams-spreadsheet* module.

2.13 Callable 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: **callable 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 *CallableActors* standalone actor at the start of the flow.
- Add the actor that you want to channel the data into below the *CallableActors* actor that you just added. In our example, this is the *Display* actor.
- Replace each occurence of the actor that you just added below the *CallableActors* actor with a *CallableSink* sink actor. Enter as value for the parameter *callableName* the name of the actor that you added below the *CallableActors* actor. In this example this is simply *Display*.

Figure 2.29 shows the modified flow [15], using a single callable *Display* actor and multiple *CallableSink* actors. In addition to the *CallableSink* actor, there

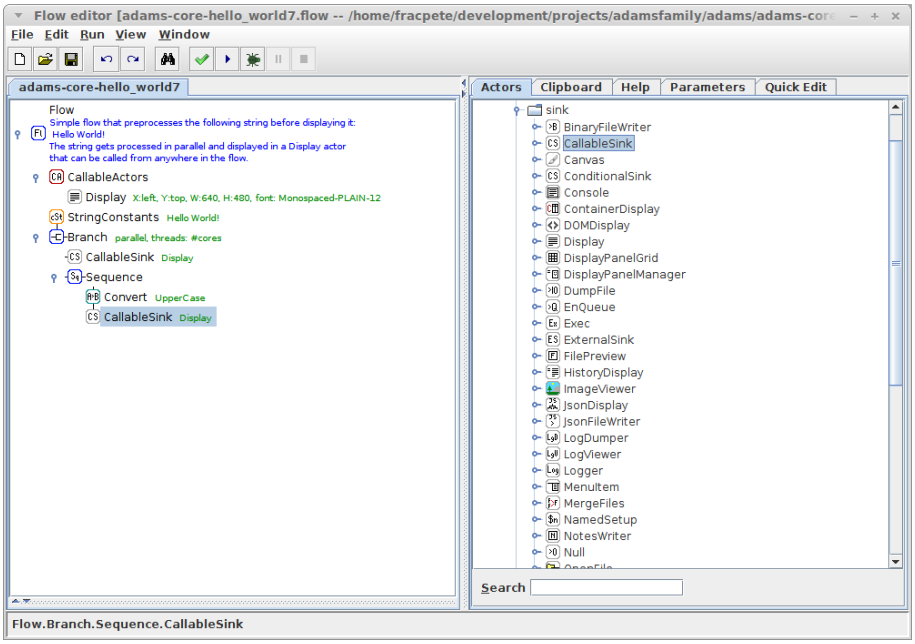


Figure 2.29: Outputting parallel processed strings in a single callable *Display* actor.

are also the *CallableSource* actor (for using the same source multiple times in a flow) and the *CallableTransformer* actor (for instance for using the same preprocessing multiple times). They are used in the same fashion as the *CallableSink* actor that we just introduced.

[15] adams-core-hello_world7.flow

Combining views

The *GridView* standalone allows you to define several graphical actors to be displayed in a grid layout, by adding them below this actor. This can be useful, for instance, if two plots have very different scales and plotting them in the same graph wouldn't make much sense. Using *GridView*, you can create a plot with two rows and one column for displaying the two *SequencePlotter* actors below each other. The actors below the *GridView* actor get referenced from within the flow using *CallableSink* actors.

The *TabView* standalone works just like the *GridView* actor, but instead of displaying the graphical actors in a grid, they get displayed in a tabbed pane (in the order they are below the *TabView* actor).

2.14 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 [16]. 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:

- *ExternalFlow* – for executing complete flows.
- *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.

Once you have an *ExternalXYZ* actor, you can edit this flow directly by selecting the *Edit...* menu item as shown in Figure 2.30.

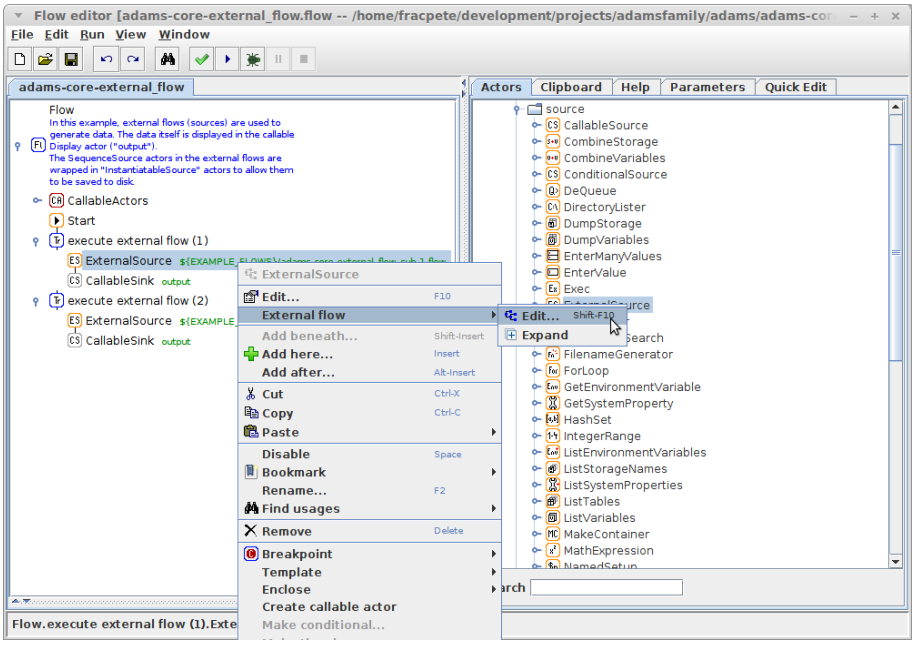


Figure 2.30: Editing an external flow directly.

[16] 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,

Tip: In the flow editor, you can *inline* an external flow by selecting *Expand* from the popup menu on the external actor. This will load the external flow and place it below the external actor as read-only sub-tree (see Figure 2.31). You can simply remove the actors using the *Collapse* option from the popup menu of the external actor.

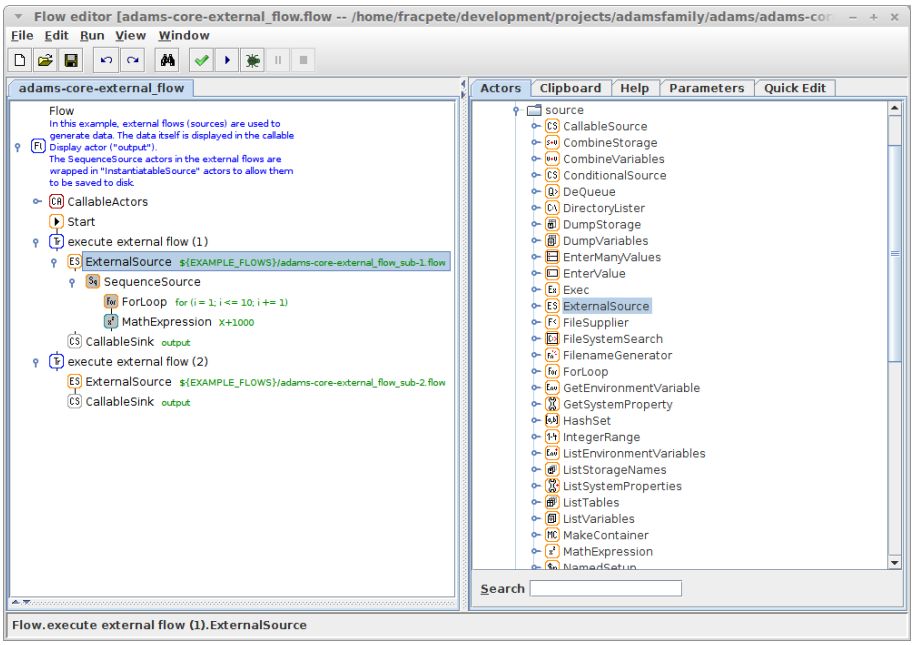


Figure 2.31: An *inlined* or *expanded* external flow.

2.15 Interactive actors

Most of the times, flows that you generate will simply be executed, without any user interaction. However, sometimes user interaction can be very useful in making the flow easier to use. Imagine a flow that takes a file as input, e.g., using the *SingleFileSupplier* source, reads and processes it. If the file varies, you will have to change the source actor each time you want to run the flow with a different file. To address this shortcoming, ADAMS offers a range of interactive actors:

- *EnterValue* (source) – allows the user to enter a value or choose from a range of options.
- *EnterManyValues* (source) – allows the user to enter one or more values, supporting various data types.
- *PasteFromClipboard* (source) – Forwards the content (if any) of the system's clipboard (*CopyToClipboard* allows you to copy textual data to the system's clipboard).
- *SelectDateTime* (source) – pops up a dialog for selecting a date/time, date or time, depending on configuration.
- *SelectDirectory* (source) – pops up a dialog for selecting a directory.
- *SelectFile* (source) – allows the user to select one or more files. File extensions for narrowing down the list of files being displayed is possible as well.
- *ConfirmationDialog* (transformer) – prompts the user with a dialog, offering "yes" and "no" as options. If not custom string tokens are defined for the "yes"/"no" actions, the current token will be forwarded in case of the user selecting "yes" ("no" simply drops the token).
- *Inspect* (transformer) – allows the user to view the content of a token with the specified panel provider, e.g., image viewer.

Using the *SelectFile* source in your flow now, the user won't have to modify the flow anymore. It is now far less likely that the user accidentally modifies and breaks the flow in the process.

The source actors allow you to specify *default* values, e.g., a default directory and default file(s) in case of the *SelectFile* actor. This cuts down the time the user has to spend clicking through directories in the file chooser dialog.

Some of these interactive actors can be switched to "silent" mode, i.e., non-interactive. Counterinuitive as it seems, a rather handy feature when developing a the flow and constant dialogs are simply too annoying. Once development of the flow has finished, the non-interactive setting can be reversed again. You can either set the *nonInteractive* flag for each of these actors manually, or use the flow editor's menu to either turn the interactive nature on or off ("Edit" → "Interactive actors").

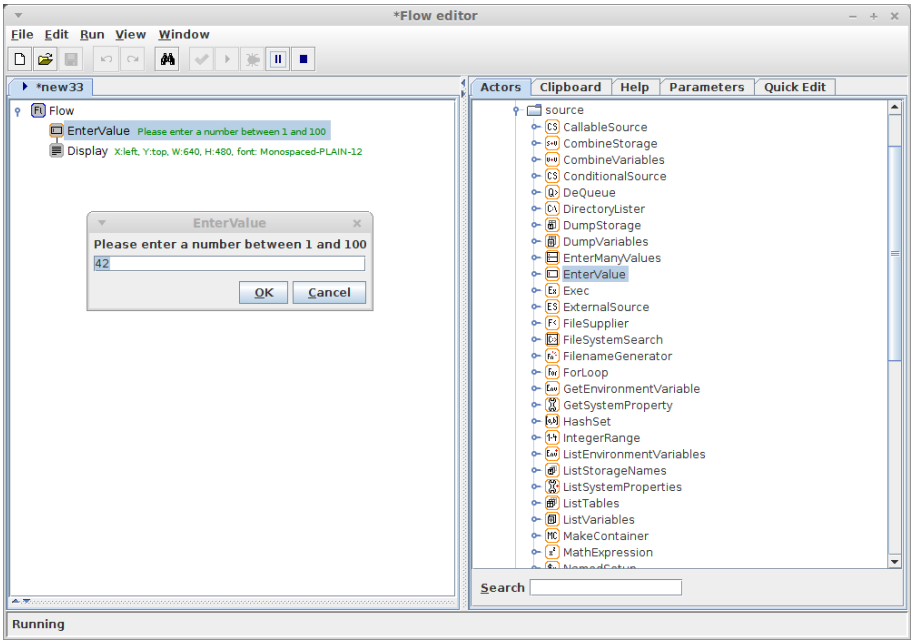


Figure 2.32: Simple flow that prompts the user to enter a value, using a default value of "42" and a custom message.

2.16 Templates

ADAMS comes with a powerful templating mechanism, that speeds up the inception of new flows. Templates allow 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 2.33 to 2.35 show the use of this mechanism.

The "meta-module" module allows you to use these templates also at run-time, dynamically generating flows on-the-fly using special actors.

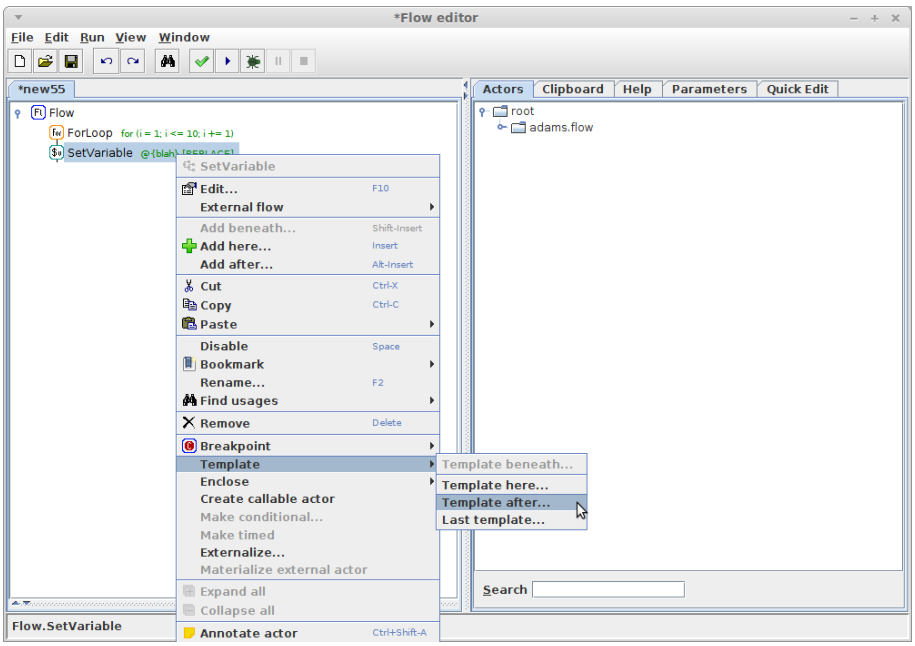


Figure 2.33: Adding a sub-flow generated from a template to an existing flow.

2.17 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

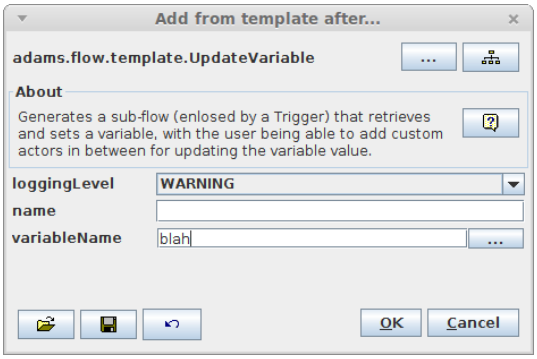


Figure 2.34: The options of the *UpdateVariable* template.

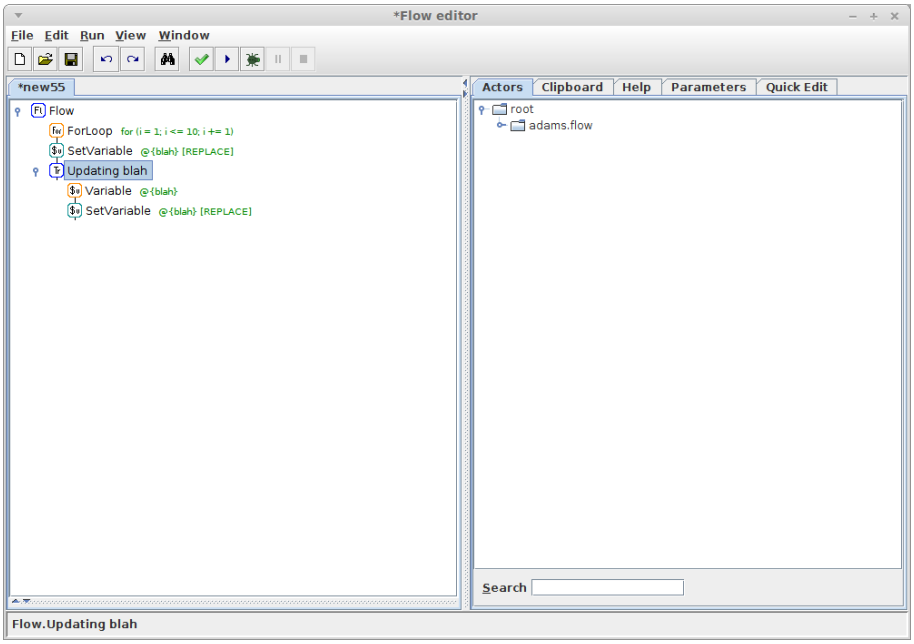


Figure 2.35: The added sub-flow.

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 case of attaching variables to options that are arrays, the variable is expected to be a *blank-separated list* of values.

In the following example [17], 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 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 2.36.

The complete flow is displayed in Figure 2.37. With "quick info" enabled, the *SingleFileSupplier* now also hints that the file it is forwarding is variable-based:

[17] adams-core-variables1.flow

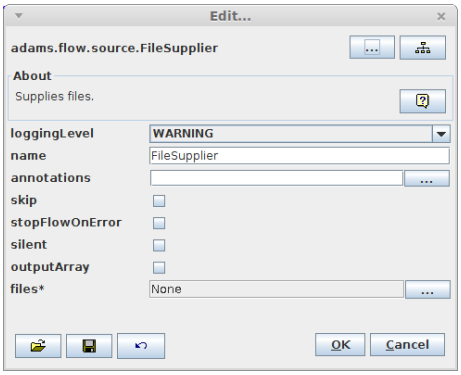


Figure 2.36: The asterisk ("*") next to an option indicates that a variable is attached.

@{*filename*}.

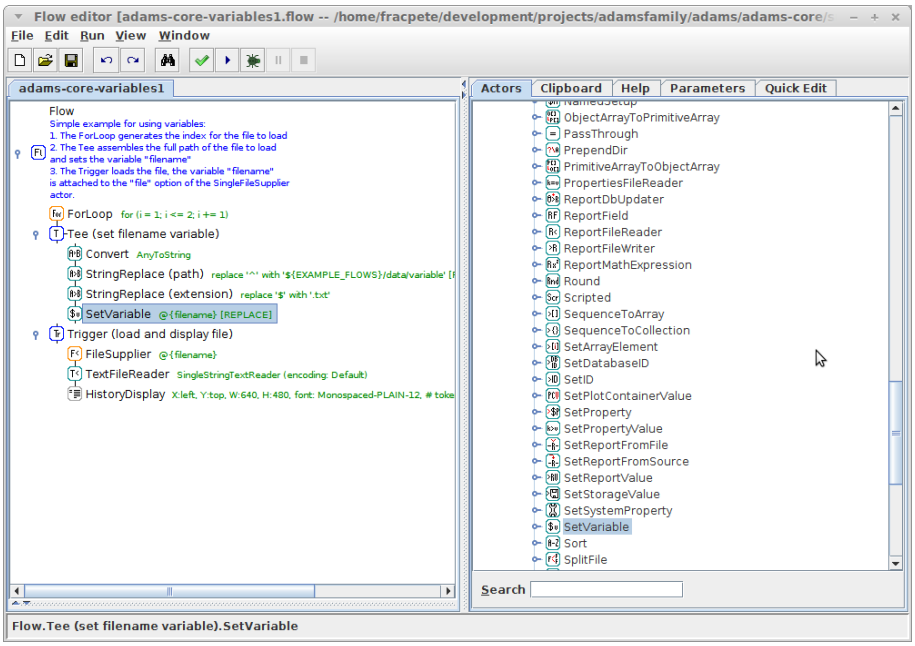


Figure 2.37: Using a variable to control what file to load and display.

The variable mechanism can also be used to dynamically execute another external actor at runtime (see section 2.14 on external actors).

In Figure 2.38 you can see a flow that uses a *ForLoop* to execute three external flows using a variable attached to the *actorFile* option.

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 2.39.

Overview of actors

The following actors are available to handle variables:

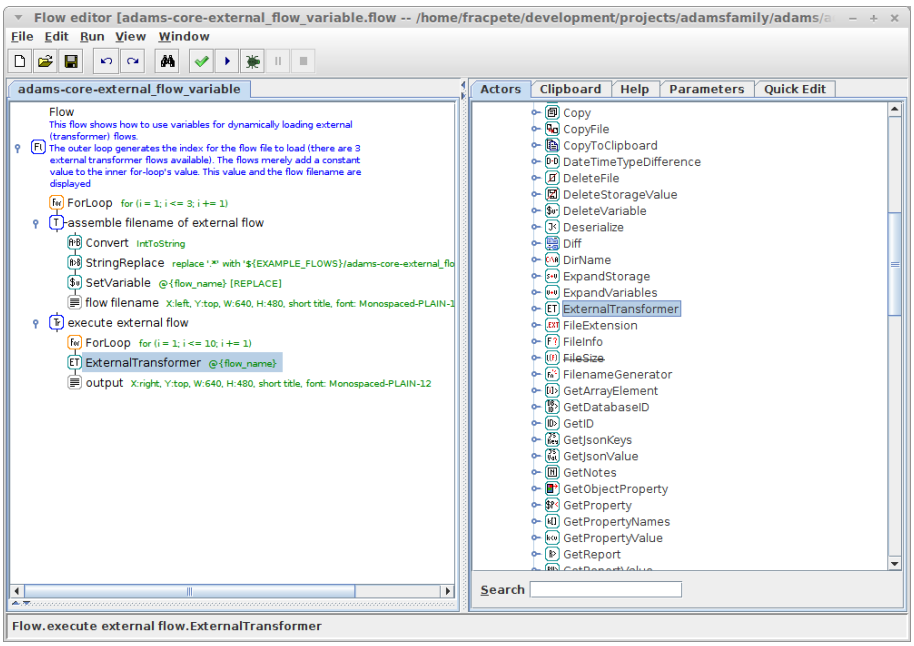


Figure 2.38: Using a variable to control what external flow to execute (flow).

- *ListVariables* – source actor for outputting the names of currently available variables.
- *DumpVariables* – source actor for outputting a spreadsheet with variable names and their associated values.
- *CombineVariables* – source actor for combining one or more variables into a single string using a supplied expression.
- *Variable* – source actor for outputting the value associated with the variable.
- *VariablesArray* – outputs the associated values of several variables as a string array (source).
- *ExpandVariables* – similar to the *CombineVariables* source, this actor expands all variables in the string(s) passing through.

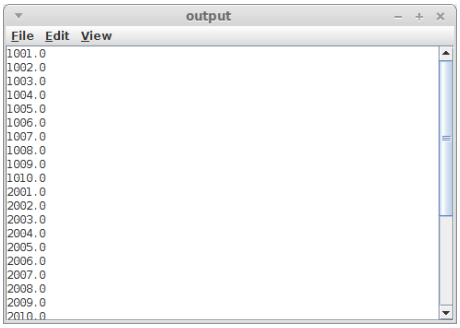


Figure 2.39: Using a variable to control what external flow to execute (output).

- *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).

Using callable actor as variables
One drawback of ADAMS is the absence of multiple inputs, only a single input is supported and containers can mitigate that only to a certain degree. Instead of only using variables values, ADAMS can also harness data generation on-the-fly, by attaching callable actors to options. Attaching a callable actor works just like attaching a simple variable, only the naming convention is different:

```
@{callable:<callable-actor-name>}
```

The `callable:` prefix tells ADAMS that the following name is referencing a callable actor. It then locates the actor and executes it to obtain the value for using with the option in question.

This mechanism has mainly three caveats:

- The callable actor (or sequence of actors) gets executed whenever the actor, which has one or more callable actor references attached, gets executed. Depending on the actors in use, this can be rather computationally expensive.
- Since the variable mechanism has no notion of *where* in the flow it is, only callable actors that are defined below the *Flow* actor can be used (kind of "super callable actors").
- The callable actor should generate the correct type for the option it is attached to. Otherwise, the value gets converted and parsed as a command-line value. Though the flow will be able to recover, this will slow things down as an exception will get output whenever this occurs.

Using storage items as variables
Similar to attaching callable actors, storage values can be attached like variables as well. Once again, a custom prefix, "storage", is used to distinguish these special variables from ordinary ones:

```
@{storage:<storage-value-name>}
```

The referenced object is then obtained from storage and set.

This mechanism has two caveats:

- The storage value is retrieved whenever the actor, which has one or more storage value references attached, gets executed.
- The storage value needs to have the correct type for the option it is attached to. Otherwise, the value gets converted and parsed as a command-line value. Though the flow will be able to recover, this will slow things down as an exception will get output whenever this occurs.

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 [18] 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 handle properties:

- *GetProperty* – transformer that retrieves the current value of a property of the token passing through.
- *SetProperty* – transformer that modifies a single property of a callable actor using the string representation of the token passing through.
- *UpdateProperty* – transformer that modifies a single property of an object passing through using the specified value.
- *UpdateProperties* – allows you to update multiple properties (each property is associated with a particular variable) of the actor that this actor manages.

Special variables

Often, flows use resources that are relative to the flow itself. In order to make this easier, there are two special variables available at runtime:

- *flow_dir* – stores the path of the flow
- *flow_filename_long* – stores path and file name of the flow
- *flow_filename_short* – stores only the file name of the flow

[18] See `http://download.oracle.com/javase/tutorial/javabeans/introspection/` for more information on Java Introspection.

2.18 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 2.40 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 2.41.

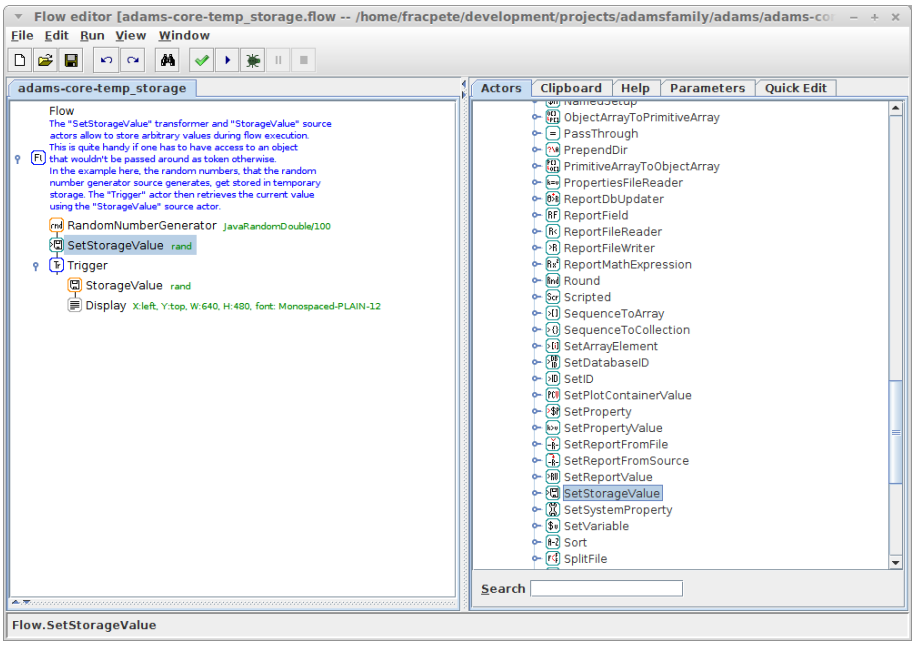


Figure 2.40: Flow demonstrating the temporary storage functionality.

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

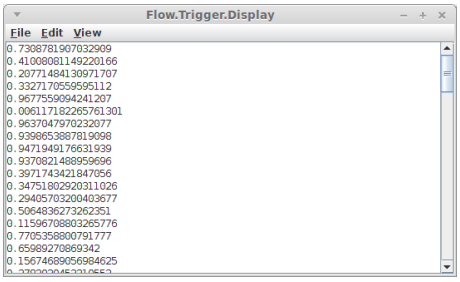


Figure 2.41: Output of flow demonstrating the temporary storage functionality.

encourage re-generation of values on demand, the storage system also offers least-recently-used (LRU) caches [19]. 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 2.42 shows the use of the LRU cache functionality, with Figure 2.43 displaying a snapshot in time of the storage inspection panel available through the *Breakpoint* control actor. Finally, Figure 2.44 shows the final output of the flow.

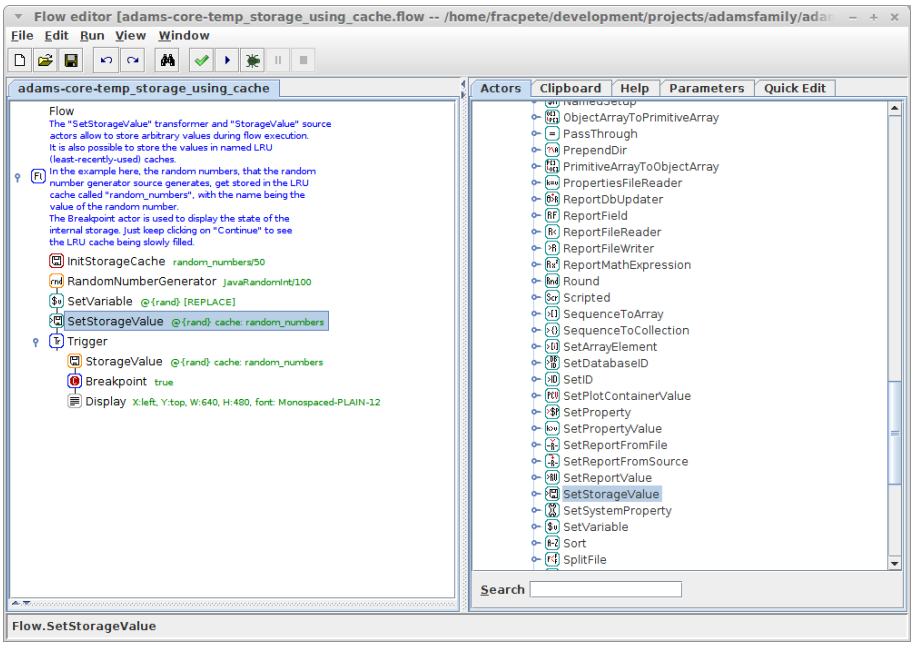


Figure 2.42: Flow demonstrating the LRU cache storage functionality.

Overview of actors

The following actors are available to handle variables:

[19] See `http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used` for more information on LRU caches.

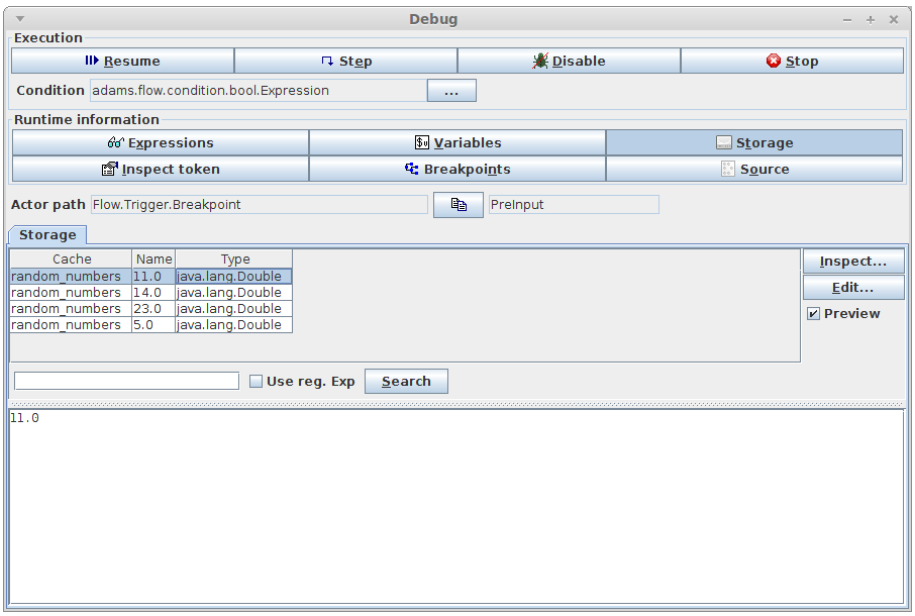


Figure 2.43: Display of the temporary storage during execution.

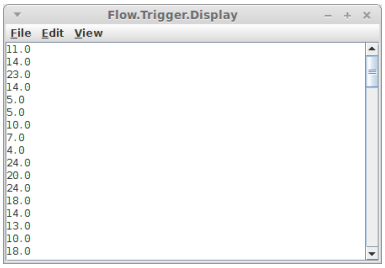


Figure 2.44: Output of flow demonstrating the LRU cache storage functionality.

- *InitStorageCache* – standalone actor for initializing a named LRU cache with a specific size.
- *CombineStorage* – source actor for combining string representations of storage items and variables into a string.
- *ExpandStorage* – similar to the *CombineStorage* source, this actor expands all storage items and variables in the string(s) passing through.
- *GetStorageValue* – transformer that replaces the incoming string with the storage value that the string represents. associated with the specified name.
- *ListStorageNames* – source actor for outputting the names of the currently stored items.
- *DumpStorage* – source actor for outputting the names of the currently stored items and their associated values (in string representation) in a spreadsheet.
- *StorageValue* – source actor for outputting the storage value associated

with the specified name.

- *StorageValuesArray* – source actor for outputting the storage values associated with the specified names as an array.
- *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.

2.19 Debugging your flow

2.19.1 Breakpoints

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 a *Breakpoint* actor is reached and the condition evaluates to *true*, the debugging control panel will get displayed (see Figure 2.45).

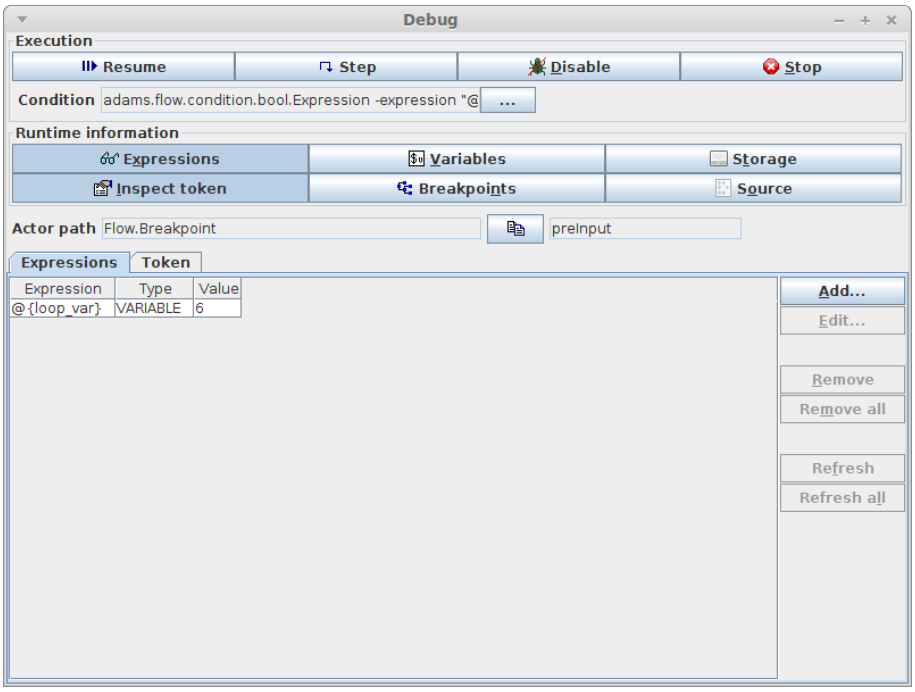


Figure 2.45: The debugging control panel.

The functionality of the control panel is best explained with an example. The flow [20] in Figure 2.46 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.

[20] adams-core-breakpoint.flow

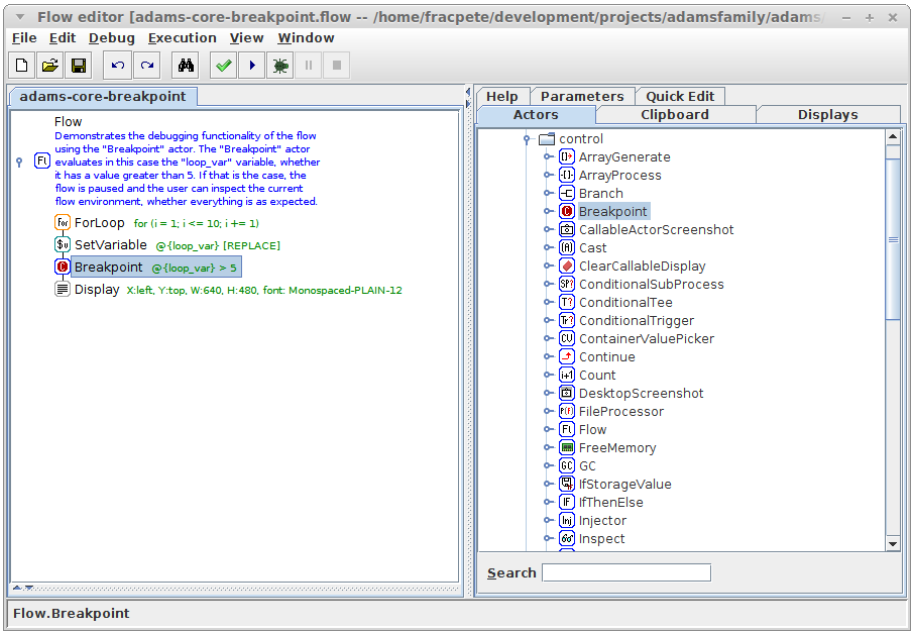


Figure 2.46: Example flow with *Breakpoint* actor.

- The buttons in the *Execution* group allow you to resume the flow execution, step through it, actor by actor, stop the flow or simply disable the current breakpoint.
- It is possible to update the breakpoint condition whenever the breakpoint is reached.
- 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 2.45), 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 passed through the breakpoint (Figure 2.47) and also all breakpoints currently present in the flow (Figure 2.48). Breakpoints can be disabled and enabled, modified when to stop.

While a flow is running, you have some basic tools for inspection at hand as well (available from the *Run* menu):

- *Variables* – allows you to monitor the variables of a flow and how they change. Note that updating the dialog is quite expensive and will slow down your flow considerably.
- *Storage* – when a flow is paused, you can inspect the current storage items; as soon as you resume the flow, the dialog will disappear again, as it doesn't get refreshed automatically.

A breakpoint can be triggered at various stages of executing an actor:

- pre/post input of a token (transformers, sinks)

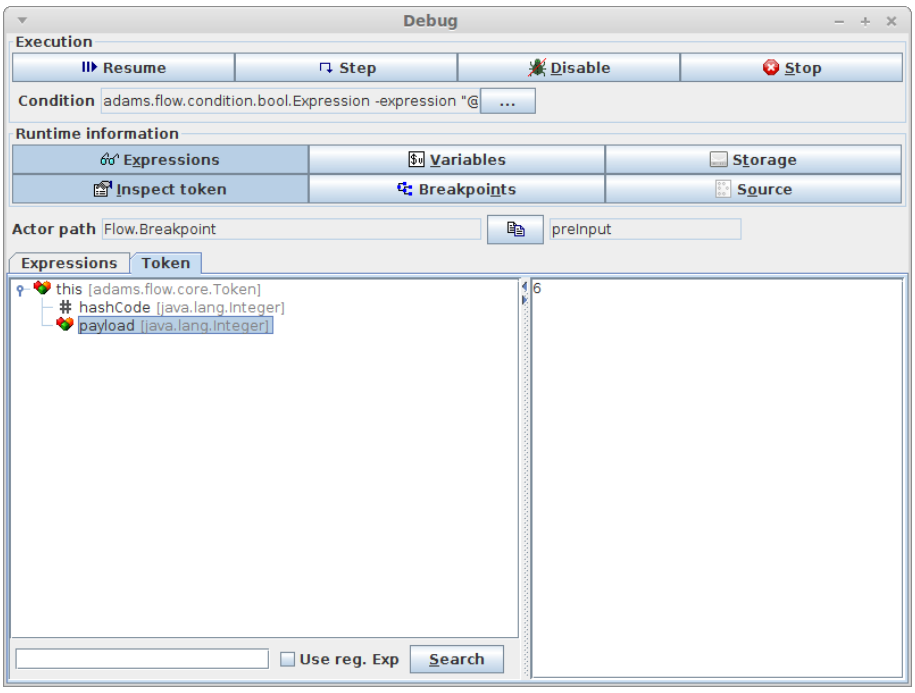


Figure 2.47: The *Inspection* tab of the debugging control panel for the current token.

- pre/post execute
- pre/post output of a token (sources, transformers)

Inspecting a token is only available at stages *preInput*, *postOutput* and possibly at *preExecute*, if the actor is a transformer or sink.

Stepping through a flow, using the *Step* button from the control panel, always uses the *preExecute* stage (*pre/postInput* are not available, unfortunately due to API limitations).

2.19.2 Monitoring

With ADAMS it is possible to *eavesdrop* on the flow execution by attaching a so-called *flow execution listener* to the *Flow* actor and enable the listening process there as well.

The following listeners are available:

- *CurrentlyExecuted* – displays all the actors (with their start times) that are currently being executed.
- *Debug* – the *Breakpoint* control actors use this listener, setting the appropriately configured breakpoints.
- *ExecutionCounter* – counts for each actor how often it was executed.
- *ExecutionLog* – writes all calls to the input, execute and output methods to a log file.
- *MultiListener* – allows you to listen with multiple listener setups to the flow execution.

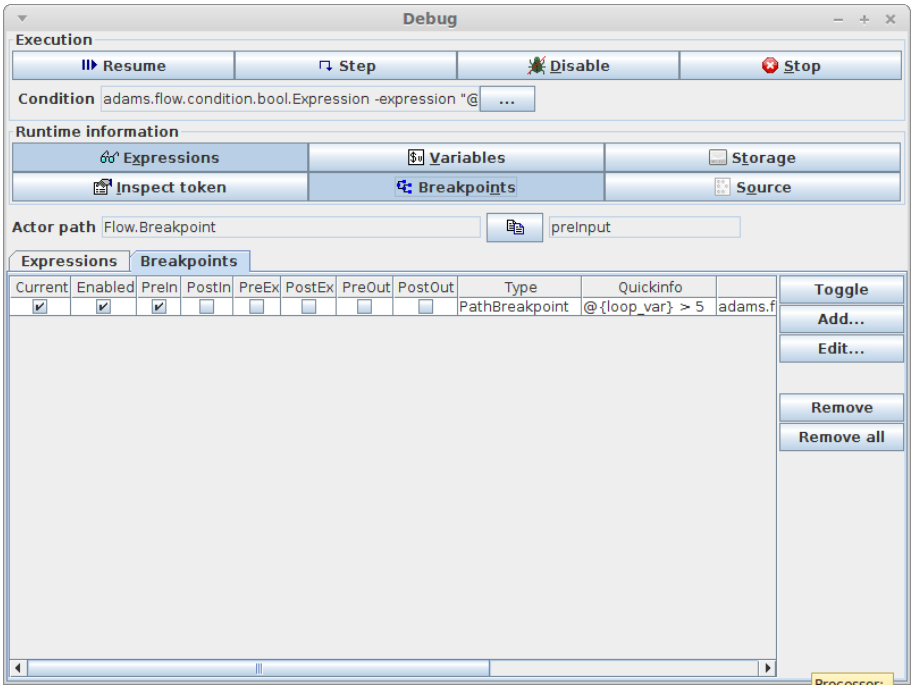


Figure 2.48: The *Breakpoints* tab of the debugging control panel.

- *NullListener* – dummy listener, does nothing.

2.20 Passwords

In various places, ADAMS requires the use of passwords, for instance, when connecting to databases. ADAMS does not offer any proper encryption of the passwords, merely a weak obfuscation using Base64[21] encoding. Keep this in mind when designing flows and making the available to other people.

[21] http://en.wikipedia.org/wiki/Base64

2.21 External processes and classes

ADAMS allows you to start up external processes or call Java classes that are present in the classpath from within a flow. The following actors are available:

- *Java* – standalone that calls the *main* method of a Java class, using the current JVM.
- *JavaExec* – standalone that starts up a new JVM using the current classpath and JRE. stdout and stderr can be further processed in the flow.
- *Exec* – source that calls any external executable and allows to further process either stdout or stderr.

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.

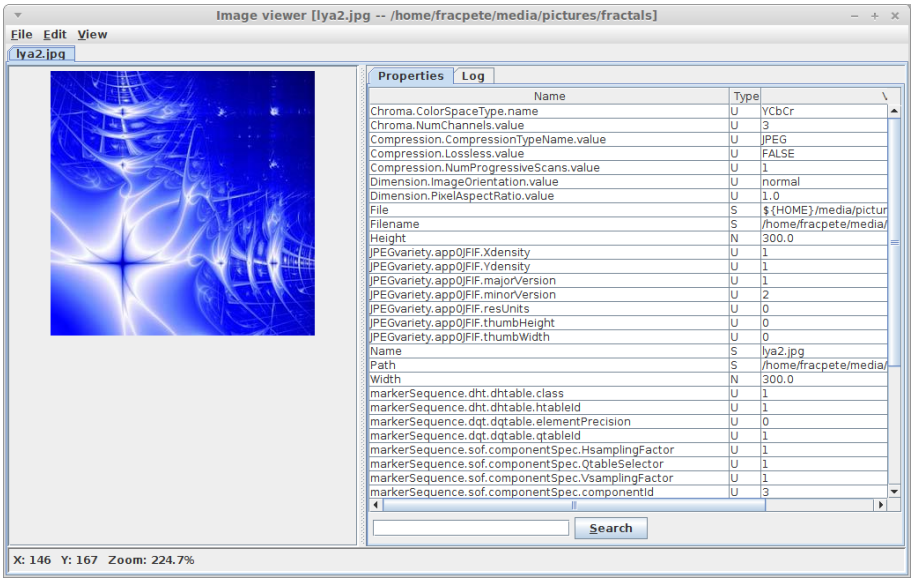


Figure 3.1: Displaying a fractal in the Image viewer.

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.

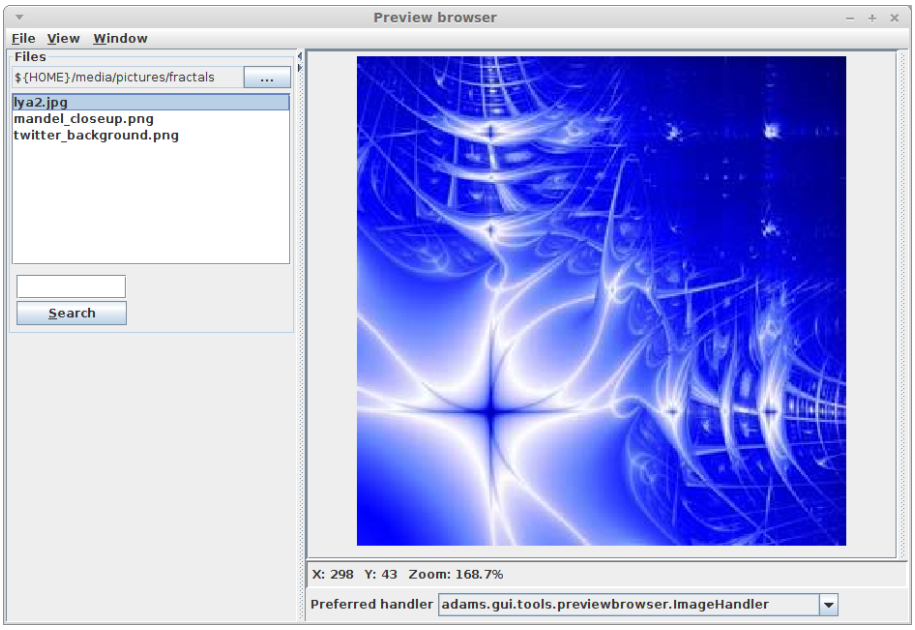


Figure 3.2: Preview browser displaying an image.

Serialized files can be inspected as well, e.g., for model files generated by WEKA. Other modules may offer specific viewers for the objects stored in such a file.

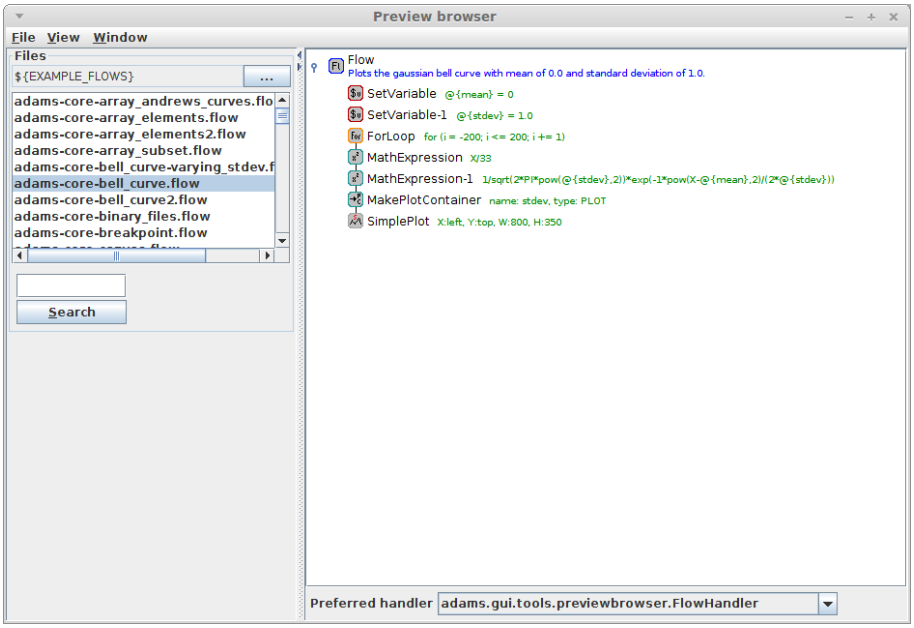


Figure 3.3: Preview browser displaying a flow.

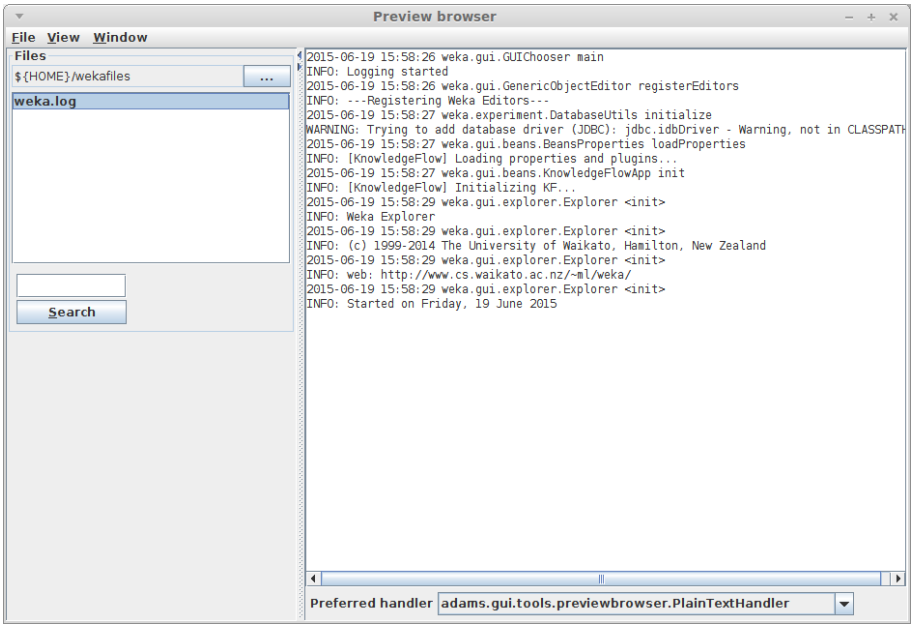


Figure 3.4: Preview browser displaying an plain text file.

Chapter 4

Tools

Among the items in the *Tools* menu are the most important tools of ADAMS, the interfaces for editing and running flows.

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

The *Flow runner* is an interface to execute flows without the user being able to modify them. See chapter 2.3 for more information.

4.3 Actor usage

The class *adams.flow.core.ActorUsage* allows you to generate a spreadsheet that generates an overview of which actors are used in what flows. Here is an example command-line for Linux:

```
adams.flow.core.ActorUsage \
  -dir ./flows \
  -recursive \
  -no-path \
  -output $HOME/actors.csv \
  -logging-level INFO
```

The command looks recursively for flows, starting in the *./flows* directory. The generated output, omitting the path from the flow files, is written to *actors.csv* in the user's home directory.

Rather than using a static spreadsheet, you can also use this tool from the main menu: *Help -¿ Actor usage*. After selecting a directory containing flows (which will get searched recursively), you will be presented with a dialog that displays the generated spreadsheet (see Figure 4.1). This dialog also allows you

to select one or more flow files and then edit them, by clicking on the *Edit* button. You can directly edit a single flow by double-clicking on it in the table as well.

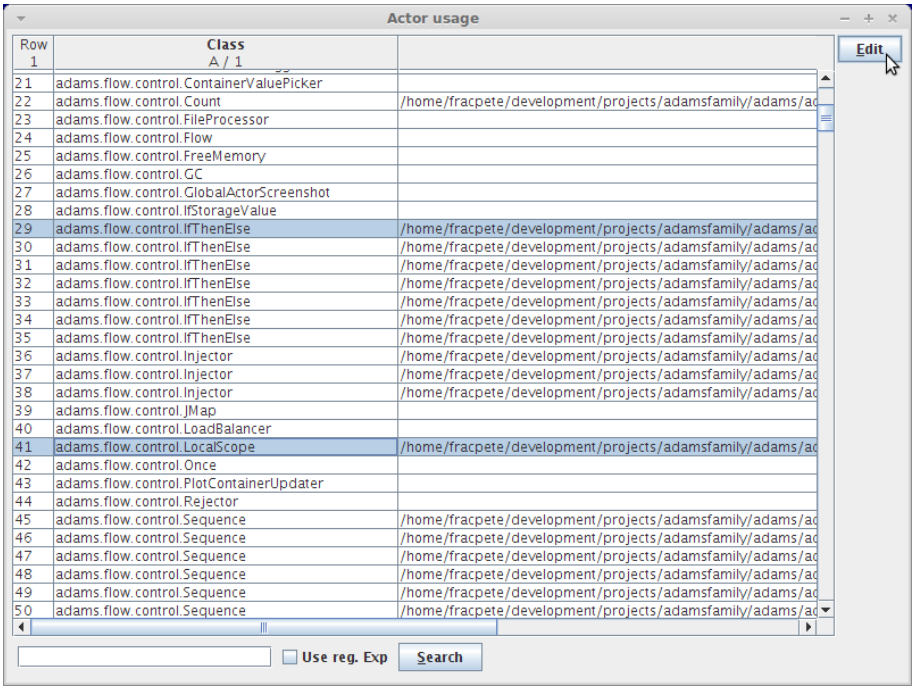


Figure 4.1: Overview of actor usage in flow files.

4.4 Text editor

The *Text editor* is a very simply editor for plain text files. It allows you to view and edit one file at a time. It also supports printing and, if the *net* module is present, sending the files via Email.

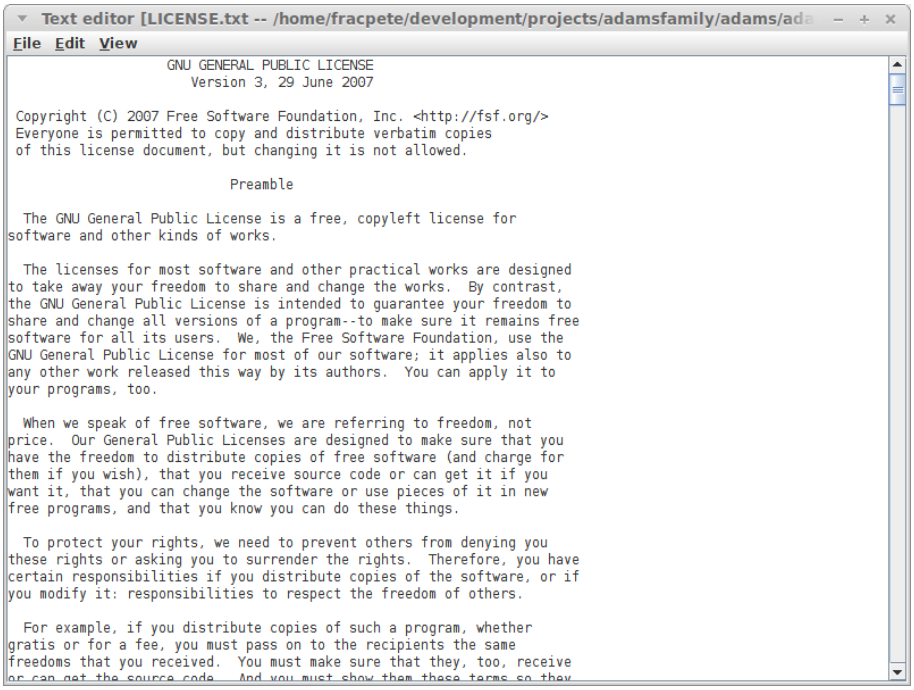


Figure 4.2: Editor for viewing/editing plain text files.

4.5 Comparing text

The *Comparing text* tool allows you two compare two files or content pasted from the clipboard (using the *paste* buttons at the bottom) or a mixture of both. Figure 4.3 shows the open dialog and the comparison of the files in the background (*red* depicts changes between the files, *blue* a deletion and *green* an addition).

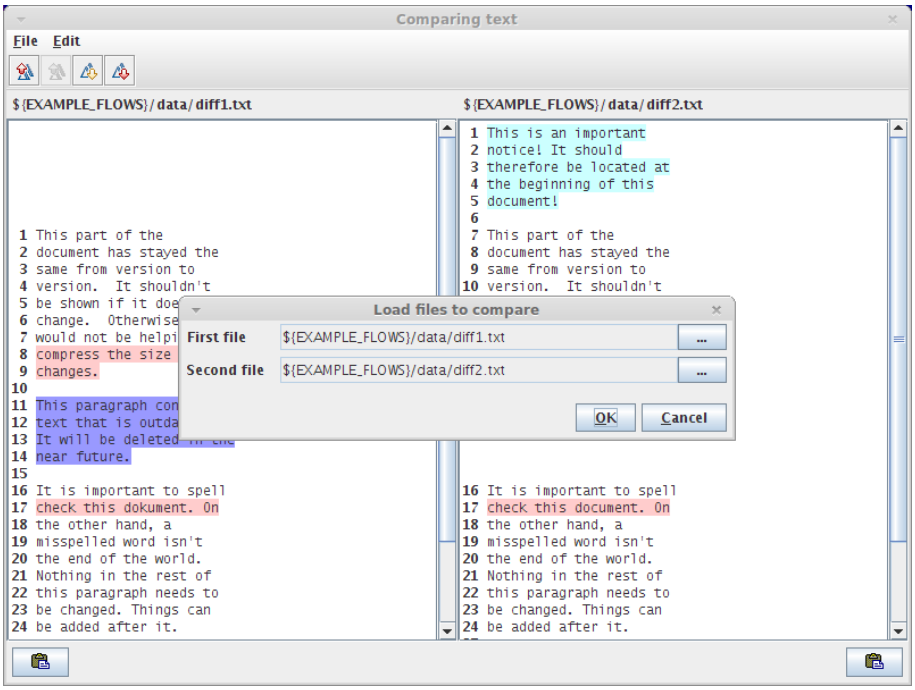


Figure 4.3: Comparing two text files.

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

Whenever file names are being used in a flow, you run the danger for making your flow only executable on your own machine. In order to make it easy to use flows on multiple computers with different directory structures, ADAMS introduces the concept of *placeholders*. Placeholders are basically system-wide defined variables for directories. This allows you to define a placeholder called *XYZ* and point it to directory */some/where* on computer 1. On computer 2, on the other hand, you point it to */somewhere/completely/different*. As long as the directory structure below this placeholder is the same, the flow is guaranteed to work.

In Figure 5.1 you can some see placeholders already defined. Here, the placeholders are used for example flows for various presentations.

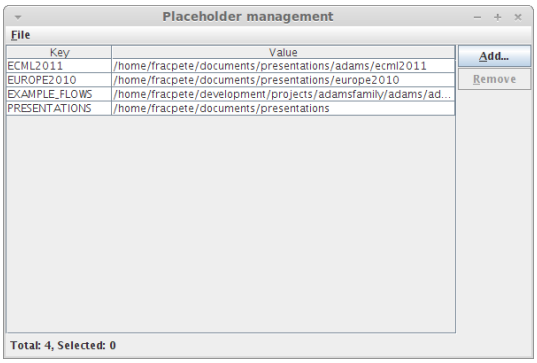


Figure 5.1: Viewing the currently defined placeholders.

Adding placeholders
In order to add a new placeholder, you need the following two steps:

1. Add the name for the new placeholder, e.g., *TEST* (see 5.2)
2. Add the directory that this placeholder is to represent (see 5.3).

After you added this placeholder the management panel will look as shown in panel 5.4. In order to make the changes persisten, you need to save the changes (see 5.5) and restart the application.

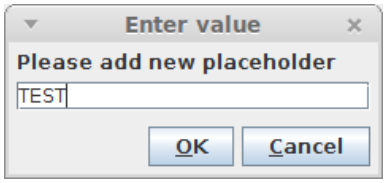


Figure 5.2: Entering the name for a new placeholder.

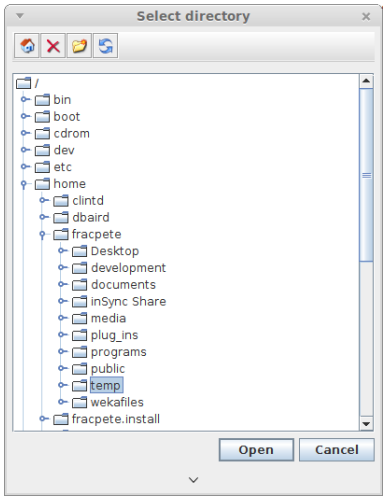


Figure 5.3: Selecting the directory that the new placeholder represents.

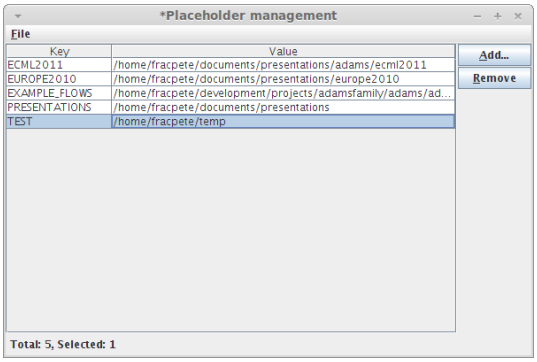


Figure 5.4: The updated view of the placeholders.

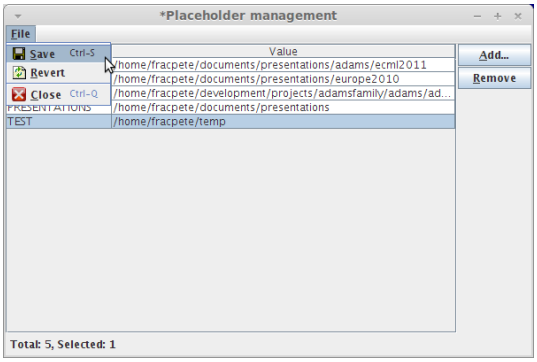


Figure 5.5: Making the placeholder changes persistent.

Editing placeholders
By double-clicking on a cell, you enter the edit mode of the cell and you can either change the name of the placeholder or the path. Figure 5.6 shows the latter.

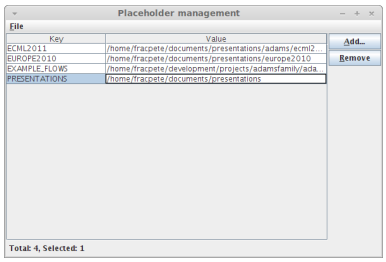


Figure 5.6: Editing the path of a placeholder.

Double-clicking a second time on the path, while you are in edit mode, you can bring up a dialog for selecting a directory (see 5.7). This is less error prone than manually entering the path. Of course, after you have updated a placeholder, you need to make these changes persistent again by saving the configuration and restarting the application.

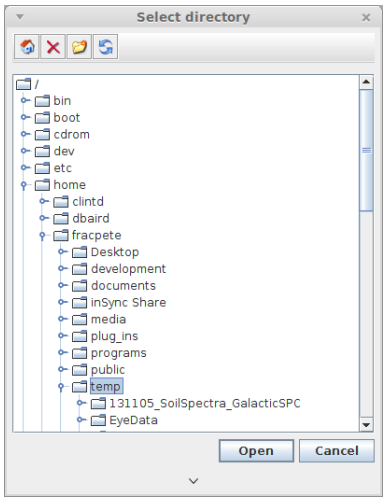


Figure 5.7: Selecting the new directory that the placeholder should represent instead.

5.2 Named setup management

ADAMS allows you to define setups of, e.g., filters that can be referenced then by their name, hence *named setup*. In case of filters, this would happen by using the special filter *NamedSetup*.

In Figure 5.8 you can see the currently defined setups of a test system – your view might look different.

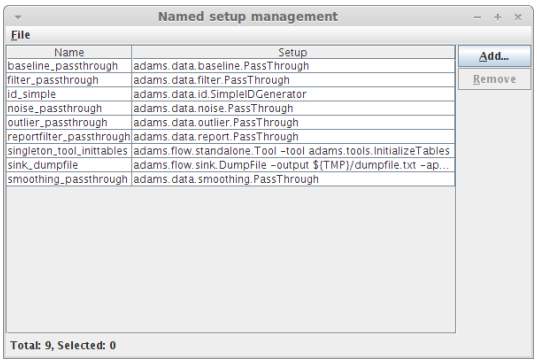


Figure 5.8: Viewing the currently defined named setups.

Adding a new named setup is a three-stage process: first, you select the class hierarchy (see 5.9); second, you select and configure the actual setup that you want to reference (see 5.10); third, add the *nickname* for this setup (see 5.11). This will update the main view as shown in Figure 5.12. In order to make these changes persistent, you need to save them by selecting the *Save* menu item from the menu of the management panel.

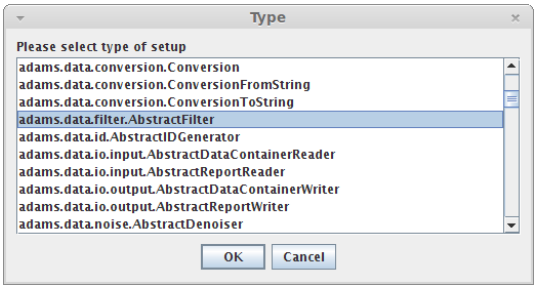


Figure 5.9: The class hierarchy for the named setup.

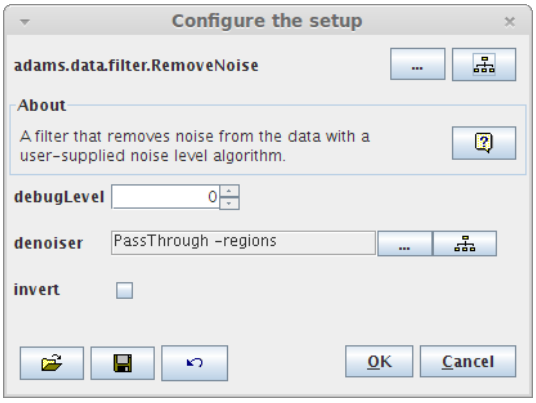


Figure 5.10: Selecting the configuration that the new named setup represents.

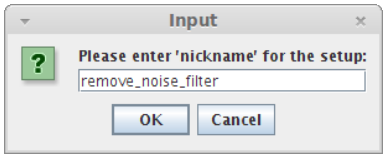


Figure 5.11: The *nickname* for the setup.

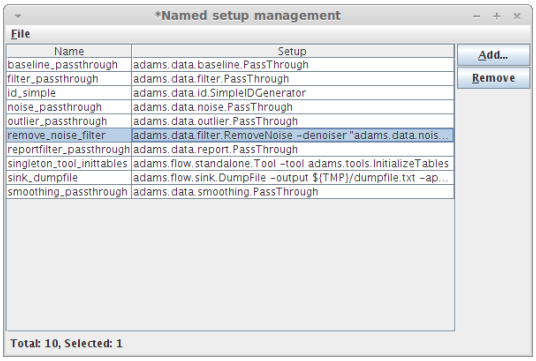


Figure 5.12: The updated view of the named setups.

5.3 Favorites management

The last thing you want to do, is wasting time on configuring the same setup in, e.g., the object edit over and over again. Figure 5.13 shows how to use the *favorites* mechanism for selecting a favorite, replacing the currently displayed object in the object editor completely. Figure 5.14, on the other hand, shows how to add the setup from a property as a new favorite, using the right-click menu of the property. Favorites get grouped by the superclass they belong to in the object editor.

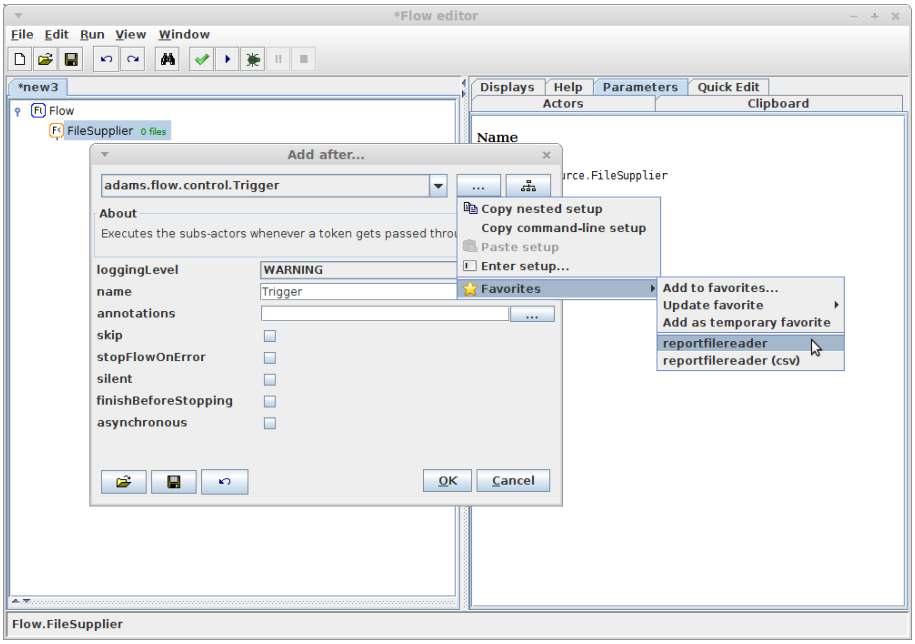


Figure 5.13: Making use of a favorite in the Flow editor.

ADAMS distinguishes between *permanent* and *temporary* favorites. The latter are only available in the current session and won't get stored on disk (in *GenericObjectEditorFavorites.props*). They are considered more of an extended clipboard.

Of course, ADAMS comes with a management interface for maintaining all the various setups, allowing you to store, edit, rename and remove (named) configurations.

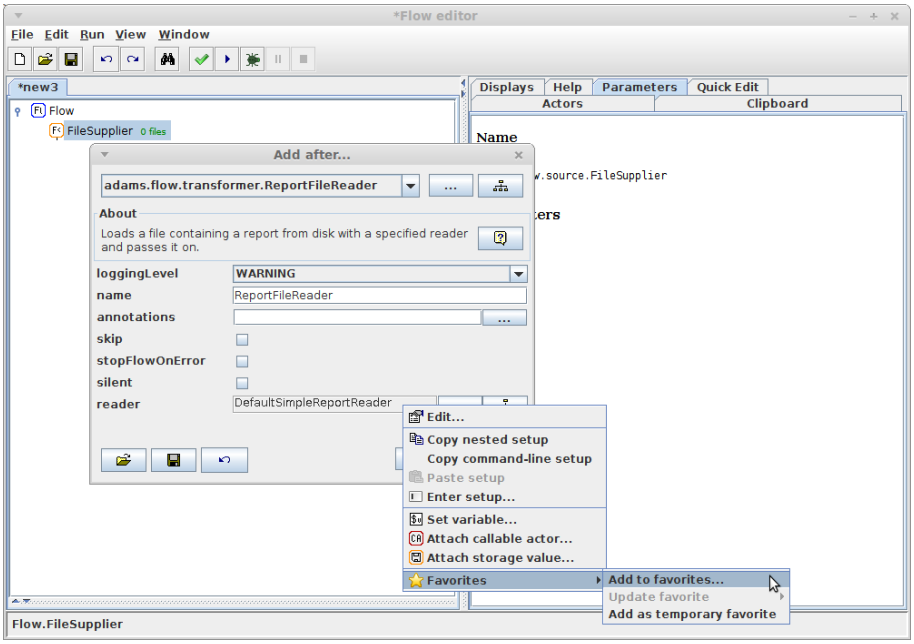


Figure 5.14: Adding a setup in the object editor to the favorites.

Adding a favorite

When starting from scratch with the favorites, then it will most likely be the case that you haven't got a superclass group yet in your favorites that you want to add the new favorite to. In that case, you need to *Add* a superclass on the left side first (see 5.15). This automatically pops up the dialog then that allows you to configure a favorite for this superclass (see 5.16). Accepting the configuration will prompt you with a dialog requesting a *name* for the favorite (see 5.17). Once this is done, the view is refreshed as seen in Figure 5.18.

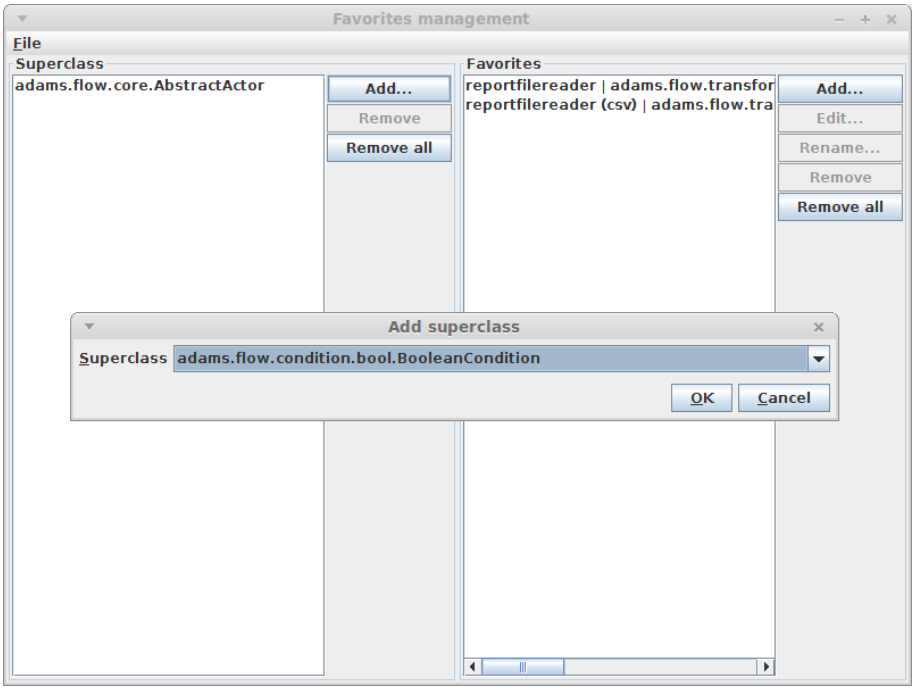


Figure 5.15: Adding a favorite for new superclass.

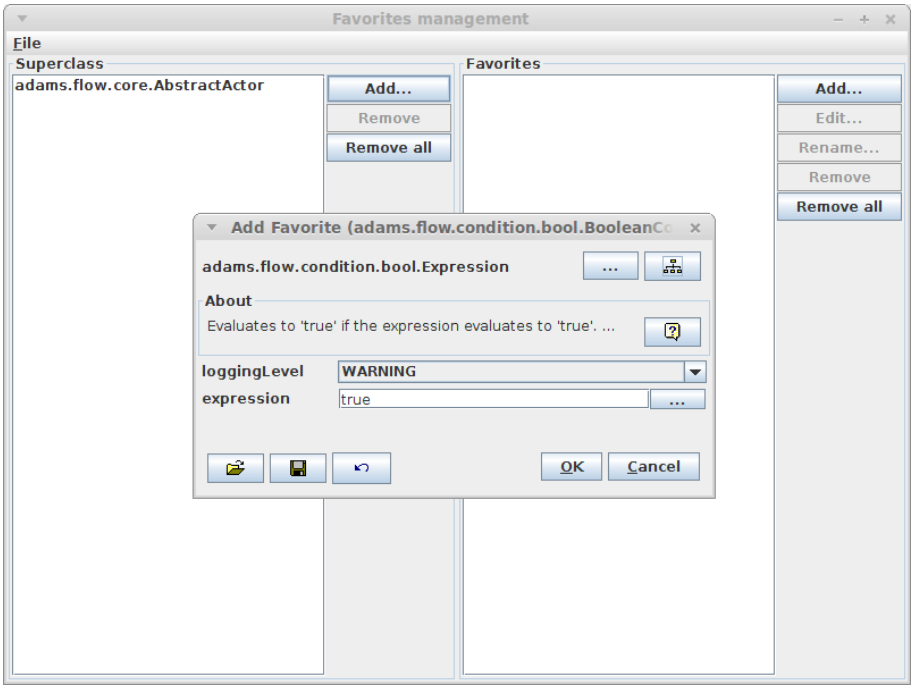


Figure 5.16: Configuring the new favorite.

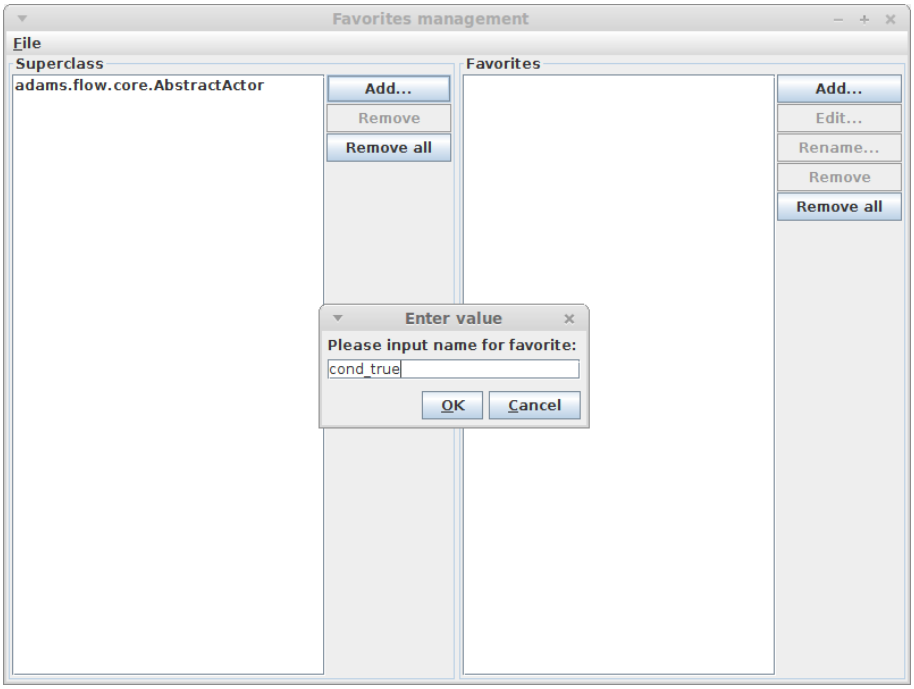


Figure 5.17: Naming the favorite.

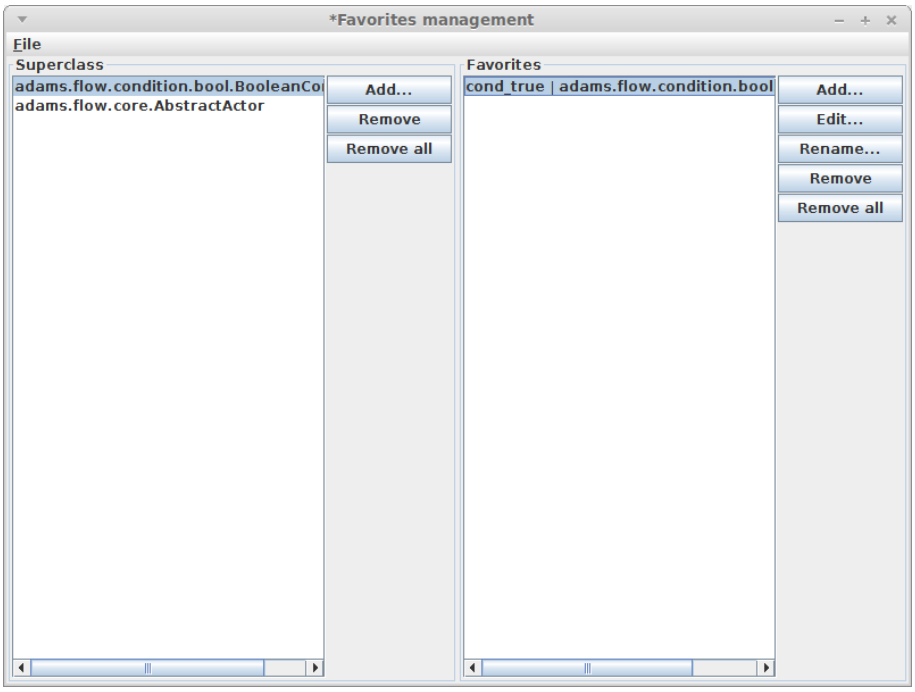


Figure 5.18: The updated favorites view.

Editing a favorite

It is not uncommon that favorites can change slightly (or even more drastic) over time. Being able to update the setups is therefore important. By selecting an existing favorite on the right-hand side and clicking on *Edit*, you can change the existing setup (see 5.19). If the new setup is accepted, the view gets refreshed and the new setup is being displayed as shown in Figure 5.20.

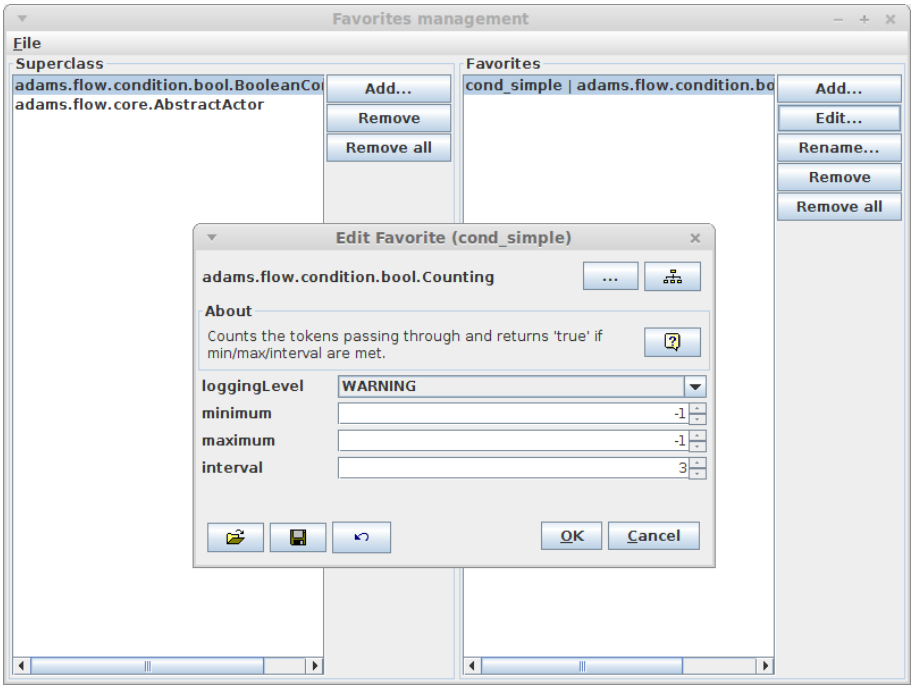


Figure 5.19: Changing a different setup for a favorite.

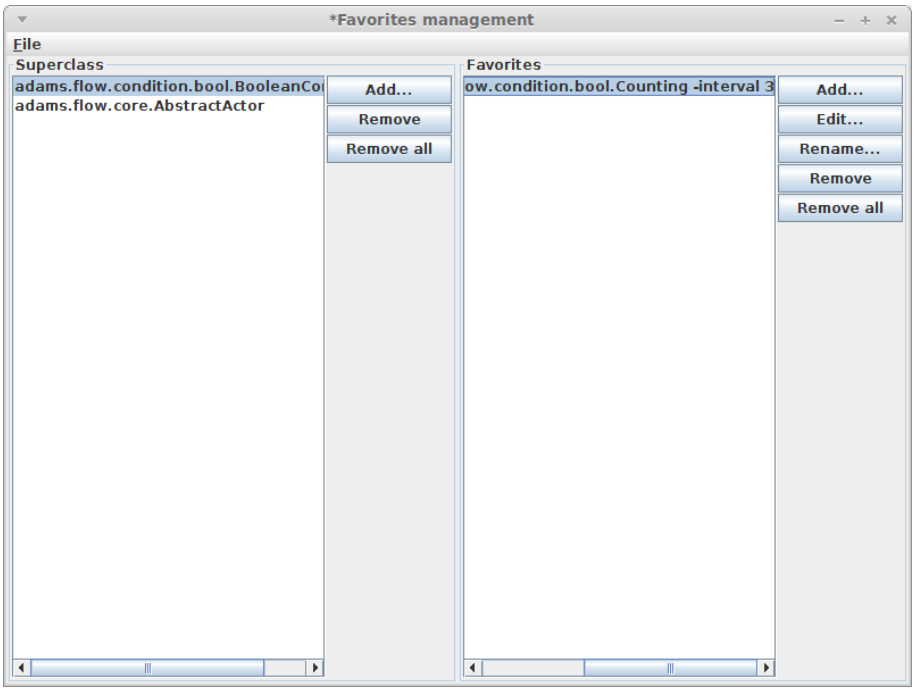


Figure 5.20: The view with the updated favorite.

Renaming a favorite

With the number of favorites growing or simply updating them, it can happen that renaming of one or more favorites is required. By selecting a favorite on the right-hand side, you can click on *Rename* to bring up a dialog for the new name (see 5.21). If this dialog is confirmed, the view gets refreshed and the renamed favorite is being displayed as shown in Figure 5.22.

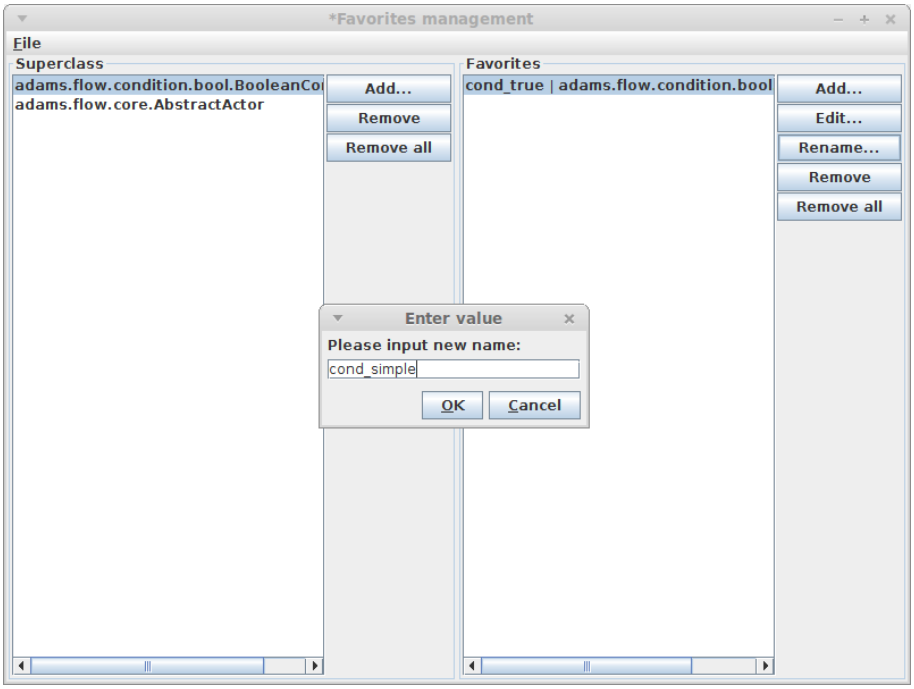


Figure 5.21: Choosing a new name for the favorite.

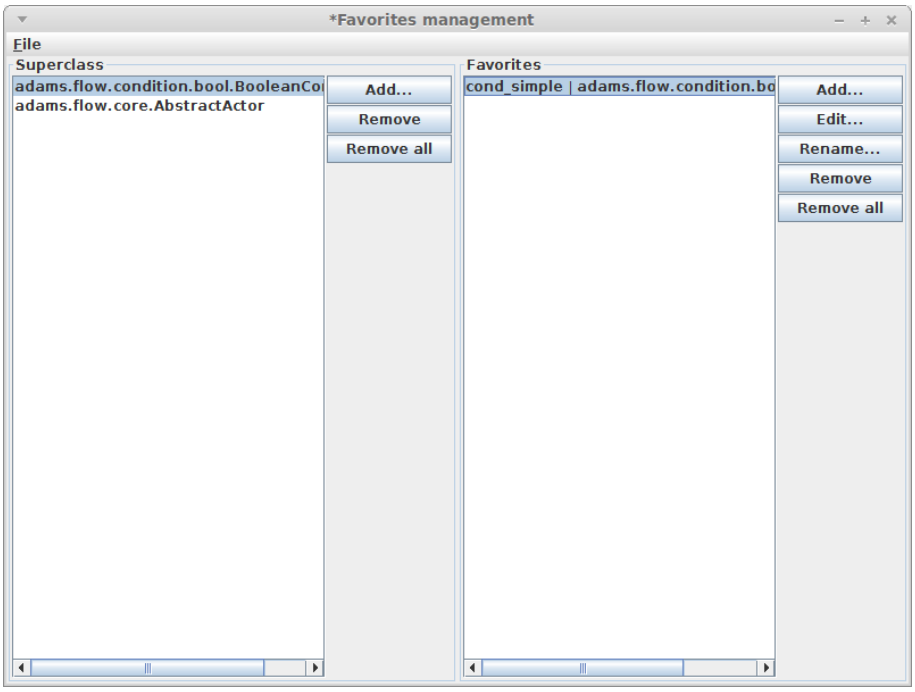


Figure 5.22: The view with the renamed favorite.

Saving the favorites

Of course, in order to make the changes permanent, you have to save them to disk. You can do this by selecting *File* → *Save* from the menu as shown in Figure 5.23.

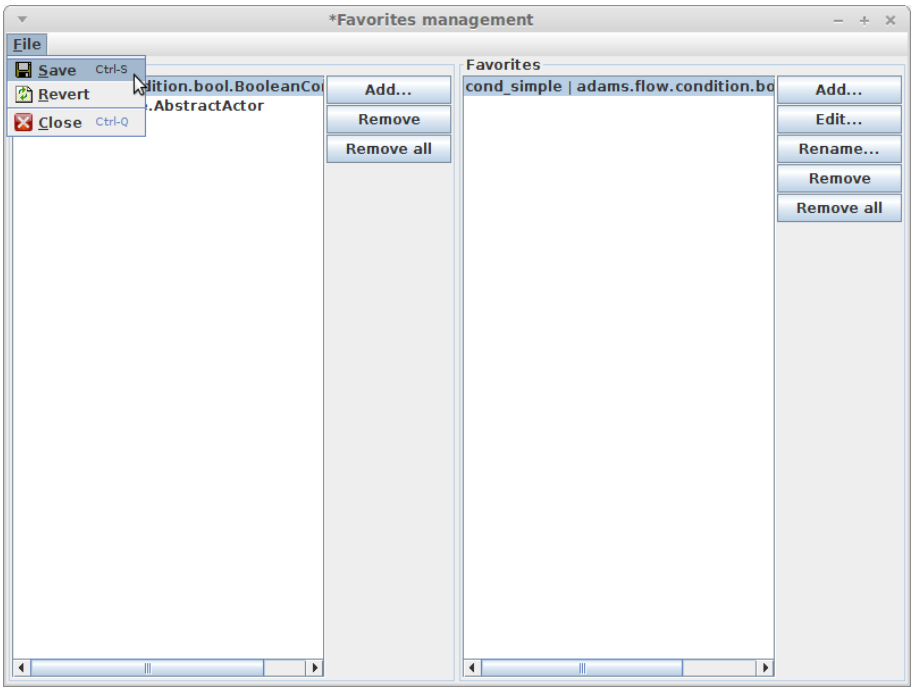


Figure 5.23: Saving the modified favorites.

Chapter 6

Customizing ADAMS

Though ADAMS may lack somewhat preference dialogs in the user interface, it nonetheless allows you to customize a lot of the behavior and the way things are displayed using properties files or environment variables. The following sections explain the basics of how this customization works and goes into detail for some of the user interfaces in ADAMS.

6.1 Environment variables

The following variables are recognized:

- `ADAMS_OPTS` – Instead of supplying command-line options, you can also set the options using this variable. For instance, to always use the expert menu mode, use the following:
  `ADAMS_OPTS=-user-mode EXPERT`

6.2 Properties files

A properties file is a plain text file that ends with the extension `.props`. Each properties file contains key-value pairs that are separated by an equals sign ("="). A backslash at the end of a line can be used to break up long lines and continue on the next one. The default setup is defined in the file present in the jar archive. But you can override this behavior in two places: your home directory (`$HOME/.adams` for *nix and `%USERHOME%\adams` for Windows) and the current directory that the application is executed from. The current directory approach, if ADAMS is installed in a directory accessible to all users, can be used to define system-wide configurations. The home directory approach, on the other hand, is for user-specific customizations (e.g., preferred keyboard shortcuts). Overriding a properties file works by simply creating a new file with the exact same file name (case-sensitive) and providing a new value for a key that exists in the default properties files. Only the file name needs to be the same, you do not need to create a directory structure in the home or current directory. Here is an example: if you want to override the properties files *adams/some/where/Blah.props*, then you simply create a *Blah.props* file,

the `adams/some/where` part is omitted. The order in which properties files are read, is as follows:

1. jar archive
2. home directory
3. current directory

6.3 Main menu

The menu that ADAMS presents to the user, is defined in the following properties file:

```
adams/gui/Main.props
```

With this configuration you can determine the menu layout, the shortcuts, whether additional menu items get automatically discovered and added (see section 9.4 for details on adding new menu items) and whether certain menu items should get black-listed, i.e., not shown in the menu (in case of automatic menu item discovery).

Menu layout
The main menu is generated using the *MenuBar* key. This key simply lists the names of the menus that the menu bar should offer in a comma-separated list. Here is a simple example:

```
MenuBar=Program,Visualization,Windows
```

The menu items for each of the menus listed there have a key in the properties file that starts with *Menu-* and then has the nme of the menu. The value itself is once again a comma-separated list, but this time listing the class names of the menu item. The "-" character can be used to insert a separator. For instance, the entry for the *Visualization* key could look like thisL

```
Menu-Visualization=\
  adams.gui.menu.ImageViewer,\
  adams.gui.menu.PreviewBrowser
```

The *Windows* menu is a special one which gets populated automatically.

Shortcuts
Keyboard shortcuts do not only speed up interaction with an application, they are also a very personal thing. A key for shortcut consists of the prefix *Shortcut-* and the class name of the menu item. The value for the key is then according to the format defined for the *getKeyStroke(String s)* method of the *javax.swing.KeyStroke* class. You can use `ctrl` for the *Control* key, `shift` for the *Shift* key, `alt` for the *Alt* key and *meta* for the *Apple key*. The following key defines the `Ctrl+F` shortcut for the Flow editor.

```
Shortcut-adams.gui.menu.FlowEditor=ctrl pressed F
```

Automatic menu item discovery
Adding new menu items to the main menu, e.g., from other modules that you

refernce, can be quite useful. Automatic discovery takes out the hassle of having to manually maintain the properties file by adding menu items whenever a module offers a new menu item. Turning the automation on or off is done using the following key and using either "true" or "false" as value:

```
AutomaticMenuItemDiscovery=true
```

Black-listing menu items
With automatic discover enabled, you give up control on *what* menu items are being displayed (and the *where* as well). In some cases, in can be necessary to suppress a menu item (or lift the ban for one). Suppressing or *black-listing* an item is very easy, you simply need to add a key to properties file that prefixes the menu item's class name with *Blacklisted-*. The value for this property is of course boolean, with the values "true" or "false". For instance, the menu item *adams.gui.menu.SomeViewer* can be suppressed using the following key:

```
Blacklisted-adams.gui.menu.SomeViewer=true
```

6.4 Flow editor

The flow editor already comes with a basic preference dialog (*Program* → *Preferences* → *Flow*), but you can still customize it further. See section 9.5.1 for customizing the shortcuts in the main menu and section 9.5.2 for customizing the popup menu for the actor tree (menu layout and shortcuts).

6.5 Proxy

In companies or organizations, the use of proxies for internet access is quite common. In order for you to be able to go through the proxy, you need to configure ADAMS' proxy settings accordingly. You can find the settings in the *Preferences* dialog (*Program* → *Preferences* → *Proxy*). Basically, you have to specify the type of proxy (http or socks), the proxy server and the port its listening on and also exclude hosts on your network from being accessed through the proxy. These are usually: *localhost*, *127.0.0.1* and everything inside your domain. For instance, if your local domain is *blah.com*, then you can use the following wildcard: **.blah.com*. Some proxies require authentication, which you can provide as well in the dialog, once you have checked the *Requires authentication* checkbox. See Figure 6.1 for an example setup.

6.6 Time zone

ADAMS allows you to change the time zone it is operated in to one that is different from the system's one. You can find the settings in the *Preferences* dialog (*Program* → *Preferences* → *Time zone*). If you choose *Default*, then this will simply use your system's default time zone. See Figure 6.2 for an example setup.

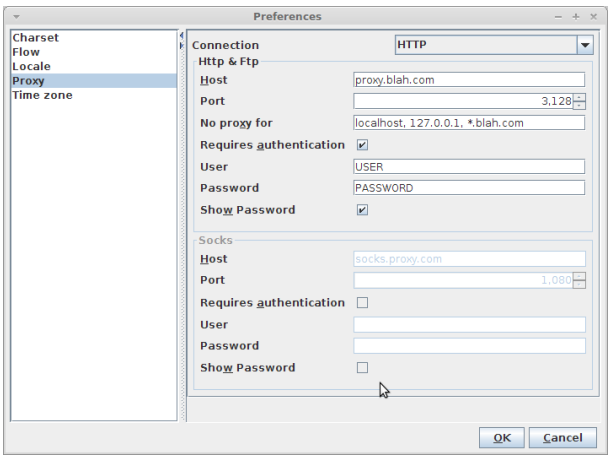


Figure 6.1: Proxy preferences

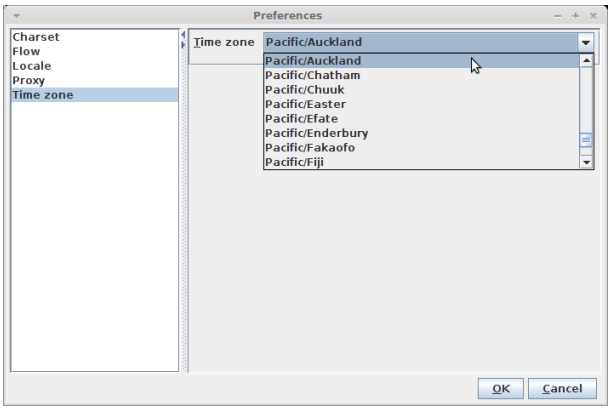


Figure 6.2: Time zone preferences

6.7 Locale

Just like with the time zone settings, you can also change the locale settings that ADAMS is operating with. By default, it uses the system's locale. You can find the settings in the *Preferences* dialog (*Program* → *Preferences* → *Locale*). If you choose *Default*, then this will simply use your system's default locale. See Figure 6.3 for an example setup.

6.8 Database access

In order to add support in ADAMS for a database, in addition to MySQL[1] and sqlite[2], the following steps are required:

- Place the jar archive of the JDBC driver in the *lib* directory.

[1] `http://www.mysql.com/`
[2] `http://www.sqlite.org/`

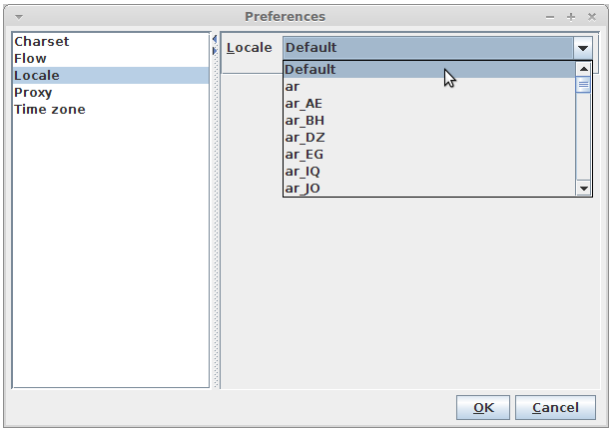


Figure 6.3: Locale preferences

- Update the *Drivers.props* properties file, adding the classname of the JDBC driver to the *Drivers* key. For instance, use *oracle.jdbc.OracleDriver* for the Oracle driver:

```
Drivers=\
  com.mysql.jdbc.Driver,\
  org.sqlite.JDBC,\
  oracle.jdbc.OracleDriver
```

  This is a comma-separated list, so just append your JDBC driver(s).

6.9 Browser

Since release 6, Java can launch the desktop's default browser. With the huge variety of desktops for Linux, this does not work properly all the time, unfortunately. Or it simply launches the wrong browser. If your desktop on Linux is not supported, ADAMS uses a fallback method to determine an available browser. The *LinuxBrowsers* property in the *adams/gui/core/Browser.props* properties file defines the order of the binaries that ADAMS looks for. The first binary that it can find, it will use.

However, if you want to use a specific browser, you can do that as well. You simply have to supply an absolute path to the browser's binary in the *DefaultBrowser* property. This will override any automatic browser discovery. Here is an example for specifying the *Firefox* browser on Linux:

```
DefaultBrowser=/usr/bin/firefox
```

This override can be used on all platforms.

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 requirements are as follows:

- Java 1.8+
- Maven 3.0+
- TexLive 2010+ (for compiling the LaTeX documentation)

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.cms.waikato.ac.nz/svn/adams/base/trunk/`

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.cms.waikato.ac.nz/svn/adams/base/trunk adams
```

Further modules are available from these repositories:

- **addons** (less common used modules)
  `https://svn.cms.waikato.ac.nz/svn/adams/addons/trunk/`
- **incubator** (experimental modules)
  `https://svn.cms.waikato.ac.nz/svn/adams/incubator/trunk/`

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: `https://adams.cms.waikato.ac.nz/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 Unix/Linux/Mac systems, is as follows:

```
$HOME/.m2
```

On Windows, use the following instead:

```
%USERPROFILE\.m2%
```

Configure maven
Download the following configuration file and place it in your maven home directory that you just created:

`https://adams.cms.waikato.ac.nz/resources/settings.xml`

This file will point maven to ADAMS' repository manager, which manages not only all the ADAMS modules, but also all its dependencies. It specifies a range of public repositories, like Maven Central.

Proxy
If you are behind a proxy, you need to tell Maven about it. Let's assume that your proxy is called *proxy.blah.com* and its port is *3128*.

If you don't need a password to connect to it, you have to add the following tag to your *settings.xml* file:

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.blah.com</host>
    <port>3128</port>
    <nonProxyHosts>localhost|*.blah.com</nonProxyHosts>
  </proxy>
</proxies>
```

If your proxy requires a user/password, then you have to 1) generate a master password with Maven (which gets stored in *settings-security.xml* in your Maven home directory) and then 2) the actual password for the proxy. The details are explained on the Maven homepage[1]. Once you've created the passwords, you have to add the following tag to your *settings.xml* file and replace the *USER* and *ENCRYPTED_PASSWORD* placeholders accordingly:

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.blah.com</host>
    <port>3128</port>
    <username>USER</username>
    <password>{ENCRYPTED_PASSWORD}</password>
    <nonProxyHosts>localhost|*.blah.com</nonProxyHosts>
  </proxy>
</proxies>
```

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`

[1] `http://maven.apache.org/guides/mini/guide-encryption.html`

- 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

Make sure that libraries that you use are publicly available from Maven Central, `http://search.maven.org/`, otherwise they won't be considered.

7.2.5 Troubleshooting

- `SSL handshake alert: unrecognized_name` – this error occurs starting with Java 7 (or 1.7.0). You can avoid this by setting the `JAVA_TOOL_OPTIONS` environment variable with the following value:

  ```
  -Djsse.enableSNIExtension=false
  ```

  On Linux or Mac OSX, you would place the following command in your `$HOME/.profile` file (effective after logging out and back in again):

  ```
  export JAVA_TOOL_OPTIONS="-Djsse.enableSNIExtension=false"
  ```

  You can see whether this is set correctly, when Maven reports the following just after issuing a `mvn` command on command-line:

  ```
  Picked up JAVA_TOOL_OPTIONS: -Djsse.enableSNIExtension=false
  ```

7.3 IntelliJ IDEA

IntelliJ IDEA[7] is a light-weight, but very powerful integrated development environment (IDE), developed by JetBrains[2].

More details on installing and customizing IntelliJ IDEA can be found here: `https://adams.cms.waikato.ac.nz/dev_start.html`

7.4 Eclipse

Prior to using IntelliJ IDEA, the choice of IDE was Eclipse[8]. It offers great support for Maven and LaTeX - provided you install the proper plug-ins.

7.4.1 Plug-ins

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

[2] `https://www.jetbrains.com/`

- m2e – adds proper maven support
  `http://eclipse.org/m2e/`
- 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/`

Furthermore, install the *buildhelper* m2e connector:

```
Window
-> Preferences
-> Maven
-> Discovery
```

For viewing the source code correctly, use the following code formatting setup:
`https://adams.cms.waikato.ac.nz/resources/eclipse-code-formatting.xml`

7.4.2 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*).

7.5 Custom Maven project

The ADAMS website allows you to create a custom Maven setup of a custom-tailored ADAMS distribution with just the modules that you want to include. You can access this functionality here:
`https://adams.cms.waikato.ac.nz/roll-your-own`
Figure 7.2 shows a screenshot of the website.

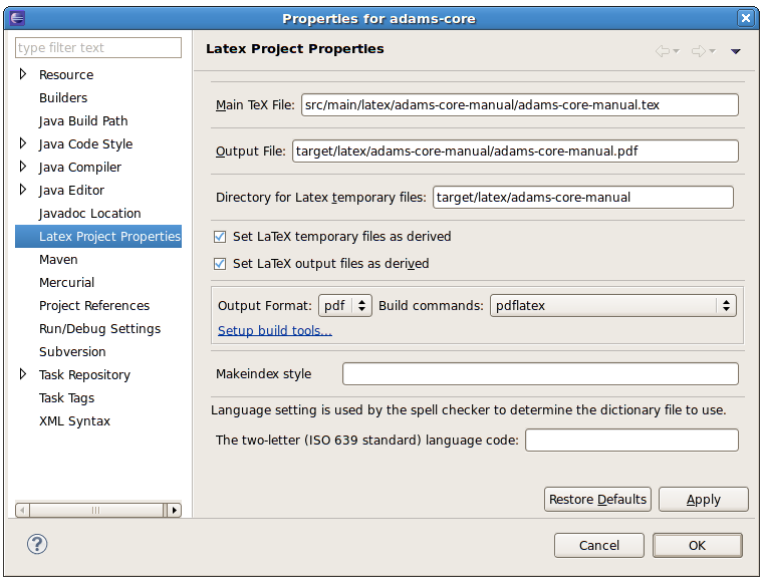


Figure 7.1: texlipse configuration for the *adams-core* module.

7.6 Non-maven approach

Though it is recommended, the use of Maven is not required. If you download a release or snapshot of ADAMS, you can simply link your project against all the jars in the `lib` directory for compiling your code.

Creating a new release can be done like this as well: simply drop any additional jars that your project requires and/or generates in the `lib` directory. You only need to create a new archive from the ADAMS directory to get an extended ADAMS release/snapshot. No need to update any scripts, as all jars in the `lib` directory get added to the classpath automatically.

When using Eclipse with this approach, the only disadvantage is that you will have to manually attach the sources to the project (located in the `src` directory). Otherwise you won't be able to view ADAMS classes as source code. This is something that gets handled automatically by the *m2e* Eclipse plugin for Maven.

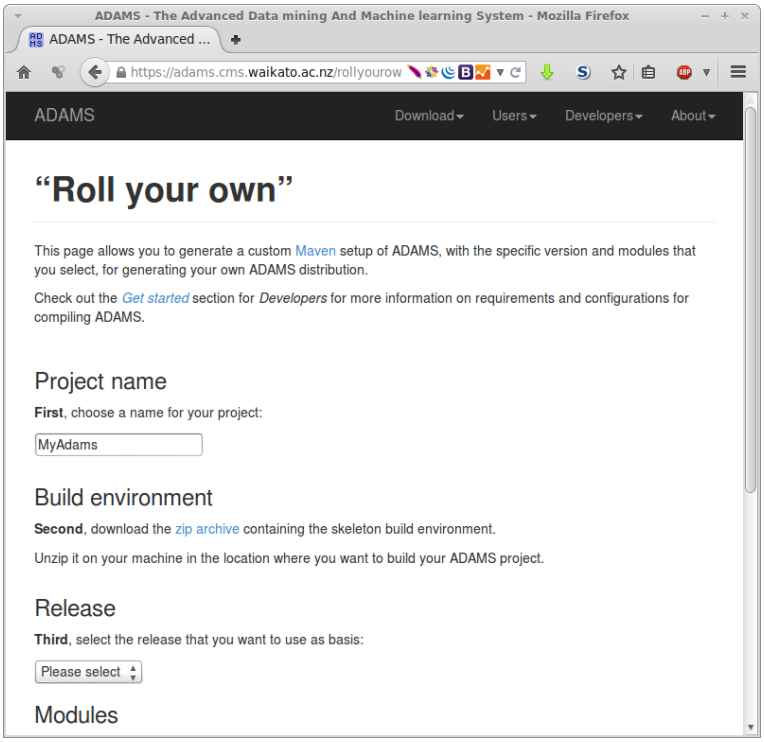


Figure 7.2: Screenshot of the "Roll your own" section of the ADAMS website.

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

The API of the flow component of ADAMS is simple by design. The idea was to provide not just a graphical interface for setting up and manipulating flows, but also enabling other people for embedding flows in their own code. Limiting the interface to only a few methods was therefore necessary. The following sections provide an in-depth discussion of the API.

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 (i.e., 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*.

8.1.2 Setting up a flow

ADAMS distinguishes between primitive actors, like the *Display* actor, and ones that handle other actors, like the *Branch* actor. The *actor handlers* can be divided into ones that have a fixed number of sub-actors, like the *IfThenElse* actors (always two sub-actors), and others that can have a more or less arbitrary number of sub-actors (a lower bound may be defined, though), like the *Branch* actor.

Usually, the *Flow* control actor is the outermost actor. But this is not necessary. In theory, any actor can be setup, executed and destroyed again. Only if you need things like variables and internal storage, you will need a control actor like the *Flow* actor to provide this kind of functionality.

Setting up a flow conists basically of nesting the actors like in the flow editor. The tree structure in the flow editor is a 1-to-1 representation of the underlying actor nesting. For actors that handle sub-actors (implementing the *ActorHandler* interface), you can use the `set(int, AbstractActor)` method for setting/replacing a sub-actor at the specified index. Actors that implement the *MutableActorHandler* interface instead, adding of new actors is much simpler: either using the `add(AbstractActor)` method (appends the actor at the end) or `add(int, AbstractActor)`, which adds/inserts the actor at the specified location. To remove any previous existing actors, you can call the `removeAll()` method. Instead of adding the actors one-by-one, some actors (mainly *MutableActorHandler* ones) offer methods for setting/getting an array of sub-actors, like the `setActors` and `getActors` methods of the *Flow* actor.

The following piece of code sets up a little flow that generates a number of random numbers between 100 and 200, which get used as input in a mathematical expression (simply dividing the numbers by PI), before dumped them into a text file in the temp directory.

```
import adams.flow.control.Flow;
import adams.flow.source.RandomNumberGenerator;
import adams.data.random.JavaRandomInt;
import adams.flow.transformer.MathExpression;
import adams.parser.MathematicalExpressionText;
import adams.flow.sink.DumpFile;
import adams.core.io.PlaceholderFile;
...
Flow flow = new Flow();

RandomNumberGenerator rng = new RandomNumberGenerator();
rng.setMaxNum(10);
JavaRandomInt jri = new JavaRandomInt();
jri.setMinValue(100);
jri.setMaxValue(200);
rng.setGenerator(jri);
flow.add(rng);

MathExpression math = new MathExpression();
MathematicalExpressionText expr = new MathematicalExpressionText();
expr.setValue("X / PI");
math.setExpression(expr);
```

```
flow.add(math);

DumpFile df = new DumpFile();
df.setOutputFile(new PlaceholderFile("${TMP}/random.txt"));
flow.add(df);
```

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, below `src/main/resources`) 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 modules 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 (below `src/main/resources`).

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 `ClassLister.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 `ClassLister.blacklist` properties file. The format for this file is similar to the `ClassLister.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 `og.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[1]:

```
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.

[1]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

Actors keep track of variables that have been attached to either one of their own options (e.g., *fieldIndex* option of the *StringCut* transformer) or to options of one their sub-objects (e.g., the *numDecimals* option of the *DoubleToString* conversion used by the *Convert* transformer). Options like sub-actors, as used by actor handlers such as *Tee* or *Branch*, are excluded from this monitoring.

Attaching a variable to an option has some side-effects that you need to be aware of when variable values change:

- affected actors get re-initialized, since the configuration has changed, resuling in calls of the *reset()* and *setUp()* methods.
- actor handlers recursively call the *setUp()* of their sub-actors.

In order to prevent losing the internal state, due calling the *reset()* method, you can backup the current state of member variables in an actor. For instance, the *Count* control actor keeps track how many tokens have passed through. This counter gets zeroed when calling *reset()*. You can backup/restore the current state using the *backupState* and *restoreState* methods. These methods use an internal hashtable to backup key-value pairs. The following code is used by *Count* to backup the counter *m_Current*:

```
public final static String BACKUP_CURRENT = "current";

protected Hashtable<String,Object> backupState() {
  Hashtable<String,Object> result = super.backupState();
  result.put(BACKUP_CURRENT, m_Current);
  return result;
}

protected void restoreState(Hashtable<String,Object> state) {
  if (state.containsKey(BACKUP_CURRENT)) {
    m_Current = (Integer) state.get(BACKUP_CURRENT);
    state.remove(BACKUP_CURRENT);
  }
  super.restoreState(state);
}
```

Of course, this counter now never gets zeroed, since we back it up all the time. In order to zero the internal counter, i.e., if an option of the *Count* actor

itself was modified and it should get zeroed, you have to *prune* the backup. You can do this by using the *pruneBackup* method, which gets called in case one of its own members got modified. The code to achieve this is as follows:

```
protected void pruneBackup() {
  super.pruneBackup();
  pruneBackup(BACKUP_CURRENT);
}
```

9.2.4 Graphical output

Using the `AbstractGraphicalDisplay` as superclass instead of `AbstractDisplay`, allows you to take advantage of some additional functionality: menu, *SendTo* framework integration.

Methods that require implementation are as follows:

- `globalInfo()` – a short description of the sink, available as help in the GUI
- `accepts()` – the classes that this sink can process and display
- `newPanel()` – generates the panel that is added to the dialog
- `clearPanel()` – removes the currently displayed data
- `display(Token)` – processes and displays the content of the token provided (of one of the accepted classes)

If it is necessary to extend the default menu, you can override the `createMenuBar()` method, which generates the `JMenuBar` that is used in the dialog.

It is recommended to implement the `DisplayPanelProvider` interface as well. By doing this, the sink can be selected in the `DisplayPanelManager` sink, which keeps a *graphical* history of the tokens passing through, by creating a single panel per token.

9.2.5 Textual output

Instead of directly sub-classing `AbstractDisplay`, you should use `AbstractTextualDisplay` instead. This abstract class already implements various interfaces like `MenuBarProvider` and `SendToActionSupporter`, to provide the user with a menu for saving the output, changing font size, etc and also enabling the user to take advantage of the *SendTo* framework. In the simplest case, this is printing the textual output on a printer.

You only need to implement the following methods in order to get a fully functional interface:

- `globalInfo()` – a short description of the sink, available as help in the GUI
- `accepts()` – the classes that this sink can process and display
- `newPanel()` – generates the panel that is added to the dialog
- `clearPanel()` – clears the (textual) content of the panel
- `display(Token)` – processes and displays the content of the token provided (of one of the accepted classes)
- `supplyText()` – returns text that is currently on display

If it is necessary to extend the default menu, you can override the `createMenuBar()` method, which generates the `JMenuBar` that is used in the dialog.

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. NB: Since ADAMS uses Maven as build system, non-Java files need to placed below the *src/main/resources* directory.

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 [9] 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*) 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

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=nz.ac.waikato.cms.adams \
     -DarchetypeArtifactId=adams-archetype-module \
```

```
-DarchetypeVersion=0.4.10 \
-DgroupId=nz.ac.waikato.cms.adams \
-DartifactId=adams-funky \
-Dversion=0.0.1-SNAPSHOT
```

This command will base the module on the latest ADAMS release, using the `0.4.10` 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/resources1/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.

9.4 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 (`ClassLister.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 with 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.

9.5 Flow editor

The flow editor itself allows for some customization:

- Adding menu items to the main menu.
- Changing the layout of the tree popup menu.

9.5.1 Main menu

The flow editor can add new menu items dynamically to its main menu. You can only need to derive a new class from the following abstract superclass and place it in the `adams.gui.flow.menu` package (or update the `ClassLister.props` file accordingly if in another package):

```
adams.gui.flow.menu.AbstractFlowEditorMenuItem
```

When you implement your new class, you need to do three things:

1. Determine in which menu the item should get added (you can start a new menu as well).
2. Create the `AbstractBaseAction` that does the actual work and also defines how your menu item looks.
3. React to updates in the user interface.

Here is an example class, called `RunningHelloWorld`, which is available if there is at least one flow currently running. The menu item titled "Say hi" simply pops up a dialog with the words "Hello World!".

```
package adams.gui.flow.menu;

import adams.gui.action.AbstractBaseAction;
import adams.gui.core.GUIHelper;
import adams.gui.flow.FlowEditorPanel;

public class RunningHelloWorld extends AbstractFlowEditorMenuItem {
  // we want to add our menu item to the "View" menu
  public String getMenu() {
    return FlowEditorPanel.MENU_VIEW;
  }
  // the action that handles the dialog
  protected AbstractBaseAction newAction() {
    return new AbstractBaseAction("Say hi") {
      GUIHelper.showInformationMessage(getOwner(), "Hello World!");
    }
  }
  // action is only available if at least one flow is running
  public void updateAction() {
    m_Action.setEnabled(getOwner().isAnyRunning());
  }
}
```

Shortcuts definitions can be stored in the `FlowEditorShortcuts.props` files (package `adams.gui.flow`, below `src/main/resources`). The definition can be accessed and converted into a `KeyStroke` object using the the following call:

```
adams.gui.core.GUIHelper.getKeyStroke(
    adams.gui.flow.FlowEditorPanel.getEditorShortcut("File.New"));
```

This example accesses the shortcut definition stored in property `Shortcuts.File.New`.

9.5.2 Popup menu

Each item in the popup menu that is displayed in the tree when opening the right-click menu for one or more actors is derived from the following class:

```
adams.gui.flow.tree.menu.AbstractTreePopupMenuItem
```

Even sub-menus are derived from this superclass, but instead of returning a simple `JMenuItem` object in the `getMenuItem(StateContainer)` method, a `JMenu` object is returned, which encapsulates other menu items.

The layout of this popup menu is defined in the `FlowEditor.props` file, in the `adams.gui.flow` package (below the `src/main/resources` directory). The key for the menu is called `Tree.PopupMenu` and the value for this property is a simple comma-separated list of class names (use "-" if you want to add a separator).

Shortcuts definitions can be stored in the `FlowEditorShortcuts.props` files (package `adams.gui.flow`, below `src/main/resources`). The definition can be accessed and converted into a `KeyStroke` object using the the following call:

```
adams.gui.core.GUIHelper.getKeyStroke(
    adams.gui.flow.FlowEditorPanel.getTreeShortcut("Help"));
```

This example accesses the shortcut definition stored in property `Tree.Shortcuts.Help`.

The following example menu item pops up a "Hello World!" dialog if the flow/actor is editable:

```
package adams.gui.flow.tree.menu;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JMenuItem;
import adams.gui.core.GUIHelper;
import adams.gui.flow.tree.StateContainer;

public class HelloWorldItem extends AbstractTreePopupMenuItem {
  protected JMenuItem getMenuItem(final StateContainer state) {
    JMenuItem result = new JMenuItem("Hello world");
    result.setEnabled(getShortcut().stateApplies(state));
    result.setAccelerator(getShortcut().getKeyStroke());
    result.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        getShortcut().execute(state);
      }
    });
    return result;
  }
  protected AbstractTreeShortcut newShortcut() {
    return new AbstractTreeShortcut() {
      protected String getTreeShortCutKey() {
     return "HelloWorld";   // won't do anything unless props file is updated
      }
      public boolean stateApplies(StateContainer state) {
        return state.editable;
      }
      protected void doExecute(StateContainer state) {
        GUIHelper.showInformationMessage(state.tree, "Hello World!");
      }
    };
  }
}
```

9.6 Image viewer

The Image viewer allows you to add custom plugins to the menu. The superclass for all plugins is:

```
adams.gui.visualization.image.plugins.AbstractImageViewerPlugin
```

- *canExecute(ImagePanel)* – determines whether the plugin can be applied to the current image panel.
- *doExecute()* – executes the plugin and returns a string depending on success (*null*) or failure (*error message*).
- *createLogEntry()* – can be used to output a string that should appear the in viewer's log tab; return *null* if that does not apply.

9.7 Database access

In order to add support in ADAMS for a database, in addition to MySQL[2] and sqlite[3], the following steps are required:

[2] `http://www.mysql.com/`
[3] `http://www.sqlite.org/`

- Add a dependency for the JDBC driver to your *pom.xml* project definition (search on Maven Central[4]). For instance, adding the Oracle JDBC driver:

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
```

- Add or update the *Drivers.props* properties file, adding the classname of the JDBC driver to the *Drivers* key. For instance, use *oracle.jdbc.OracleDriver* for the Oracle driver:

```
Driver=oracle.jdbc.OracleDriver
```

  This is a comma-separated list. If there are already entries, just append your JDBC driver(s).

[4] http://search.maven.org/

Chapter 10

JUnit tests

Any additional JUnit test should be derived from the following superclass:

```
adams.test.AdamsTestCase
```

Regression tests
It is possible to suppress regression tests:

- all: `-Dadams.test.noregression=true`
- quick info: `-Dadams.test.quickinfo.noregression=true`
- data processors: `-Dadams.test.data.noregression=true`
- actors: `-Dadams.test.flow.noregression=true`

Chapter 11

Temporary files

By default, the `$TMP` placeholder uses the system's temporary directory. This is determine by Java's *java.io.tmpdir* system property. However, by setting the *adams.io.tmpdir* system property, one can override this application-wide. You can use `-Dadams.io.tmpdir=/some/where/tmp` in the command-line as parameter to the JVM.

Chapter 12

Parser plugins

The parsers for expressions, like mathematical expressions and boolean expressions are quite powerful as they are. However, in the past, adding new functions required changing the lexer and parser generator grammars, re-generating Java code and recompiling. It is now possible to add functions (return a value) and procedures (don't return anything - not used at the moment, reserved for future use) by simply deriving classes from an abstract superclass.

A new function only needs to implement the *adams.parser.plugin.ParserFunction* interface (analog for procedures: *adams.parser.plugin.ParserProcedure*). The function name is defined by the *getFunctionName()* method, with the name only consisting of letters, numbers and underscores. You can supply up to 10 parameters to your function. In order to avoid clashes in the parser, this function name then gets prefixed with *f_* (analog for procedures: *p_*).

Below is an example function for return the value of an environment variable, with the name env and available in the parser via *f_env(name)*:

```
public class Env extends AbstractParserFunction {
  public String getFunctionName() {
    return "env";
  }
  public String getFunctionSignature() {
    return getFunctionName() + "(String): String";
  }
  public String getFunctionHelp() {
    return getFunctionSignature() + "\n"
      + "\tFirst argument is the name of the environment variable to retrieve.\n"
      + "\tThe result is the value of the environment variable.";
  }
  protected String check(Object[] params) {
    if (params.length != 1)
      return "Only accepts single parameter, which must be name of the "
        + "environment variable to retrieve!";
    return null;
  }
  protected Object doCallFunction(Object[] params) {
    return System.getenv().get(params[0]);
  }
}
```

12.1 Programmatic hooks

A flow is usually a self-contained unit, which makes it hard to hook into it from a programmatic point of view. However, using the `ProgrammaticSink` pseudo-sink you can easily add listeners that listen for tokens arriving at this actor. Here is a little code snippet that shows how to use this sink. The flow simply generates integer tokens in the `ForLoop` actor and the `ProgrammaticSink` simply outputs the tokens to stdout.

```
public static void main(String[] args) throws Exception {
  Environment.setEnvironmentClass(Environment.class);
  // assemble flow
  Flow flow = new Flow();
  ForLoop forloop = new ForLoop();
  flow.add(forloop);
  ProgrammaticSink psink = new ProgrammaticSink();
  psink.addTokenListener(new TokenListener() {
    public void processToken(TokenEvent e) {
      System.out.println(e.getToken().getPayload());
    }
  });
  flow.add(psink);
  // setup flow
  String result = flow.setUp();
  if (result != null) {
    System.err.println("Failed to set up flow: " + result);
    return;
  }
  // execute flow
  result = flow.execute();
  if (result != null) {
    System.err.println("Failed to execute flow: " + result);
    flow.wrapUp();
    flow.cleanUp();
    return;
  }
  // finish up
  flow.wrapUp();
  flow.cleanUp();
}
```

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* – Reutemann, Peter and Vanschoren, Joaquin (2012). *Scientific Workflow Management with ADAMS*. Proceedings of the Machine Learning and Knowledge Discovery in Databases (ECML-PKDD), Part II, LNCS 7524, pp. 833837. `https://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] *IntelliJ IDEA* – Even the community edition of IntelliJ IDEA is a powerful environment for development Java projects. `https://www.jetbrains.com/idea/`

[8] *Eclipse* – An open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. `http://eclipse.org/`

[9] *JUnit* – JUnit is a unit testing framework for the Java programming language. `http://junit.org/`

[10] *DateFormat* – For parsing date/time strings and turning date/time objects into strings. `http://download.oracle.com/javase/1.5.0/docs/api/java/text/DateFormat.html`

[11] *Regular expressions* – The regular expression handling as available in Java. `http://download.oracle.com/javase/1.6.0/docs/api/java/util/regex/Pattern.html`