Eberhard Karls Universität Tübingen

Fakultät für Informations- und Kognitionswissenschaften
Wilhelm-Schickard-Institut

Studienarbeit Bioinformatik

A Processing Framework for Mayday

Verfasser:

Florian Battke

July 21, 2006

betreut von

Dr. Kay Nieselt & Janko Dietzsch

Universität Tübingen
Fakultät für Informations- und Kognitionswissenschaften
Sand 1, 72076 Tübingen

Abstract

Almost all microarray data analyses include repetitive tasks such as data imputation, normalization and filtering for interesting features. Automation of these recurring processing steps gives researchers more time to focus on the actual analysis.

In this work I present a new plugin for Mayday, the Mayday Processing Framework (MPF). The MPF offers an extensible framework for integrating data processing functions ("modules") into Mayday. Modules can easily be combined into processing pipelines using the user-friendly GUI offered by the MPF Designer. A graph visualization algorithm was developed to make pipeline creation as intuitive as possible. Created pipelines can be stored for later use.

With the MPF Applicator, modules can be used to process probe lists and several probe lists can be processed in batch mode without any user intervention.

The output of processing are new probe lists. These may contain unmodified probes from the input probe lists, changed input probes and new probes. New and changed probes are annotated using Mayday's meta information objects (MIOs) to describe which processing module created or modified the probe and how the module's options were set.

The new framework greatly simplifies the implementation of new modules by providing mechanisms for manipulation of Mayday's data structures while shielding programmers from their complexity and by offering a range of objects to specify module options. These option objects automatically create GUI elements and validate user input.

Key features of the MPF are automatic memory-efficient management of new and changed probes and probe lists, creation of descriptive MIOs as well as consistent recovery when processing jobs are cancelled or fail to complete.

Contents

1. Introduction

1.1 Mayday

“Mayday is a workbench for visualization, analysis and storage of microarray data. It features a graphical user interface and supports the development and integration of existing and new analysis methods. Besides the infrastructural core functionality, Mayday offers a variety of plug-ins, such as various interactive viewers, a connection to the R statistical environment, a connection to SQL-based databases, and different data mining methods [...]In addition, so-called meta information objects are provided for annotation of the microarray data allowing integration of data from different sources.”

[DGN06]

This section will give a very short introduction into Mayday’s most important data structures (also see figure 1.1). For an in-depth description of Mayday’s implementation, the reader is reffered to section 4.3 of [Geh04].

Microarray data is most commonly represented as an *expression matrix*, where columns represent *experiments* and rows represent genes or ESTs which are subsumed under the term *probes*. Thus, a cell c_{ij} in the matrix contains the value for probe i in experiment j. This value can be a ratio (the result of comparing the measured expression value to a reference experiment), the logarithm of such a ratio, or simply the result of some measurement directly derived from scanning a microarray.

Mayday treats an expression matrix as a list of probes, where each `Probe` represents one row in the matrix. Essentially, a `Probe` is a vector of expression values of type `Double`. Every `Probe` must

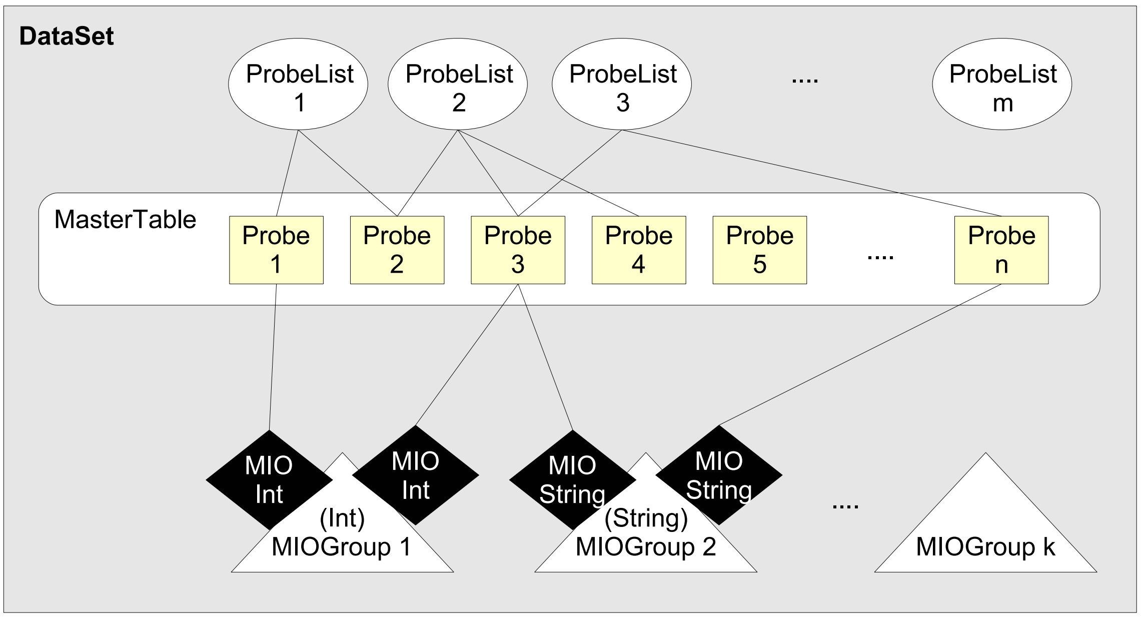


Figure 1.1: Overview of Mayday’s core data structures and their connections. The `DataSet` *contains the* `MasterTable`*, all* `ProbeList`*s and all* `MIOGroup`*s. The* `MasterTable` *contains all* `Probe`*s. In this example,* `ProbeList` *1 contains* `Probe`*s 1 and 2,* `Probe` *3 is contained in* `ProbeList`*s 2 and 3 and is annotated by a* `MIO` *object of type* `String`*. This* `MIO` *is contained in* `MIOGroup` *2. All connections presented here are bidirectional, that is, both objects know of their connection partners.*

have an identifier which is part of that `Probe`'s annotation. The annotation can further contain a short description and a more comprehensive description which might be plain or HTML-formatted text. All `Probe`s are stored in the so-called *master table*, an unordered set of probes. Within a `MasterTable`, each `Probe`'s identifier must be unique.

In addition to being contained in the `MasterTable`, a `Probe` can also be part of an arbitrary number of *probe lists*. A `ProbeList` represents a subset of the `MasterTable`. A `Probe` may only be included once in any given `ProbeList` (i.e. `ProbeList`s are sets in the mathematical sense), but several `ProbeList`s may reference the same `Probe` object. The `Probe` keeps a list of `ProbeList`s it is contained in. Using several `ProbeList`s, the `MasterTable` is divided into (not necessarily disjunct) subsets. A special probe list, called *global* probe list, contains all probes of the `MasterTable`.

An important concept in Mayday are *meta information objects* (MIOs). MIOs can be assigned to `Probe`s as well as `ProbeList`s to provide additional information such as gene annotations or computations performed to obtain a certain `ProbeList` (e.g. clustering). Mayday supports MIOs for different data types (String, Integer, Double, etc.). To structure meta information, Mayday uses so-called *MIO groups*. Using `MIOGroup`s, users get access to semantically related sets of MIOs. This allows selecting one MIO group for a specific function instead of selecting a MIO for each probe that will be transformed by that function. Every MIO belongs to exactly one `MIOGroup` and every `MIOGroup` contains MIOs of exactly one data type.

A `MasterTable`, its `ProbeList`s and the associated `MIOGroup`s together make up a *data set*. The `DataSet` is the highest object in Mayday's hierarchy of data structures.

1.2 Motivation

Evaluating data from microarray experiments is a complex task. But there are some steps prior to the actual data analysis that are almost identical in most cases. These make up the "traditional" pipeline of data processing and include the substitution of missing values (imputation), the normalization of the probe values, logarithmic transformation of the data and, quite often, a filtering step to remove genes deemed "uninteresting".

Until recently, Mayday did not possess a mechanism to automate these repetitive tasks, i.e. for every dataset imported into the program, a user would have to run each of the pipeline's steps manually. This situation has several drawbacks: While these preprocessing steps are necessary before data analysis can take place, they consume research time better spent on the actual analysis. Furthermore, when these steps are performed manually, the user has to keep track of the parameters used in order to ensure that all datasets are processed in the same way.

In this work, I present the *Mayday Processing Framework* (MPF), a Mayday plugin that allows users to automate repetitive tasks. The next section will outline the requirements formulated for the MPF. Later chapters will provide some insight into the inner workings of the plugin and into its application.

2. Requirements

For a processing framework like the MPF, several essential requirements can be formulated:

- The MPF should be a framework allowing the easy addition of processing modules. It should be as simple as possible to write new modules. Programmers should only have to worry about how their module processes data without having to understand subtle details of Mayday's data structures.
- The framework should be as generic as possible. It should not restrict the way modules can process the data.
- One very important aspect is the possibility to create and run *batch jobs*, i.e. the MPF should provide a means to apply a processing function to an unlimited number of datasets in an unsupervised way. Ideally, the batch job should be able to run while the user leaves the computer alone and the MPF should try to do as much work as possible, i.e. in the event of an error in one job, all other jobs should still be finished when the user returns to the computer.
- Another fundamental part is the creation of processing pipelines from processing functions. The MPF should allow modules to be combined to pipelines in whatever way the user prefers. The pipelines should contain option values for the included sub-modules and provide a way to present some of these internal options as external options to users of the pipeline. Users should be able to save and load pipelines along with all contained options. From the users' point of view, complex processing pipelines should act exactly like simple processing modules, making it possible to use one processing pipeline as a sub-module within another.
- Using the MPF should be as simple as possible so that users without a computer science background would have no problems applying processing modules as well as creating new processing pipelines from these modules.

In the following chapters I show how these requirements are met.

3. Structure of the MPF

The Mayday Processing Framework (MPF) can be divided into four major parts, whose general characteristics are described here. Details for each part are given in chapter 4:

1. Basic processing modules and their options
   Basic modules are created by programmers and perform one processing task, e.g. logarithmic transformation. Module options are used to change the way that task is performed, e.g. what log base is used.

2. The MPF Applicator
   The Applicator links Mayday and the processing modules. Here, the user can select the module to apply, create and run batch jobs etc.

3. Processing pipelines
   These are created by users of the MPF and combine several processing modules. No programming is needed to create pipelines.

4. The MPF Designer
   The Designer is used to create processing pipelines in a graphical ('point-and-click') environment.

3.1 Basic modules

Every processing module has five defining characteristics: These are a *name* and a *description* of the module's function that gives enough information about the module to make it clear what it can be used for. The number of *inputs* and *outputs* is needed to ensure the integration of the module into the MPF and into processing pipelines. A set of *options* is offered that users can manipulate to change the way the module works (for very simple modules, this set can be empty). And finally the *function* that performs the actual processing is specified.

3.2 Options

Option objects are used to store values that influence the way a module processes its input. As with the modules themselves, option objects should be easy to implement to allow the addition of specialized options needed by new processing modules.

Every option object contains the *name* of the option and a *description* detailing its semantics, i.e. what the option does, which values are allowed and what they mean. The option object is responsible for *creating GUI objects* that allow changing the option's value, for *checking* user input for validity and providing helpful messages when invalid input is entered and for *converting* the option's value to/from a `String` for storage purposes. All options are based on the `OptBase` class.

3.3 MPF Applicator

The Applicator links Mayday and the MPF. It is called by Mayday's `PluginManager`. Using the Applicator to process data consists of three steps:

1. Selecting the processing module or pipeline that will be applied to the input probe lists. Here, the Applicator checks whether enough input probe lists are provided for the selected module. If there are more inputs than needed, a batch job is created. In this step, users can also start the Designer to edit existing processing pipelines or create new ones.

2. Assigning input probe lists to the input slots of the selected module. For modules that expect more than one input probe lists, the user can decide which probe list to assign to each of the input slots before calling the Applicator. The situation gets even more complex for batch jobs where every input probe list has to be assigned to a certain job number and, within that job, to a specific input slot.
   This step is only necessary for modules that require more than one input. For most modules, the Applicator will simply skip this step.

3. Setting module options (or leaving them at their default values).

After the options have been set, the job(s) can be started. The Applicator will then call the processing modules with the assigned input, display progress and error messages and collect the output. If an error occurs within one job, the Applicator will clean up memory and proceed with the next job. If the user decides to cancel processing, the Applicator will notify the currently running module of that fact before exiting. The output of all jobs that have been completed until then will be returned to Mayday.

3.4 Processing pipelines

Processing pipelines are created using the MPF Designer. They contain a set of processing modules (or pipelines) together with option values and describe a directed graph structure on these modules where each module is a node in the *filter graph*. Connections between nodes indicate the path that input data is taking through the graph, i.e. the order that the modules are executed in.

Internally, all processing pipelines are managed by the `ComplexFilter` class. This class is responsible for checking

- whether all input and output slots of all modules are properly connected
- whether the graph contains cycles (these would create an infinite loop during execution) and
- whether processing pipelines contained in a pipeline recursively depend on this pipeline. From the graph's point of view this is the same as having cycles in the graph, but it is harder to detect because the embedded pipeline's graph is represented by only one node in the enclosing pipeline graph.

`ComplexFilter` takes care of loading and saving pipelines and calculating the processing order of embedded modules as well as executing the filter graph and reporting the current progress to the Applicator.

3.5 The MPF Designer

The Designer is used to create processing pipelines from modules. It provides a GUI for adding and removing modules, for setting their options and creating connections between the output slots of one module and the input slots of another.

4. Details

The MPF is a fairly complex plugin with about 6800 lines of code in 68 classes. (Some of these classes are processing modules that will not be discussed here except for the examples in section 5., some of them are internal classes that will also not be discussed).
The core classes can be divided into five groups:

1. Classes used by every processing module
2. Classes belonging to the Applicator
3. Classes used to represent processing pipelines
4. Classes belonging to the Designer
5. Static classes used everywhere in the MPF

I will start with a detailed discussion of these five groups before presenting the overall class diagram of the MPF's structure.

4.1 Basic processing modules

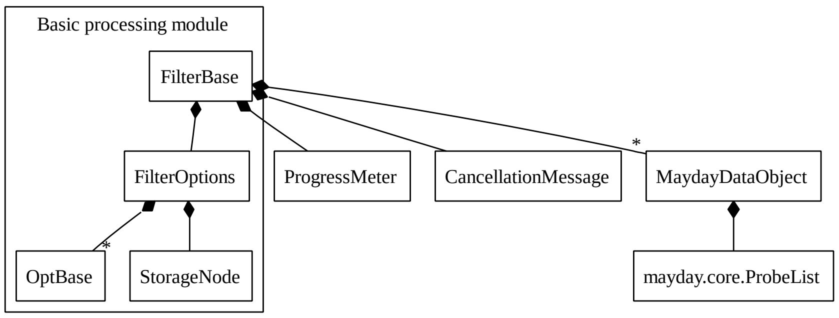


Figure 4.1: UML diagram of the classes used by basic processing modules.

Every basic processing modules is derived from the `FilterBase` class. Options are stored in an instance of `FilterOptions` containing a set of `OptBase` descendants. `FilterOptions` uses the `StorageNode` class to create a representation of its contents that can be stored to a file and loaded at a later time. During the execution of the module, the module can check its `CancellationMessage` instance to see whether the user requested to abort the current process. Modules should perform this check often enough to prevent unreasonably long waiting periods. The `ProgressMeter` class is used to inform the user about the processing status (as a percentage) and to write messages to the log. A very important role is played by the `MaydayDataObject` class which encapsulates a `mayday.core.ProbeList` instance and provides methods to easily and safely manipulate the probe list's contents.

4.1.1 `FilterBase`

`FilterBase` is the base class for all processing modules within the MPF. I decided to make this a real class instead of an abstract class or interface because I wanted to incorporate some basic functions that make the implementation of descendant classes easier. Some of the functions are needed to use modules in the MPF Applicator or to integrate them into processing pipelines. Besides declaring the abstract `execute()` method, `FilterBase` defines the basic set of information needed to include a processing module in the MPF:

- The *name* of the module is used in the list of available modules in Applicator and Designer.
- The *description* should contain all the information users need to decide whether this is the module they want to use and to understand how using it will affect their data.
- The list of *options* contains all `OptBase` descendants used to change the way the module works.
- The *number of* input and output *slots* is used by the Applicator to decide how a module is applied to multiple input probe lists and by the Designer to incorporate the module into a filter graph.
- The module *version number* plays an important role with respect to processing pipelines: Upon loading a pipeline, the `ComplexFilter` class has to ensure that all modules contained in the pipeline are available. This means checking whether the modules exists *in the right version* because if the expected version differs from the one found on disk, the pipeline might not work as planned. This can either lead to an exception during pipeline execution (which is bad enough as it is), or – and this is even worse – the pipeline might just work but the data is processed differently than what the pipeline description promised to the user, for example because the semantics of one submodule's option changed.
  As a consequence, module implementors and pipeline designers should always increment the version number when the number of pipeline input or output slots has changed, when options changed their meaning or the option list was reordered or resized.

For more information on how the `FilterBase` class can be extended to implement new processing modules, see chapter 5., consult the JavaDoc or read the source code of the modules provided with the MPF base installation.

4.1.2 `OptBase`

As the parent class for all types of option values, `OptBase` defines methods needed to load, save, display and edit the represented option value. In addition, some more fields are needed so that options can be externalized when a processing module becomes part of a processing pipeline.

Every option has a name and a description. The name should be as short as possible while still being descriptive enough for users to have a basic idea what the option does. Further information as to the option's semantics, its interaction with the values of other options for a particular module, etc. should all be put into the `Description` string. In addition to those two fields, every option also has a field detailing whether it may be used as an externalized option in the context of a processing pipeline. The implementor of a processing module must decide whether to allow externalization of a particular option (which is the default behaviour). In some cases, it can be necessary to deny externalization. More on externalization can be found in section 4.3.4.

One member that is not present in the `OptBase` class is the option's value. Implementors of descendant classes are strongly encouraged to add another member "`Value`" of the appropriate type to their class. I decided not to make this part of the base class because doing so would have forced `Value` to be of type `Object`, thereby circumventing all type safety Java provides.

4.1.3 `FilterOptions`

`FilterOptions` is used to encapsulate the set of options of a particular processing module. Most of its functions are only needed by the MPF core and do not concern implementors of new modules. These include saving and loading the option set via the `StorageNode` class (discussed later in this section), showing and updating the GUI window and dealing with user input.

It is important to know that options are referenced by their index in the processing module's `Options` object. Changing the order of options can lead to problems when the module is being used as part of a processing pipeline. Thus every modification that changes the individual indices of a module's options is a reason to increment the module's version number.

Users of this class need to know about the following functions:

`void add(OptBase)`	appends an option to the end of the option set.
`void createOptionList()`	recreates the GUI window. Call this function whenever you change the contents of the option set outside of the module's constructor so that changes are visible to the user.
`OptBase get(int)`	retrieves an option from the option set by its index.
`void remove(OptBase)`	removes an option from the option set.

When the user decides to change a processing module's options, `FilterOptions` creates a window containing GUI elements for all `OptBase` descendant objects it contains. For each option object, `OptBase.getEditArea()` is called. To prevent unnecessary creation of GUI elements, these are created lazily by a call to `createEditArea()` if necessary. Subclasses overload this function to insert appropriate GUI elements into the EditArea.

By clicking on the "Accept" button of the option window, the user triggers the validation of all options by means of each option's `validate()` function. If an option can not validate its current input, it has to inform the user about that fact and about acceptable input values. The option dialog will remain open until

- the user clicks "Cancel" and all option values are restored to their previous contents or
- the user clicks "Accept" and all options validate successfully.

Only upon successful validation are all options asked to `accept()` their new values. This ensures consistency because either all options are updated or none of them are. Figure 4.2 shows this part of user interaction.

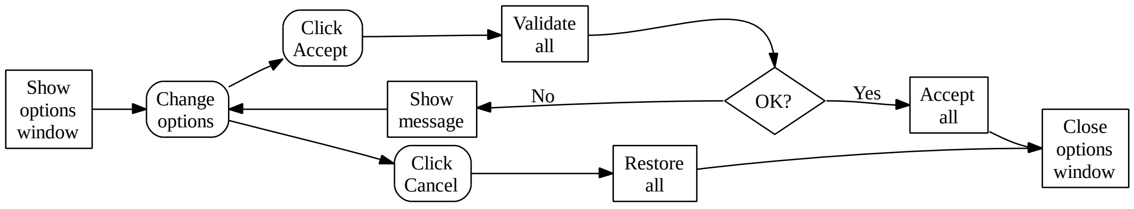


Figure 4.2: Flow diagram for user interaction with an option window.

4.1.4 `StorageNode`

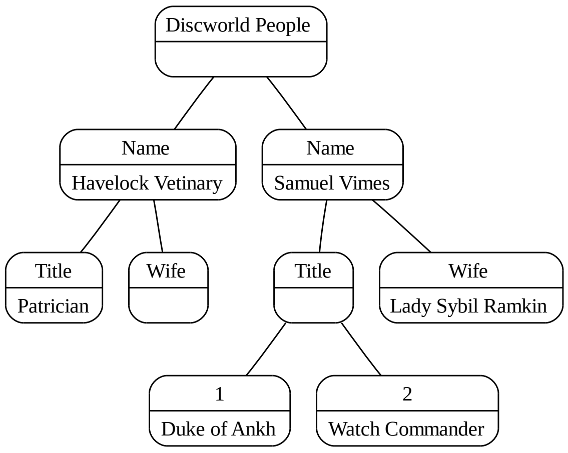
The `StorageNode` class is used to represent data in a tree structure. It provides methods for storing to and loading from a file. Every instance of this class represents a node in a graph, more precisely, every instance stores a key/value pair of `Strings` and represents the root of a directed (sub)tree of other `StorageNode` instances.

The use of only one class to represent the root, internal nodes and leaf nodes in a tree makes it possible to use any node as starting point for saving/loading data, i.e. individual subtrees within one big tree can be stored to/loaded from different files. When storing a subtree starting from any root node k, the root itself is not stored in the file.

The serialized form of a node is one line of text containing a pair of `Strings` separated by "=". To make this format robust, occurrences of "=" and line breaks must be removed from the key and value strings. This is done by replacing a line break with `~newline~`, "=" by `~equals~` and "~" by `~tilde~`. The hierarchy of nodes in the tree is represented by indentation, so that every line in a file created by storing a tree is of the following form:

<indent><key>=<value><line-break>
where <indent> consists of n times the space character (ASCII 32)

A child ℓ of a node k with indentation length n_k directly follows k in the file (that is, no node f with $n_f \leq n_k$ comes between k and ℓ) and has the indentation length $n_\ell = n_k + 1$. Note: The use of indentation to represent the hierarchical tree structures makes it necessary to remove leading whitespace from <key> strings. The following example illustrates the correspondence between a graph and its representation on disk:



File representation of this tree when saved from the root node.

```
Name=Havelock␣Vetinary\n
␣Title=Patrician\n
␣Wife=\n
Name=Samuel␣Vimes\n
␣Title=\n
␣␣1=Duke␣of␣Ankh\n
␣␣2=Watch␣Commander\n
␣Wife=Lady␣Sybil␣Ramkin\n
```

Figure 4.3: Example of `StorageNode` *file representation of a tree.*

4.1.5 `CancellationMessage`

This class only holds one `Boolean` value, initialized to `false`. During the execution of a processing module or pipeline, the MPF Applicator creates an instance of `CancellationMessage` and connects it to the module. In pipelines, this instance is in turn connected to all sub-modules of the pipeline so that all active `FilterBase` descendants point to the same object. If the user decides to cancel a running process, Applicator sets the `Boolean` to `true` and all processing modules can see this change instantly.

4.1.6 `ProgressMeter`

Progress information is very important when processing large datasets because this usually takes up all CPU cycles so that without any indication of work being done users might suspect that their system simply crashed. Within the MPF, the `ProgressMeter` class is used to transport progress information from processing modules to the GUI.
Instances of ProgressMeter come in two flavours:

- Master instances
  A master instance takes care of creating and updating a `JProgressBar` component. Setting its percentage to $x\%$ will result in the `JProgressBar` displaying a value of $x\%$.
- Child instances
  A child instance has a link to its parent (which may be another child instance or a master instance) and does not maintain its own `JProgressBar`. Child instances represent one

(sub)task in a list of tasks. In addition to a percentage value, they also store a *baseline* and a *factor* value. When the percentage is set to x%, these two values are used to calculate the new percentage $\hat{x}$ of the parent instance as

$$\hat{x} = \text{baseline} + \text{factor} \times x$$

Consider the following example: We want to read $2n$ lines from a file, sort them and save the first n lines to another file. This job obviously contains three subtasks. The following table shows the values for *baseline* and *factor* for the three `ProgressBar` child instances. Note that the complete job can be divided into five parts of equal size $\frac{1}{5} = 0.2$:

baseline= 0
factor= 0.4
	baseline= 0.4
	factor= 0.4
		baseline= 0.8
		factor= 0.2

Read lines	Sort lines	Write lines

Now let's say *Sort lines* has sorted 50% of its input. To communicate that fact it sets its associated `ProgressMeter` instance's percentage value to 0.5. This in turn sets the master instance's percentage to

$$0.4 + 0.4 \times 0.5 = 0.6$$

which means that our overall progress is 60%.

As with the `CancellationMessage`, the Applicator creates a new master instance M when processing is started. For every job i, a new per-job child instance J_i is created. (This applies to batch mode as well as single-job execution as the latter is handled like a batch job list containing only one element.) Within processing pipelines, child instances are created for every submodule.

In addition to the direct setting of percentages, this class also provides a function that converts a simple statement like "n steps of m total steps completed" into a percentage value.

4.1.7 `MaydayDataObject`

Though initially designed as a simple placeholder for `ProbeList` objects, `MaydayDataObject` (MDO) has developed into one of the most important classes of the MPF. `ProbeList`s are very delicate objects that require careful handling if one doesn't want to confuse Mayday. `ProbeLists`, `Probes` and Meta information objects (MIOs) form a complex interconnected structure and every member of that structure has its own special needs. `ProbeLists`, for example, do not permit two `Probes` to have the same name and changing the name of a `Probe` after it has been added to a `ProbeList` breaks the link between the two objects. Deleting such a `Probe` from the `ProbeList` might or might not work because `Probe` retrieval within the `ProbeList` is done via name matching. The `ProbeList` itself contains a HashMap where `Probe` names are keys and `Probes` are values. I found that visualization plugins may fail if the key is not equal to the name stored in the associated `Probe` object. Many more examples of these difficulties can be found by reading the source code comments in `MaydayDataObject.java`.

In addition to shielding the implementors of processing modules from these problems, `MaydayDataObject` is also needed for efficient memory usage and cleanup. When a module changes a probe's values, the MDO has to make sure that these changes aren't performed on the original input `Probe` but rather on a clone. During the execution of a processing pipeline, MDO keeps track of which `Probes` have already been cloned to prevent needless (time- and memory-consuming) cloning. When processing has to be aborted due to an exception or because of a user request, MDO takes care to clean up used memory, to remove newly created probes from `MIOGroups` they were added to, etc.

Finally, `MaydayDataObject` takes care of *silencing* affected `ProbeLists` and `MasterTables` for the duration of processing. Silencing means that changes to a probe container are not immediately communicated to visualizers and probe list views. Updating visualizers is very expensive in terms of computations and thus silencing leads to a huge speed improvement when visualizers are opened an a processed data set. `MaydayDataObject` uses reference counting to make sure that silenced containers are "un-silenced" when they are no longer used by the MPF.

To guarantee data consistency, users should only manipulate `ProbeLists` using functions provided by the MDO. Direct manipulation of the `ProbeList` is almost certain to corrupt Mayday's data structures. Also, `Probe`s contained in the MDO must not be manipulated directly as this prevents the MDO from keeping track of changes and cloning `Probe`s when necessary. The following functions cover the most commonly needed operations on `ProbeLists`[1]:

`void clear()`	removes all `Probes` from this MDO.
`Probe add(Probe)`	adds a `Probe` to this MDO, changing its name to make it unique. Returns the added `Probe`.
`Probe cloneProbe(Probe)`	creates a duplicate of a given `Probe`
`void dismiss()`	cleans up memory used by this MDO. This function must be called on MDOs that will not be used again.
`int getNumberOfExperiments()`	returns the experiment dimension of the data
`int size()`	returns the probe dimension of the data
`java.util.Iterator<Probe> iterator()`	returns an Iterator over all `Probes` in this MDO. This allows using the Java 1.5 for-loop to remove or insert `Probes` without invalidating the iterator.
`Probe newProbe(String)`	creates a new `Probe` with the given name, modifying the name when necessary to make it unique.
`void remove(Probe)`	removes a `Probe` from this MDO
`Probe setProbeValue(Probe, int, double)`	changes one experiment value in a `Probe`, returning the same `Probe` or a new one if the `Probe` was cloned in the process.
`Probe replaceValues(Probe, Probe)`	sets all experiment values of one `Probe` to new values, returns the changed `Probe`.

I will now describe the key operations performed by MDOs in more detail. To clarify the connections between probe lists, probes and meta information, I will use diagrams where `ProbeLists` are shown as circles, `Probe`s as rectangles, `MasterTable`s as rounded rectangles and meta information attached to an object as a diamond shape. Meta information groups will be shown as triangles. An arrow from one object A to another object B shows that A has knowledge of B (i.e. has a pointer to B). To keep the diagrams as concise as possible, I'll use only one probe as a representative for a set of probes and hide those elements that aren't relevant to the task at hand.

Ensuring data integrity: makeUnique() and makeUniqueProbe()

Before any changes can be made to a probe in the MDO's probe list, two things have to be assured: Firstly, the probe list must be made unique. Upon construction, the MDO points to one of the input probe lists. Applying a processing module should, however, never change the input data, so

[1]For a more detailed list, see the JavaDoc files.

a clone of the input probe list has to be created. This means that a new `ProbeList` (`pl`) is created and that all probes from the original `ProbeList` (`pl_orig`) are included in `pl`. Meta information is duplicated as well. This is done by the `makeUnique()` method (see figure 4.4). Because of the way Mayday's main data structures are implemented, `pl` must have a different name than `pl_orig`.

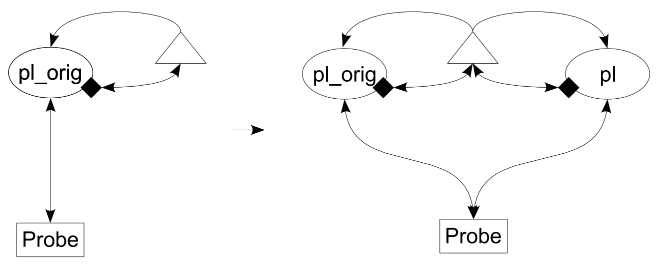


Figure 4.4: The situation before and after `makeUnique()`

Secondly, the probe itself has to be made unique ("detached" from the original input probe). This involves six steps (see figures 4.5 and 4.6):

1. Break the link between the probe and the detached probe list but keep it connected to the original probe list.
2. Clone the probe. This also clones all the probe's connections but not its meta information complement.
3. Rename the clone. The new name must be unique to guarantee that the next steps are not performed on the original probe.
4. Break the link between the cloned probe and the original probe list. Now the cloned probe is not connected to any probe list.
5. Clone the original meta information and append it to the cloned probe.
6. Add the cloned probe to the detached probe list.

The MDO keeps track of all `Probe`s it has created since its construction. This information is used to make sure that a call to `makeUniqueProbe()` is only executed when necessary. If the probe is already unique, no changes have to be made. This list of newly created probes is also used at the end of the MDO's lifetime, when all new probes have to be integrated into the `DataSet`'s `MasterTable`. If the MDO is destroyed after processing was interrupted by the user or by an exception, all new probes have to be removed from memory (see the discussion of `MaydayDataObject.dismiss()`).

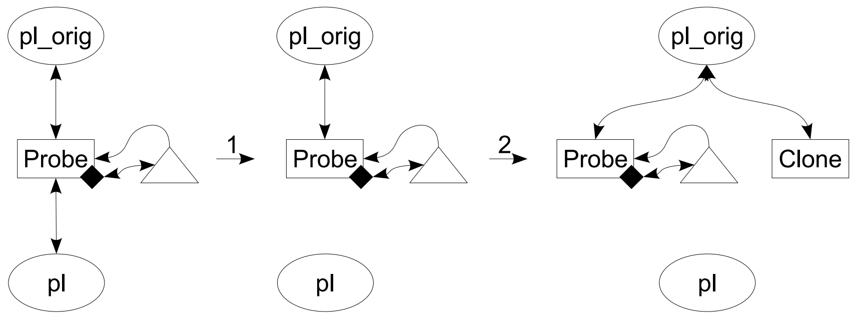


Figure 4.5: The first two steps during `makeUniqueProbe()`*:*
Removing the original `Probe` *from the detached* `ProbeList` *and creating a clone.*

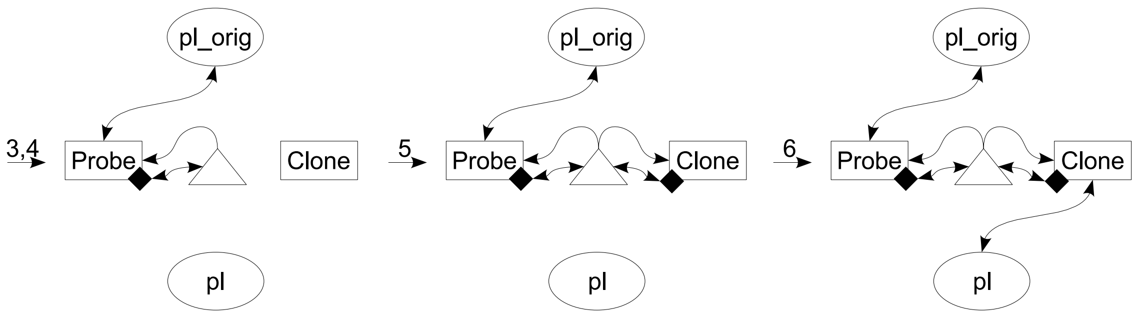


Figure 4.6: The final steps during `makeUniqueProbe()`*: Removing the clone from the original* `ProbeList`*, cloning meta information and inserting the clone into the detached* `ProbeList`.

Note: The operations for adding a probe to the MDO, creating a duplicate of a probe already contained in the MDO and creating a new probe in the MDO all work along the same lines as `makeUniqueProbe()`.

Cleaning up memory: dismiss()

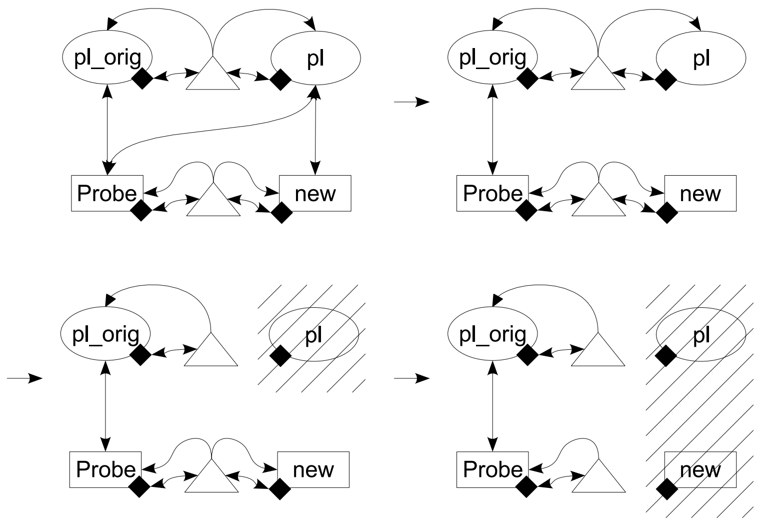


Figure 4.7: `dismiss()` *breaks all links between "live" objects and those that aren't needed any longer. Objects that can be reclaimed by the garbage collector are covered by hatching.*

When an error interrupts processing or the user decides to cancel a job, MPF Applicator will call the `dismiss()` method of all MDO instances involved in the process to remove unnecessary objects from memory. To make sure that the space occupied by these "dead" objects can be reclaimed by the garbage collector, all connections linking "live" objects to these objects must be severed. These include links from `ProbeLists` to `Probes` and vice-versa as well as links from `MIOGroups` to `Probes` and `ProbeLists`. The diagram shows the three steps (disconnecting new probes from the detached probe list, disconnecting the detached probe list's MIOs from their `MIOGroups` and disconnecting the new probes' MIOs from their `MIOGroups`). As a result, the detached `ProbeList`, all new `Probes` and all newly created MIOs are no longer accessible from live objects and can therefore be reclaimed by the garbage collector (see figure 4.7).

Integrating processing output into Mayday: reintegrate()

While a processing job is running, all active modules are only working on MDOs, that is they are performing changes on `Probe`s that aren't part of Mayday's `MasterTable` and on `ProbeList`s that are not members of the corresponding `DataSet`. After completion of a processing job, the final results need to be integrated into these structures to make them visible to Mayday (see figure 4.8). Integration of a MDO consists of multiple steps:

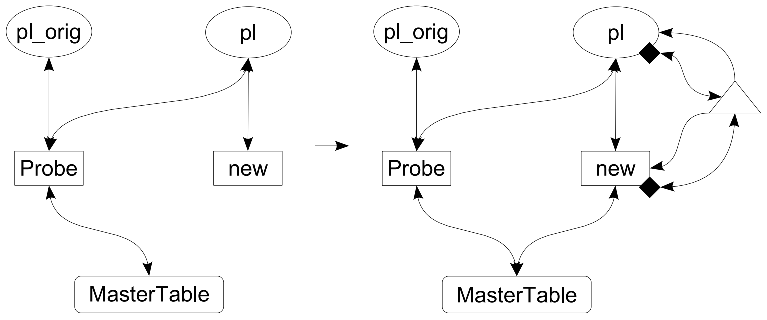


Figure 4.8: The situation before and after integration.

1. The `ProbeList` is given its final (unique) name made from the name of the input probe list and the name of the applied processing module or pipeline. (During the execution of the pipeline, the name is made unique by a temporary name modifier.)
2. The `ProbeList` is given an annotation and a meta information object describing the applied processing module and its relevant option values. This MIO is the first member of a newly created `MIOGroup`.
3. All `Probe`s created during the MDO's lifetime have to be processed:
   (a) The `Probe` is given an annotation and is added to the new `MIOGroup`.
   (b) The uniqueness of the probe name is checked again
   (c) The `Probe` is added to the `MasterTable`
4. (The `ProbeList` is added to the `DataSet`. This is not done by the MDO but after the Applicator is closed, i.e. after the call to `mayday.core.Pluggable.run(...)` returns.)

4.2 The Applicator

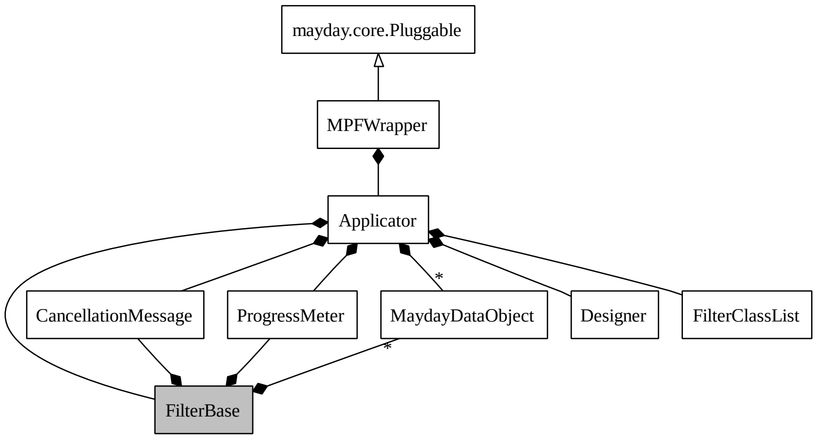


Figure 4.9: UML diagram of the classes used by the MPF Applicator.

The MPFs main class `Applicator` is connected to the Mayday plugin interface via `MPFWrapper` which implements `mayday.core.Pluggable`. `Applicator` uses the `FilterClassList` to get a list of available processing modules and `Designer` is used for creating and editing processing pipelines. When a job is being run, `Applicator` instantiates a `FilterBase` descendant and supplies it with instances of `CancellationMessage` and `ProgressMeter`. All input and output of the processing module is encapsulated in `MaydayDataObject` instances.

4.2.1 `MPFWrapper`

This class contains the Annotation for the MPF plugin as well as the plugin's entry function `run(...)` as described in the `mayday.core.Pluggable` interface. It creates MDOs for every input `ProbeList`, calls the `Applicator` and returns its output via the return value of its `run(...)` method.

4.2.2 `FilterClassList`

`FilterClassList` implements a singleton object containing all currently available processing modules. It basically does for the MPF what the `PluginManager` does for Mayday itself and, in fact, uses the PluginManager to find those processing modules that are implemented as Java classes. In addition to those basic modules, it also finds all processing pipelines that are stored in the <PLUGINS>`/mpf/` directory, parses them and adds them to the module list.
The `Applicator` and `Designer` classes use this class to get a list of all available modules while the `ComplexFilter` class needs it to recursively instantiate modules contained in processing pipelines.

4.2.3 `Applicator`

`Applicator` is the MPF's central class in terms of user interaction. From its main window, users can perform the following tasks:

- Apply a processing module to the selected probe lists
- Access the Designer to create a new processing pipeline or to edit an existing one
- Delete a processing pipeline or basic processing module

I will now concentrate on the process of applying a module to the selected input probe lists. The MPF Designer will be discussed later.
After choosing one or more probe lists from a data set in Mayday's main window, the Applicator is invoked by right-clicking in the data set window and selecting "Mayday Processing Framework" from the "Data Processing" submenu of the context menu that appears. The Applicator implements a "wizard" (also called "assistant") type of GUI that guides users through the three steps necessary before a processing module can be run. Together with running the module and interpreting the output, the complete process can be divided into five steps.

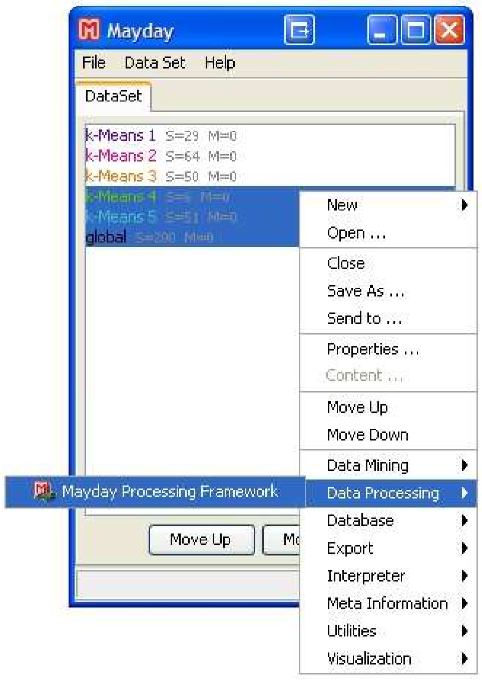
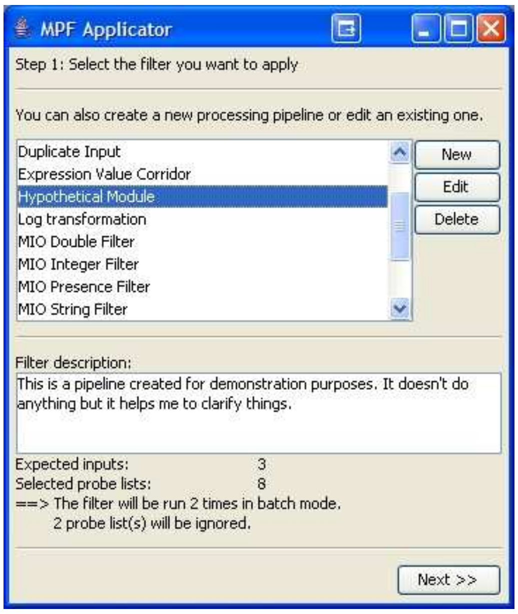


Figure 4.10: Invoking the MPF Applicator from Mayday

Step 1 – Selecting a module
In the first step, the user has to decide which processing module to apply to the selected probe lists. He can select an existing module (top left of the window), for example a basic module or a processing pipeline provided as a part of the MPF's base installation). He could also decide to modify an existing pipeline to better suit his current needs or, if no preexisting pipeline covers those needs, he could use the MPF Designer to create a new one based on basic processing modules or pipelines already available (top right).
In the lower part of the window, the Applicator gives some information about the currently selected module: The module's description is meant to help users decide whether a particular module is the right one for the job they have in mind. There's also an indication whether the supplied number of inputs is enough to run the module or whether the user selected too many inputs, in which case a batch job will be created.
In this example, I selected 8 probe lists but the module I want to apply only expects three input probe lists. Thus, Applicator will create a batch job consisting of running the module twice (thereby consuming 6 input probe lists) and discard the remaining two input probe lists.



Step 1: Select the filter you want to apply

You can also create a new processing pipeline or edit an existing one.

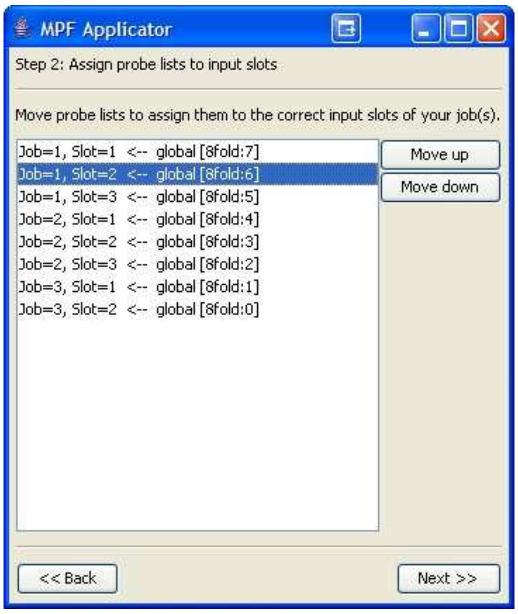
Duplicate Input
Expression Value Corridor
Hypothetical Module
Log transformation
MIO Double Filter
MIO Integer Filter
MIO Presence Filter
MIO String Filter

Filter description:
This is a pipeline created for demonstration purposes. It doesn't do anything but it helps me to clarify things.

Expected inputs: 3
Selected probe lists: 8
==> The filter will be run 2 times in batch mode.
2 probe list(s) will be ignored.

Figure 4.11: Module selection dialog with information on the selected module

Step 2 – Assigning probe lists to input slots
If a processing module expects more than one input, those different input slots need not be semantically equivalent. Thus there has to be a way to decide which of the selected probe lists will be used for a particular input slot. For a job with n input slots and m selected probelists ($m \geq n$), there are $\frac{m!}{(m-n)!}$ possible assignments. When in batch mode, this number grows even larger because now there are n slots for every job. Let k be the number of jobs then we have $n \cdot k$ input slots and m selected probe lists ($m \geq k \cdot n$) yielding $\frac{m!}{(m-kn)!}$ possible assignments.
The second step allows the user to setup this assignment. It is not necessary for modules with only one input slot and will be skipped in those cases.
To assign an input probe list to a particular input slot of one of the jobs in batch mode, the user only has to select one of the input probe lists and move it up or down in the list until its assignment is correct.



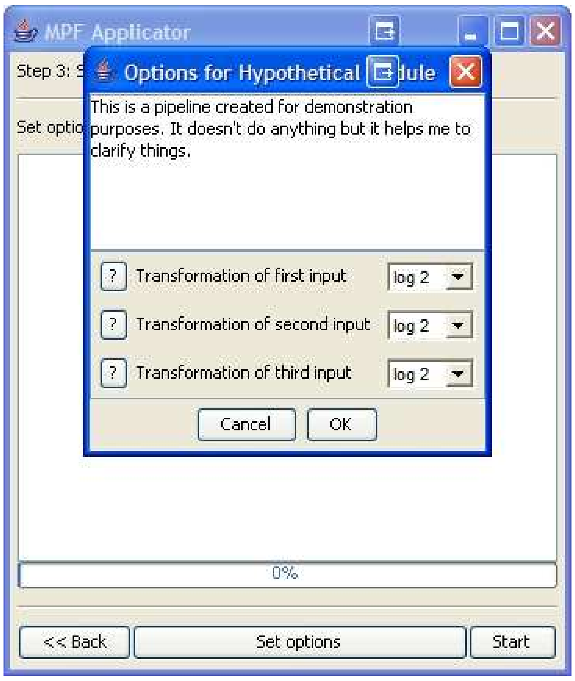
Step 2: Assign probe lists to input slots

Move probe lists to assign them to the correct input slots of your job(s).

```
Job=1, Slot=1  <--  global [8fold:7]
Job=1, Slot=2  <--  global [8fold:6]
Job=1, Slot=3  <--  global [8fold:5]
Job=2, Slot=1  <--  global [8fold:4]
Job=2, Slot=2  <--  global [8fold:3]
Job=2, Slot=3  <--  global [8fold:2]
Job=3, Slot=1  <--  global [8fold:1]
Job=3, Slot=2  <--  global [8fold:0]
```

Figure 4.12: Input assignment dialog

Step 3 – Setting processing options
Before the (batch) job is started, the user is asked to verify the options of the chosen processing module. For this, the Applicator calls the module's `ShowOptions` method which is translated into a call to the module's `FilterOptions.ShowWindow` method. All options are at their default values. After making the appropriate changes, the user closes the option dialog and processing can start. If a module has no options, processing can start right away.
In this example, the Hypothetical Pipeline does a log transformation on its input probe lists. Users can select how each of the input probe lists will be transformed.

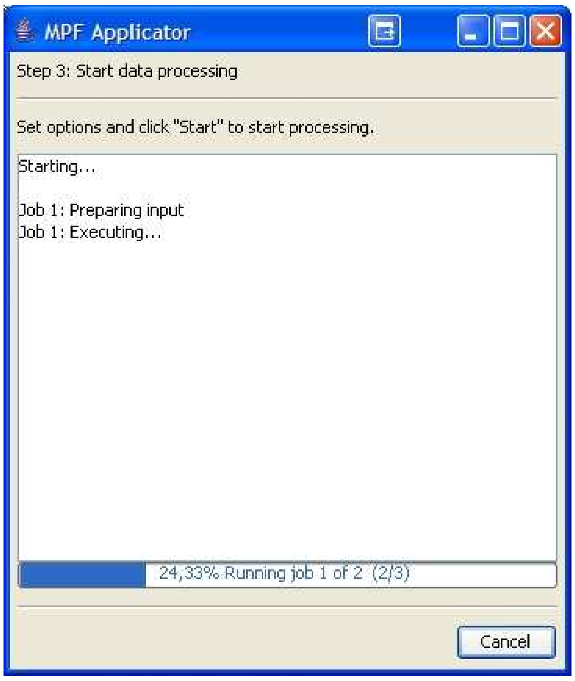


This is a pipeline created for demonstration purposes. It doesn't do anything but it helps me to clarify things.

Transformation of first input: log 2
Transformation of second input: log 2
Transformation of third input: log 2

Figure 4.13: Dialog window showing the module's description and its options

Step 4 – Running the job(s)
Once processing is started, the log window keeps the user informed of the current state of his job(s). It contains a progress bar showing the overall percentage completed as well as a message window where error messages can be displayed.
Here we can see that the first of our two jobs is currently running (log window). The progress bar tells us that 24% of all jobs have been completed and that we are currently running the first of two jobs. In this case, the first job is a processing pipeline and the second of three submodules is being executed at the moment.



Step 3: Start data processing

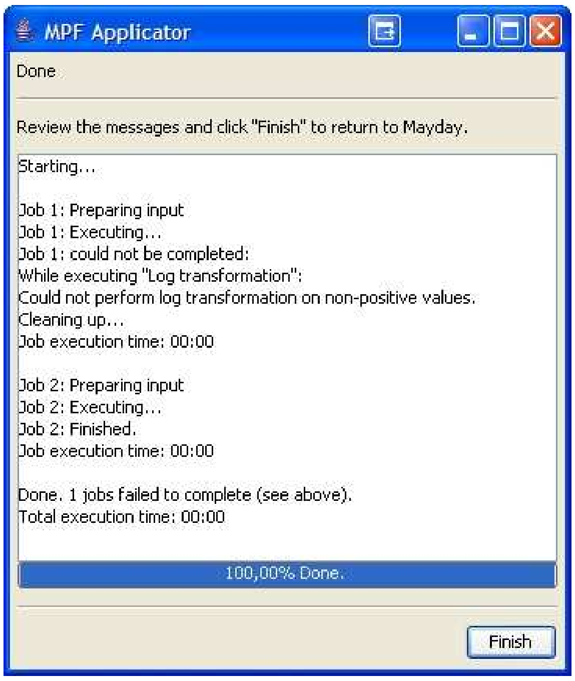
Set options and click "Start" to start processing.

```
Starting...

Job 1: Preparing input
Job 1: Executing...
```

24,33% Running job 1 of 2 (2/3)

Figure 4.14: Progress information and log messages are displayed during job execution.

Step 5 – Reviewing the log
After processing has finished, users can review the log messages to see whether all their jobs completed successfully. Error messages that are written into the log by processing modules should be descriptive enough for users to understand why a job failed.
In this example, we can see that the first job failed because the probe list contained non-positive values that could not be log-transformed
Note that the second job was completed successfully despite the failure of the first job, preventing one failed job from causing the whole batch job to stop.



Done

Review the messages and click "Finish" to return to Mayday.

```
Starting...

Job 1: Preparing input
Job 1: Executing...
Job 1: could not be completed:
While executing "Log transformation":
Could not perform log transformation on non-positive values.
Cleaning up...
Job execution time: 00:00

Job 2: Preparing input
Job 2: Executing...
Job 2: Finished.
Job execution time: 00:00

Done. 1 jobs failed to complete (see above).
Total execution time: 00:00
```

100,00% Done.

Figure 4.15: The log window

I will now explain how batch jobs are executed, focusing mainly on error handling. For the purpose of this description, please note that a single job is a special case of batch job, containing only one "sub-job". A batch job is processed according to the following pseudo code:

```
          Reorder input according to the user's slot assignments (see Step 2)
          While there are jobs to do and the user hasn't canceled processing do
             Assign input probe lists to input slots
             Create a new child instance of ProgressMeter
    (1)      Run the module
             If the module was canceled by the user
                Clean up memory
             else
                Get the module's output probe lists
                Rename and annotate them
    (2)         Add them to the global list of output probe lists
             fi
          od
          For each output probe list in the global output list do
    (3)      Integrate the probe list into Mayday's data structures
          od
```

There are three places where errors can occur (indicated by (1)–(3) above). Different things can happen at those points and they are dealt with in different ways:

(1) The processing module throws an `Exception`. This should only happen if the module is not applicable to the type of input it was given, e.g. when trying to perform log transformation on a matrix that contains negative values. In this case, the current job is canceled, any objects created during its execution are destroyed and memory is cleaned up. The Applicator then

proceeds to the next job.
A second source of problems could be that the Java VM is running out of memory and throws an `OutOfMemoryError`. If that happens, the current job is canceled, created objects are destroyed, memory is freed and the whole batch job is canceled because if memory is depleted there is no reason to try executing another job. The user is informed of the problem. In both cases, output from all successful jobs is kept for later integration into Mayday.

(2) The only case where an error could be expected here would be if the job terminates successfully but memory is so low that its output can't be added to the global output list. As this operation only takes very little memory, errors should only occur very rarely here, if at all.

(3) Integrating output probe lists into Mayday can be a time- and memory-consuming task (see the detailed description of the process in the section about `MaydayDataObject`). Should the VM run out of memory during the process, there is no way to integrate the remaining output into Mayday. The user is informed of the problem and all processing is terminated.

4.3 Processing pipelines

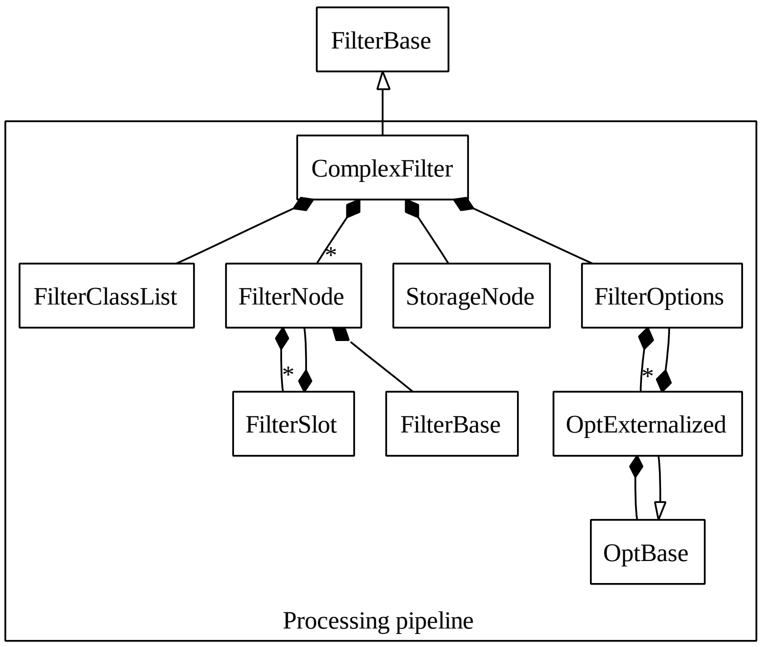


Figure 4.16: UML diagram of all classes used in processing pipelines.

Every processing pipeline is represented by an instance of `ComplexFilter` and, as such, descends from the `FilterBase` class. This makes processing pipelines the same as basic processing modules, that is, from the MPF's point of view they are black boxes whose contents don't need to be known for most purposes. The pipeline is implemented as a directed, acyclic graph of processing modules. The nodes in this graph are of type `FilterNode`, where each node has a certain number of incoming edges connected to its *input slots* and a certain number of outgoing edges connected to its *output slots*. Every input/output slot is represented by a `FilterSlot` instance. Every `FilterNode` represents one descendant of `FilterBase`. `ComplexFilter` uses `StorageNode` structures to save its pipeline to disk and accesses the singleton `FilterClassList` object to instantiate sub-modules when loading the pipeline description from disk.

A processing pipeline doesn't have any options as such, but the designer of a pipeline can choose to present options from its submodules to its users. These options are called *externalized* options and are all represented by instances of `OptExternalized`. Every object of this type encapsulates one `OptBase` descendant in a transparent way. This allows the externalized options of one processing pipeline to be externalized again upon its inclusion into another pipeline.

4.3.1 `ComplexFilter`

The `ComplexFilter` class holds a graph description in memory, saves it to and loads it from a file, makes sure that the graph can be executed and executes the graph. I will now describe how these things are done.

Graph representation

The filter graph of a processing pipeline consists of n nodes where each node represents a processing module (either a basic module or another instance of `ComplexFilter`). Every node has a fixed number of input slots (attachment points for incoming edges) and output slots (the same for outgoing edges). These numbers depend on the `FilterBase` descendant that the node represents. To model the pipeline's input and output slots, two additional, "special" nodes are used, called *global input* and *global output*. When the pipeline is executed, the MDO's it receives as input are connected to the *output* slots of the global *input* node. After execution of all submodules, the *input* slots of the global *output* node are returned as the output objects of the processing pipeline.

Edges in the graph are not represented explicitely as objects in memory. Instead, every input/output slot of a node is an instance of `FilterSlot` storing information about the node at the "other end" of the edge starting at the given slot. This information contains

- the target `FilterNode` instance and
- the slot number of the slot that the connection points to.

There are two equivalent representations of these connections in the graph: The obvious choice is to have a pointer to the target node and an integer specifying the target slot number. This representation is fine for most purposes but it can not be used directly to store the graph to disk because there's no way to recreate the connections from stored pointers. Thus, an alternative representation is used for that purpose:

`ComplexFilter` contains a method that traverses the graph and assigns *node indices* to its nodes. This is done in such a way that for two nodes with indices k and ℓ, $k < \ell$ is only true if ℓ has no incoming edges from any node m with $m \geq k$. To achieve this useful node ordering, a tree traversal algorithm is used that is loosely related to a *depth-first* approach. The difference to normal DFS lies in the structure of the graph: Every node can have multiple incoming edges. In fact, for a pipeline with more than one global input slot, we get a graph with more than one "root" node. To make sure that the ordering follows the condition stated above, the algorithm may only process a node's subtree after *all* incoming edges of that node have been assigned (as shown in figure 4.17). The algorithm is as follows:

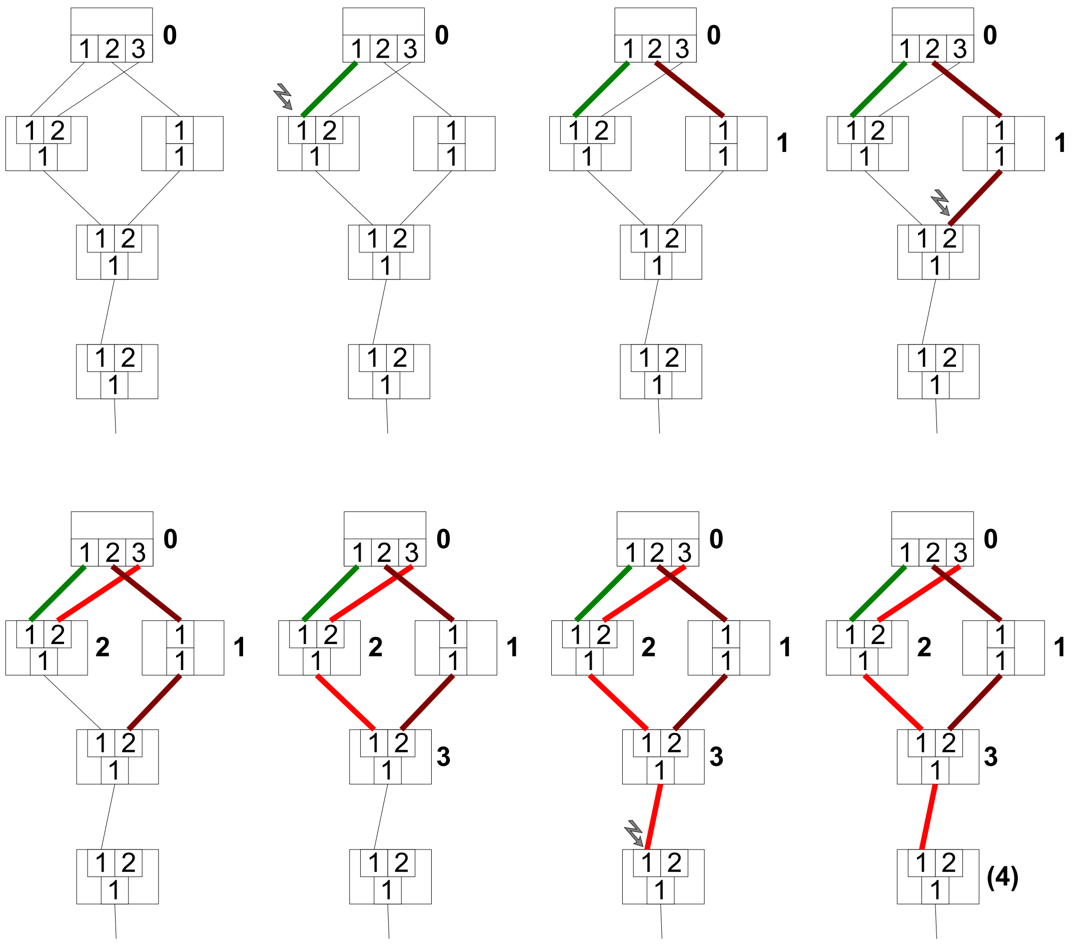


Figure 4.17: Graph traversal: Starting from the global input, "chains" starting at the (in this example: three) output slots are followed until a node is reached where the chain can't be continued (lightning symbol). Whenever a node has all the input it needs, the node is assigned the next free node index and the chain can continue. After all chains starting from the global input are done, remaining nodes are given node indices (here it is node (4)). These are the nodes that lack part of their expected input.

```
Set startingNode = global Input
Set nextIndex = 0

buildchain(startingNode) {
    Set node index for startingNode to nextIndex
    nextIndex++
    For each edge leaving startingNode do
        Let childNode be at the other end of the edge
        Increment the incoming connection counter of childNode by one
        If childNode has all the incoming connections it needs
            buildChain(childNode)
        fi
    od
}
```

The implementation in ComplexFilter is slightly different because it has to take into account the possibility of having empty slots, i.e. slots that are unconnected. In that case, the graph is not valid and the pipeline can't be executed, but the algorithm must still assign unique node indices to all nodes. This is done by keeping a list of nodes that have not been assigned a node index. After the algorithm has finished traversing the tree, those nodes are assigned the indices nextIndex, nextIndex+1, . . ., nextIndex+n.

Loading, saving and executing a graph

The node ordering is used when the graph is loaded from/saved to a file as well as during the execution of the processing pipeline:

- *Saving the graph*
  ComplexFilter uses the StorageNode class to store the processing pipeline. Along with information such as its *name*, its *description*, the number of *global input and output slots* and the *version number*, the graph structure is stored. For this, a StorageNode is created that contains one child for every FilterNode in the graph. Such a child stores the name, version and option values of the FilterBase instance represented by the FilterNode as well as the target node index and target slot index of all *incoming* connections. Outgoing connections are not saved explicitely for two reasons:
  Firstly, every graph edge is completely defined by four characteristics: starting node index, starting slot index, target node index and target slot index. The latter two are stored in a StorageNode and the first two are known implicitly from the fact that this StorageNode lies in the subtree of a particular node index (the starting node index) and that this incoming connection is the nth connection in the corresponding FilterNode's connection list (with n being the starting slot index). So there is no need to store those values explicitly.
  Secondly, if outgoing connections were also stored in pipeline descriptions, extensive testing would be necessary to make sure that there are no conflicting connections in the file (for example due to file corruption).
  I will use an adapted version of the Backus-Naur form (BNF) to describe the format of a pipeline description. Here, every StorageNode is a non-terminal whose name is the StorageNode's name. "x*" means that x can occur once, several times or not at all. The value part of each StorageNode is indicated in curly braces {} where appropriate. For nodes with variable names, the name is indicated in normal brackets ().

```
root       =    <Name> <Desc> <Ver> <InputSize> <OutputSize> <Options> <Subfilters>
Name       =    {The name of the processing pipeline}
Desc       =    {The detailed description of the pipeline}
Ver        =    {The version number of the pipeline, a positive nonzero integer}
InputSize  =    {The number of input slots, a positive nonzero integer}
OutputSize =    {The number of output slots, a positive integer including zero}
Subfilters =    {}  <Subfilter>*

Subfilter  =    (The node index, a positive integer)  {}
                <FilterName> <Ver> <InputFromIndex> <InputFromSlot> <Options>
FilterName =    {The name of the submodule, a positive nonzero integer}
Ver        =    {The version number of the submodule}
InputFromIndex= { <IntegerList> | <> }
InputFromSlot = { <IntegerList> | <> }
IntegerList   = <Integer> | <Integer> , <IntegerList>
```

```
Options    =    {}  <Option>*
Option     =    (The option's index in the parent FilterOptions instance)
                {The option's value converted to String representation}
```

- *Loading the graph*
  To load a graph from disk, `ComplexFilter` has to perform the following steps. The order of these steps is very important because later stages depend on the results of earlier ones, e.g. correctly initialized objects etc.

  1. Load the pipeline's *name*, *description* and *version* number
  2. Go over all subnodes of the `Subfilters` node.
     (a) Use the `FilterName` attribute to instantiate the correct submodule from the list of available modules in `FilterClassList`. If the expected module is not in the `FilterClassList`, the pipeline cannot be loaded. If it is there but has a different version number, a warning message will be given to the user and loading will continue albeit with no guarantee as to its successful completion.
     Special care has to be taken in the case of the global input and global output nodes. Those can not be instantiated from `FilterClassList` but have to be constructed with the appropriate number of slots as specified in the `InputSize` and `OutputSize` values.
     (b) Load the incoming connections to initialize the `FilterNode`'s `FilterSlot` objects.
     (c) Let the `FilterNode`'s `FilterBase` object load its options from the subtree below the `Options` node.
     (d) Set the submodule's node index and add it to the list of submodules.
  3. Use the incoming connections of each `FilterNode` to set up the graph, converting node indices to pointers and adding outgoing connections.
  4. Load the externalized options of the processing pipeline from the `Options` node. This has to be done after all submodules are initialized and their respective `OptBase` descendants instantiated (see 4.3.4).

- *Executing the graph*
  The node indices are very helpful for executing the processing pipeline: `ComplexFilter` only has to execute the submodules in the order of their node indices to make sure that the input for every module is ready by the time it is executed. Execution consists of

  1. assigning input to a submodule's input slots,
  2. assigning the processing pipeline's `CancellationMessage` to the submodule,
  3. creating a new child instance of the pipeline's `ProgressMeter` with the correct *baseline* and *factor* and assigning it to the submodule,
  4. calling the submodule's `execute()` method,
     (If an `Exception` is thrown by the submodule, it is caught and a new Exception is thrown. That new Exception contains the text of the old one together with a line describing the context that the problem occurred in, i.e. the name of the pipeline.)

  and repeating these steps for all submodules. Note that after all output from one module k has been handed over to other submodules in step 1, k and all its data can safely be left for the garbage collector to clean up.

Validating a graph

In a valid graph, indices can be assigned to all nodes during graph traversal. There are some more criteria that a valid graph has to fulfill. The complete list is this:

- All nodes are properly connected.
  That is, every `FilterSlot` object in the graph must point to one slot of another `FilterNode` and no two `FilterSlot`s may point to the same node.
  The proper connection of all input slots can be tested after `buildGraph()` has assigned node indices to all nodes. If any node could not be assigned a node index during the graph traversal, the graph is invalid. Checking the input slots is not sufficient, though. The output slots must also be checked because there could be less input slots than output slots in the graph. That way, all input slots could be properly connected while some output slots would be left unconnected.
  Unconnected input slots would mean that a processing module has not enough data to perform its job. Depending on the implementation of the module, it would either fail completely or it might just work, creating incorrect and unexpected results in the process. For this reason the MPF guarantees that no processing module is executed without the proper number of input MDOs. Module implementers do not need to check for missing input and module implementation is less complicated.
  Unconnected output slots, on the other hand, would mean that results are lost during the execution of the pipeline. This may not be a problem in certain situations, e.g. when executing a processing module has side-effects like creating an image file and the actual output is of no interest. Apart from the fact that “dropping” results is not desired most of the time, there’s another compelling reason to prevent it from happening: The dropped MDO would neither be integrated into Mayday nor would its `dismiss()` function ever be called. Thus, while not visible to users of the program, the MDO would take up memory because from the garbage collector’s point of view it would still be a “live” object, connected to the “visible world” by shared `MIOGroup`s and by `Probe`s that are also part of other `ProbeList`s.

- Output slots connect to input slots
  It is obvious that in a directed graph every edge is an outgoing edge at one node and an incoming edge at another node. Thus, `ComplexFilter` has to make sure that no connections can be made between two output slots or two input slots. During the creation of the pipeline in Designer, the user has to select an output slot before selecting an input slot. The connection between the two slots is made and no other way of connecting nodes exists.
  But what if a malicious user really wants to create that kind of connection and tries to manipulate the graph description file? The fact that the serialized form of the graph only contains incoming edges doesn’t allow for any such change. Of course, connection information in the graph description file could become corrupted due to hardware problems but `ComplexFilter` checks the connections while loading the file and tries to recover gracefully when something is wrong, so that users can open defective pipelines in the MPF Designer and restore all connections.

- The graph is cycle free
  If the graph of a processing pipeline were allowed to contain cycles, running that pipeline would result in an infinite loop. Preventing cycles is therefore a major part of graph validation. The problem of cycle detection can be broken down into two distinct problems that need different strategies:

  1. Cycles within one processing pipeline
     This is the obvious kind of cycle that is visible on-screen in the graph. A path starting

from one of the output slots of a node A ends in one of the input slots of a node B and vice-versa. This kind of cycle is easily detected because `buildGraph()` can't assign node indices to either of the two nodes:

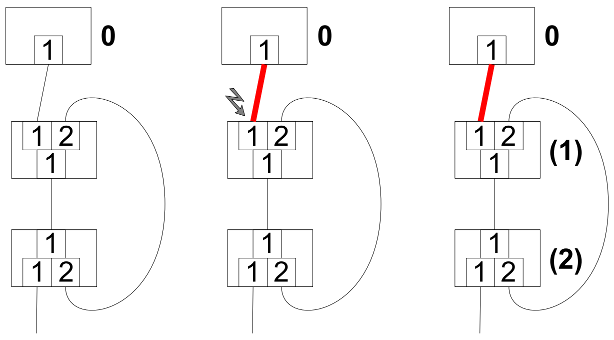


Figure 4.18: A graph with cycles breaks the chain before any node in the cycle is assigned a node index.

2. Cycles that span several processing pipelines
   Consider two very simple processing pipelines A and B. Each pipeline only contains one processing module. Let A contain B and let B contain A. Keep in mind that a processing pipeline is an instance of `ComplexFilter` and, as such, only another descendant of `FilterBase`. When a processing pipeline is embedded within another pipeline, special attention has to be paid to that nested pipeline.

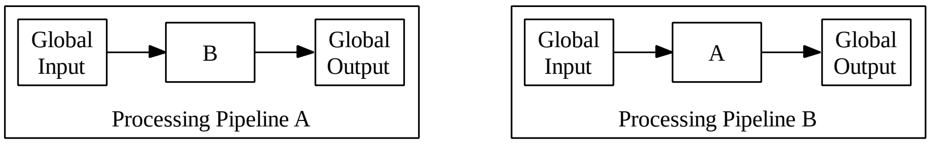
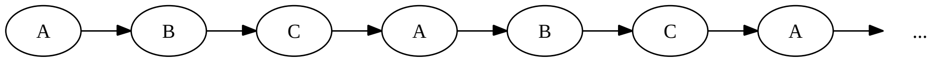


Figure 4.19: An example of nested pipelines that create an infinite recursion.

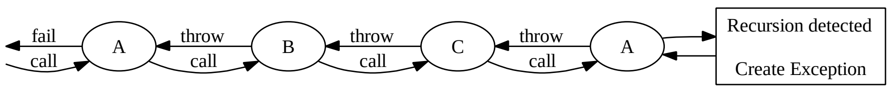
While cycles within one pipeline lead to infinite loops during filter execution, this kind of cycle that spans two or more pipelines leads to an infinite recursion. When instantiating a `ComplexFilter` object for pipeline A, the constructor needs to create an instance for the submodule B. So a new `ComplexFilter` object is created for the pipeline B. Before the constructor of B can return, it needs to construct a new instance of `ComplexFilter` for its own submodule, i.e. a new instance of pipeline A and so forth.
To prevent this, a static variable was added to the `ComplexFilter` class. This variable contains a stack of pipeline names that are currently being constructed. The `ComplexFilter` constructor can now check whether it is being called from within an infinite recursion loop and can, if that is in fact the case, terminate that loop. The pseudo-code for the constructor clarifies the method:

```
static Stack RecursionStack

ComplexFilter(FileName) {
    if FileName is contained in the RecursionStack
        Infinite recursion found. Cancel with an exception
    else
        Push FileName onto RecursionStack
        Construct the ComplexFilter instance
        Pop FileName from RecursionStack
    fi
}
```

With this method, an infinite recursion results in an `Exception` being thrown as soon as the cycle has been entered for the second time. This exception is then propagated up the call hierarchy, cancelling all enclosing calls to the constructor. Let's look at a more complex example where we have a pipeline A containing a pipeline B containing a pipeline C which in turn contains pipeline A. Without checking for recursive invocation, the program would be caught in an infinite recursion until all available resources are exhausted. This would likely result in a termination of Mayday by the Java VM:



With the protection in place, the process would look like this:



During pipeline design, the user must not be able to insert a new submodule if that submodule would create an infinite recursion. To that end, Designer also uses the `ComplexFilter` static variable: After the pipeline has been loaded into Designer, the stack is empty. Designer then pushes the pipeline onto the stack again. If the user now tries to place a submodule that would create an infinite recursion, that submodule's constructor fails because at one point the constructor of the pipeline currently being edited is invoked, finds itself in the stack and terminates with an exception. Thus the user can not create this kind of cycle.

4.3.2 `FilterNode`

Objects of the `FilterNode` class represent a node in the filter graph of a processing pipeline. Every node contains a reference to the `FilterBase` descendant it represents. It stores the current node index, and two lists of `FilterSlots`, one for input and one for output slots.

Most of the methods defined in the `FilterNode` class are needed for user interaction and will be discussed in the context of the MPF Designer later. These are methods for creating connections between nodes, for changing the number of input or output slots as well as a method that returns the `VisualNode` instance representing this `FilterNode` in the GUI (creating such an instance if necessary, i.e. lazy creation).

The remaining methods are needed in the context of graph setup and validation. These are:

`void setNodeIndex(int)`	Sets the node's index. This is called by `ComplexFilter.buildGraph()`.
`boolean connectInput(int slot, int source)`	Set the index representation of a connection. A connection exists between this node's input slot `slot` and another node X's output slot k. During `ComplexFilter.buildGraph()`, this method is called to replace the pointer to X with X's node index `source`. The function returns true if this was the last missing input of the `FilterNode`, i.e. a return value of true indicates that the chain can continue below this node, false indicates that the chain has to be broken here (see 4.3.1).
`boolean validateConnections()`	checks whether all slots are connected. This is called from `ComplexFilter.validateGraph()` and is part of ensuring the first requirements for valid graphs (see above).

4.3.3 `FilterSlot`

This class represents one outgoing or incoming edge at a `FilterNode`. Edges are not modeled as separate objects but rather by a pair of `FilterSlots`. Every `FilterSlot` instance contains three values and some functions that simplify using them. The values are:

- A pointer to the target node of the edge connected to this slot
- The index of the slot on the target node that the edge connects to
- The node index of the target node (valid only after `ComplexFilter.buildGraph()` has been called).

Consider two nodes A and B and a directed edge $E = (A[1], B[3])$ connecting the first output slot of node A to the third input slot of node B. This is represented by a pair of `FilterSlot` objects as shown on the left side of figure 4.20.
After the graph has been built by `ComplexFilter.buildGraph()`, let's assume that A received the index 5 and B received the index 7. The result is shown on the right side of the figure.

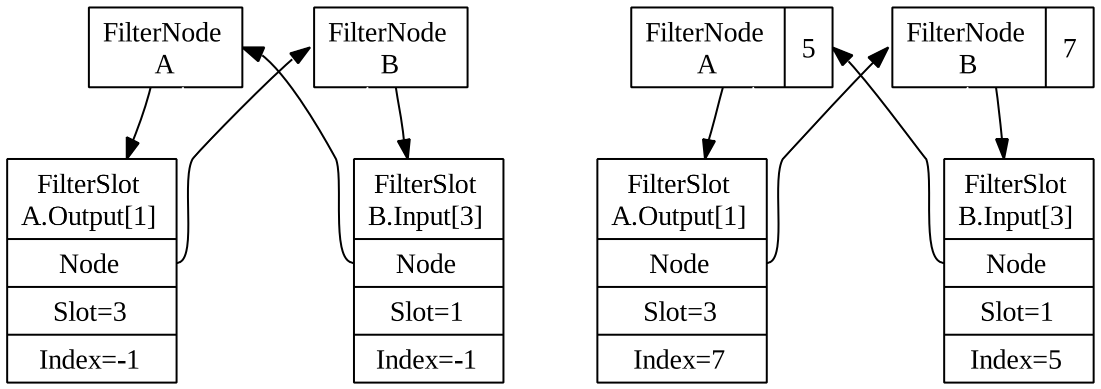


Figure 4.20: Edge representation before and after `buildGraph()` *has assigned node indices.*

4.3.4 `OptExternalized`

When users create processing pipelines using the MPF Designer, they combine several existing processing modules or pipelines into a larger module. Some of the submodules have options that can be set, for example the base for a logarithmic transformation. Often it is sufficient to set the value of that option in the Designer and then always use the pipeline with the same value for that option.

But sometimes it is more appropriate to allow the value to be changed prior to the pipeline's execution. An example for that would be a pipeline that performs some data processing and then filters the data based on attached meta information, e.g. to only keep probes for those genes that belong to a particular regulatory system. This was done for the "Spellman Pipeline" I will present in section 6.

To make a submodule option visible to users of a pipeline, the option has to be *externalized*. This means that a new object of type `OptExternalized` is added to the `FilterOptions` object of the enclosing `ComplexFilter` instance. `OptExternalized` is a wrapper that can transparently incorporate any other option as long as it descends from the `OptBase` class. This includes instances of `OptExternalized` itself when a pipeline is used as a submodule of another pipeline and one of its externalized options is made an external option of the enclosing pipeline.

Requirements

Normally, an externalized option is taken out of context from the users' point of view. Consider a pipeline that, among many other submodules, contains a module for logarithmic transformation. If the creator of that pipeline decides to externalize the option for the logarithm base, just presenting the original `OptDouble` as an option of the pipeline isn't likely to be the correct choice. Most of the time, the option will have to be renamed or its description changed to tell users more about *why* that particular option can be manipulated and *how* changing its value will affect the result of applying the pipeline to their data.

This gives the requirements for `OptExternalized`:

1. Instances must have their own `name` and `description`,
2. they must also have a second name, the "long name" that gives information about the *wrapped* option,
3. GUI elements must come from the wrapped `OptBase` descendant, and
4. changes to the option's value must be made directly to the wrapped object.
5. The requirements have to be met in a transparent fashion so that there's no limit to the nesting of `OptExternalized` instances.

Serialized representation

As a consequence of these requirements, some of the methods defined in the `OptBase` class can simply be wrapped by `OptExternalized` while others need more effort. Before discussing those methods, note that an externalized option is represented by three values:

1. `parentFilter` – A pointer to the instance of `ComplexFilter` that the *externalized* option belongs to,
2. `SubfilterNode` – A pointer to the `FilterNode` that the *wrapped* option belongs to

3. `OptionIdx` – the index of the *wrapped* option in the `SubfilterNode`'s `FilterOptions` member object.

The values of `SubfilterNode` and `OptionIdx` completely define the wrapped option which can now be obtained by calling

`SubfilterNode.attachedFilter.Options.get(OptionIdx)`.

The `parentFilter` is needed when deserializing the externalized option from a pipeline definition file. The externalized option is saved as a `StorageNode` whose value is

`<Name>|<Description>|<SubfilterIndex>|<OptionIndex>`.

When loading an externalized option from disk, the value of `<SubfilterIndex>` can be used together with the value of `parentFilter` (which is handed over as one of the arguments of the `OptExternalized` constructor that is called from `ComplexFilter.LoadFromStream()`) to restore the value of `SubfilterNode` because

`SubfilterNode = parentFilter.sortedFilters.get(SubfilterIndex)`.

This shows once again how vital the correct order of execution is when loading processing pipelines from disk: Externalized options have to be created *after* all `FilterNodes` are instantiated (along with their options) and the `sortedFilters` member of `ComplexFilter` is completely initialized.

Methods that are simply wrapped

To wrap a method `x`, `OptExternalized`'s implementation of `x` simply calls `getOption().x` where `getOption` is defined as `SubfilterNode.attachedFilter.Options.get(OptionIdx)`. The following `OptBase` methods are simply wrapped:

- `boolean setVisible(boolean)`
- `void notify(FilterOptions)`
- `boolean validate()`
- `void accept()`
- `void cancel()`
- `String getAnnotation()`

More complicated methods

To meet the third and fourth requirements on the list above, the GUI objects for changing the option (that is, its `EditArea`) are the same for the `OptExternalized` instance and for the wrapped `OptBase` descendant. As a result, `OptExternalized` doesn't need to know how to create GUI elements for the particular kind of option it encloses and changes to the GUI objects are immediately visible to the wrapped object because they are, in fact, performed on the wrapped object itself.

Thus the `createEditArea()` method calls the wrapped option's `getEditArea()` method (which in turn may call the wrapped option's `createEditArea` method. The identity of the edit areas is also used for comparisons: If two options point to the same `EditArea`, then they are the same.

The visibility of the externalized option is always set to true regardless of whether the wrapped object is currently visible. This is necessary because the visibility of a wrapped option O_1 could depend on the value of another option O_2 of the respective processing module. If the pipeline designer now decides to externalize O_1 but not O_2, O_1 could be invisible and the user would have no chance of making it visible again. Options that are always invisible, e.g. options that are computed by the processing module and aren't meant to be changed by the user will not only be invisible but their `externalizable` attribute will be set to `false` by the programmer of the module. They will not be eligible for externalization.

The "long name" mentioned in requirement two is computed as

```
<Name> ( <Name of the submodule> ( <SubfilterIndex> ):  <Name of the wrapped option> )
```

In addition to the methods defined in `OptBase`, `OptExternalized` also needs a method that allows changing the name and description of the externalized option as well as setting the default value for that option. This method is called `editMetaOptions()`. It uses the `FilterOptions`/`OptBase` set of classes to create a dialog window.

Saving and loading an externalized option has already been described.

4.4 The Designer

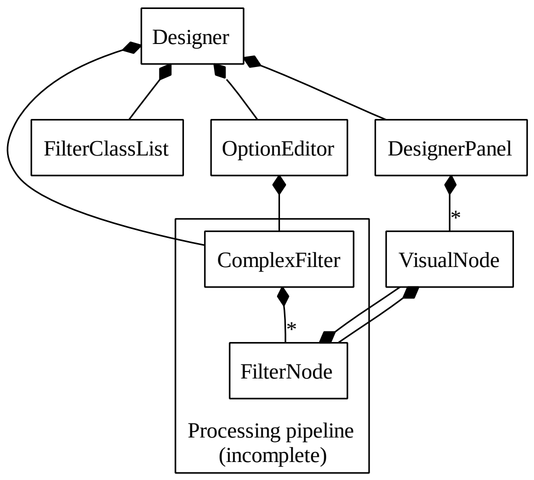


Figure 4.21: UML diagram of all classes used by the MPF Designer.

To edit a processing pipeline, the user calls the MPF Designer from the Applicator window. This leads to the construction of a new instance of the `Designer` class. The pipeline that is being edited is an instance of `ComplexFilter`. `Designer` creates a GUI window and uses `DesignerPanel`, a subclass of `javax.swing.JPanel`, to render the filter graph. Within `ComplexFilter`, every node in the graph is represented by an instance of `FilterNode`. For the graphical interface, every `FilterNode` is represented by a `VisualNode` object that takes care of creating all necessary GUI components. To allow changing the settings of the `ComplexFilter`, the `OptionEditor` class is used. New submodules can be added to the graph by selecting a processing module from the list maintained by the `FilterClassList` singleton.

4.4.1 `Designer`

`Designer` is the main class for the GUI that is used to create and edit processing pipelines. When the user decides to create a new pipeline by clicking on the "New" button in the Applicators main window, `Designer` is invoked with a new instance of `ComplexFilter`. When the user decides to edit an existing pipeline, `Designer` is called with an instance of that pipeline instead.
I will now describe the different actions users can perform in Designer together with the involved classes. For this description I will assume that a new pipeline has been created.
At first, the Designer window contains only two nodes, the global input and the global output node of the new processing pipeline.
The user can now perform four basic functions on the graph:

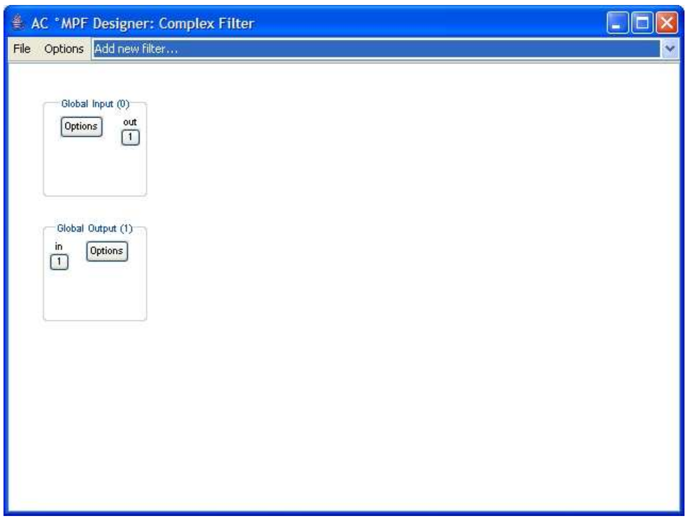


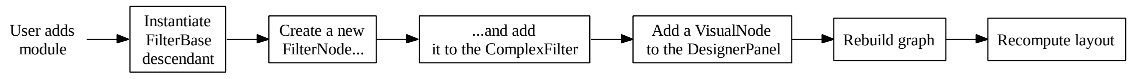
Figure 4.22: A new pipeline in MPF Designer.

- Add a module to the processing pipeline
- Create a connection between two modules
- Remove a module from the pipeline
- Change the options of a module

For each of these actions I will describe the necessary work on the user side as well as what happens inside the Designer.

Adding a module

To add a new processing module to the pipeline, the user only has to click on the drop-down list in the menu bar of the Designer window and select a module from the list.

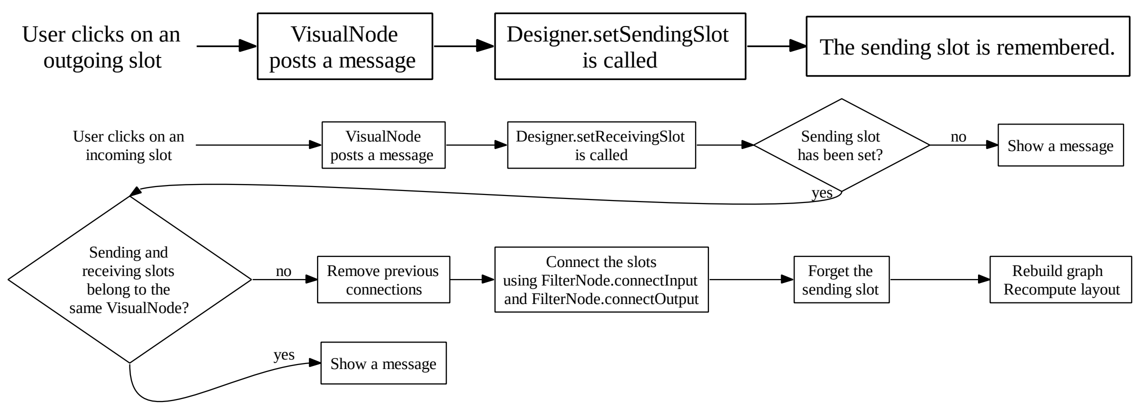

User adds module
Instantiate FilterBase descendant
Create a new FilterNode...
...and add it to the ComplexFilter
Add a VisualNode to the DesignerPanel
Rebuild graph
Recompute layout

Creating a connection between two modules

Creating a new connection is a two-step process: The user first has to click on an outgoing slot of one module and then click on an incoming slot of another module. Those two clicks are handled independently by the Designer.

With the first click, an outgoing slot is selected and stored. If the second click then selects an incoming slot, a connection between the outgoing slot selected before and this incoming slot is created. Should the first click be on an incoming slot, a warning message is issued to the user and nothing happens. If the second click is on an outgoing slot, it is regarded as a new "first" click.

If both clicks are valid, Designer checks whether one of the two selected slots is already used by another connection. If this is the case, the old connection(s) are removed. This involves disconnecting the selected node as well as the node at the other end of the connection (for the connection representation, see 4.3.3). Designer also makes sure that no object is connected to itself, thereby preventing the most trivial form of cycle.


User clicks on an outgoing slot
VisualNode posts a message
Designer.setSendingSlot is called
The sending slot is remembered.
User clicks on an incoming slot
VisualNode posts a message
Designer.setReceivingSlot is called
Sending slot has been set?
no
Show a message
yes
Sending and receiving slots belong to the same VisualNode?
no
Remove previous connections
Connect the slots using FilterNode.connectInput and FilterNode.connectOutput
Forget the sending slot
Rebuild graph Recompute layout
yes
Show a message

After the connection is made, the graph is built again and the layout is recomputed. Slots that are properly connected are shown with a green background (or border, depending on the operating system used), those that aren't are shown in red.

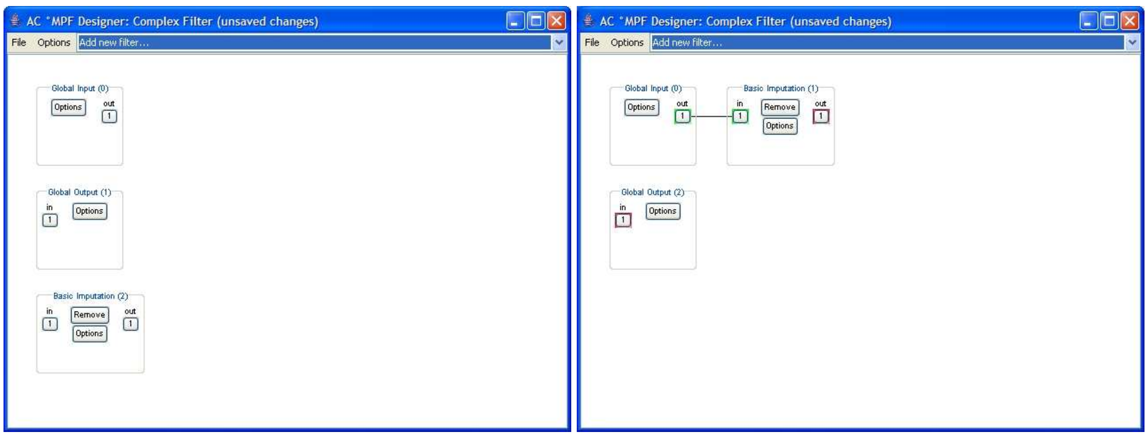
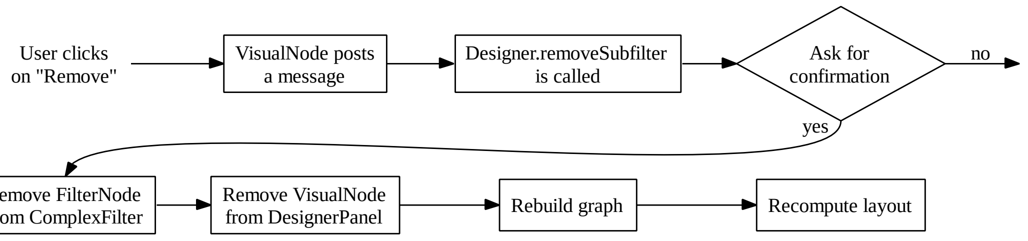


Figure 4.23: Connecting submodules, before (left) and after (right).

Removing a module

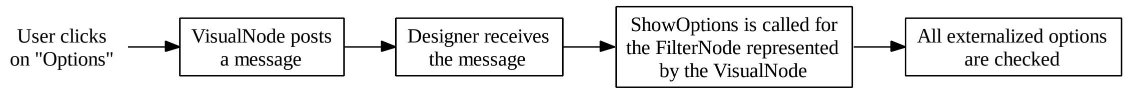
To remove a module, the user clicks on the “Remove” button contained in the module’s `VisualNode`. Designer removes all connections to the module’s input and output slots, removes the module and then rebuilds the graph and recomputes the layout.


User clicks on "Remove"
VisualNode posts a message
Designer.removeSubfilter is called
Ask for confirmation
no
yes
Remove FilterNode from ComplexFilter
Remove VisualNode from DesignerPanel
Rebuild graph
Recompute layout

Editing module options

To edit the options of a module, the user starts by clicking on the “Options” button of the `VisualNode` representing that module. The `FilterOptions` member of that module is called to create the option dialog window. After the window is closed, Designer has to check whether any options of that module that had been externalized are no longer valid.

For example this can happen when the `RWrapper` processing module is used: If the user selects a different R function from the `RWrapper`’s option dialog, the set of available options is changed to reflect the parameter list of that particular function. This means that if options of the previous R function had been externalized, those will no longer be valid.

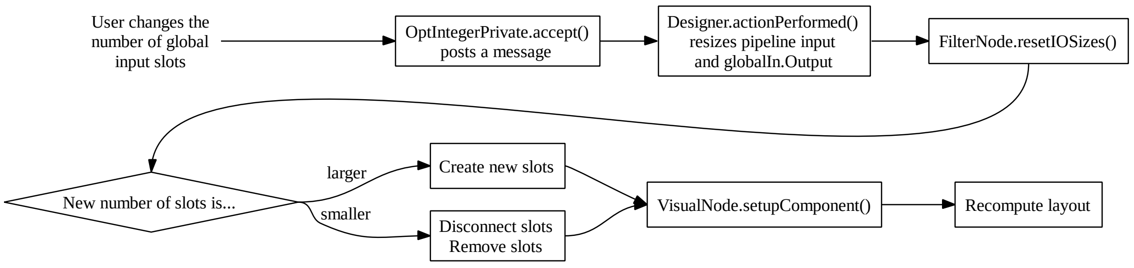

User clicks on "Options"
VisualNode posts a message
Designer receives the message
ShowOptions is called for the FilterNode represented by the VisualNode
All externalized options are checked

There’s a special case of option change that has to be considered separately from “regular” options: The global input node and the global output node don’t have a fixed number of slots. The number of output slots of the global input as well as the number of input slots of the global output can be changed in their respective option dialog. The `RWrapper` processing module uses the same mechanism to allow users to change the number of input and output slots for the R function (see 5.3).

Normally, the number of input and output slots of a `FilterNode` (and, in consequence, also of the `VisualNode`) is fixed, so changing it requires some extra work. The following list describes the process for changing the number of input slots for the processing pipeline, i.e. the number of output slots of the global input node. The process for changing the pipeline’s number of output slots is the same, only the words “output” and “input” have to be swapped.

1. The user changes the number of slots in the option window of the global input node and closes that window by clicking on “Accept”.
2. `FilterOptions` asks the option (`OptIntegerPrivate`) to validate its input and then calls its `accept()` method.
3. `OptIntegerPrivate` accepts the new value and posts a message which is received by `Designer`.
4. `Designer` changes the processing pipeline’s input slot number.
5. `Designer` changes the output slot number of the global input node and calls its `resetIOSizes()` method.

6. `FilterNode.resetIOSizes()` checks whether more slots are needed or some slots have to be removed.
    - If more slots are needed, the correct number of new `FilterSlot` objects is instantiated.
    - If less slots are needed, `resetIOSizes()` has to remove the surplus slots. Before they can be removed, existing connections at these slots have to be removed.
7. `resetIOSizes()` calls the attached `VisualNode`'s `setupComponent()` method to make the changed number of slots visible in the GUI.


User changes the number of global input slots
OptIntegerPrivate.accept() posts a message
Designer.actionPerformed() resizes pipeline input and globalIn.Output
FilterNode.resetIOSizes()
New number of slots is...
larger
smaller
Create new slots
Disconnect slots Remove slots
VisualNode.setupComponent()
Recompute layout

The menu choices

I will now discuss the different actions a user can perform by means of the menu bar.

- File → New: This creates a new processing pipeline (after asking whether any changes to the current pipeline should be saved, of course).
- File → Open: Here users can open an existing pipeline. This pipeline doesn't need to be located in the MPF plugin directory but can come from any location within the user's file system. If changes have been made to the current pipeline, the user can save them before continuing.
- File → Save: This action is only available if the current pipeline has been saved before or has been loaded from a file. New pipelines have to be saved by calling "Save as...". Every pipeline is saved to the MPF plugin directory regardless of whether it was loaded from there or from another location on disk.
- File → Save as: This is used to save a new pipeline for the first time or to save a changed pipeline while retaining the source file. Before the pipeline is written to disk, its name, description and version number have to be set and the user can choose which options to externalize. The dialog that is shown for that purpose is also available via "Options → Set filter options...".

  Before a pipeline is saved, its graph is validated by calling `ComplexFilter.validateGraph()`. Invalid pipelines can still be saved but users are informed that their pipeline will not work until the graph is made valid.

  Whenever a pipeline is saved, the `FilterClassList` singleton is informed of the fact that a new pipeline was created or that the pipeline may have changed. This triggers an update of the list of processing modules so that changes are visible in the Applicator window as soon as Designer is closed. These changes are also visible in the module selection drop-down list in Designer, so that after saving a pipeline it can be inserted into another pipeline.
- File → Close: Closes the Designer after prompting the user to save changes to the current pipeline.

- Options → Set filter options: This opens a dialog window where the name and description of the processing pipeline can be set as well as the pipeline version. Submodule options can be externalized from here, externalized options can be edited or removed.
- Options → Validate Filter: This calls `ComplexFilter.validateGraph()` to make sure that the current pipeline is a valid graph according to the rules described in section 4.3.1.

Graph layout

Apart from dealing with user input, the MPF Designer also needs to take care of displaying the pipeline graph. While the `VisualNode` class is responsible for drawing individual nodes and edges of the graph, it is the job of the `Designer` to calculate the graph layout, i.e. assign screen coordinates to every node in the graph.

The graph layout algorithm used here is not very sophisticated but it works quite well. It could be expanded to cover rare special cases but for most purposes the layout is good enough if not perfect.

The layout algorithm has to assign a pair of coordinates (x_k, y_k) to each `VisualNode` k. Consider three nodes k, ℓ and m where k has two input slots that are connected to the output of ℓ and m, respectively. To make the logical ordering of the nodes ("k is executed *after* ℓ and m") obvious, we want

$$x_k > \max(x_\ell, x_m).$$

The second coordinate of k, y_k, should reflect the fact that k is taking input from the two other nodes. A logical choice would be

$$y_k = \frac{y_\ell + y_m}{2}.$$

Of course, to calculate the coordinates of k, the coordinates of all nodes giving input to k must already be known. This can be achieved by once more returning to the node indices: If we assign coordinates in the order of the node index that `buildGraph()` has assigned to the corresponding `FilterNode`s, all coordinates we need for our calculation are properly defined in time.

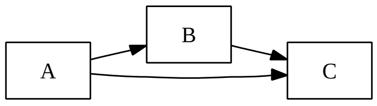
The algorithm has to consider one problematic case: If a node m has more than one output slot and those are connected to the input slots of nodes $k_1, \ldots, k_n$, the simple calculations above will yield the same coordinates for all k_i. Thus we have to check whether two `VisualNode`s overlap. If an overlap is detected, one of the nodes has to be moved. Again we can use the node index and for two colliding nodes k_i and k_j decide to move k_j away in direction of the y coordinate if $index(k_i) < index(k_j)$.

These considerations are the basis for the following pseudo-code.

```
nodeindex = 0

while (nodeindex < total number of nodes) do

    Let k be the node with k.index = nodeindex

    set x_k=0, y_k=0, count=0

    for each incoming slot i do
        let l be the node connected to i
        x_k = max ( x_k , x_l + width(l) )
        y_k = y_k + y_l
        count++
    od

    x_k = x_k + spacer
    y_k = min ( (y_k / count) - (height(k)/2) , spacer )

    for each node l with l.index < nodeindex do
        if l and k collide
            set y_k = y_l + height(l) + spacer
    od

    nodeindex++

od
```

Clearly, the runtime of this algorithm is quadratic in the number of nodes which is acceptable as pipelines will not exceed a reasonable number of nodes. Pipelines with more than a few dozen nodes are hard to imagine and should speed ever become a problem, such huge pipelines could be split into several smaller pipeline parts reducing the "master" pipeline to a few nodes, each one representing one of the smaller pipelines.

Layout problems



Obviously, this simple graph layout algorithm is not perfect. If the graph is invalid, the algorithm can not work correctly because some of the nodes have unconnected input slots. But there is also one particular case where the layout of a valid graph is not as good as it could be. Consider the following pipeline: A node A connects to a node B and a node C while B also connects to C. The algorithm calculates the coordinates (x, y) so that

$$x_c > x_b > x_a \quad \text{and} \quad y_c = \frac{y_b + y_a}{2}.$$

We know that $y_b = y_a$, thus $y_c = y_a = y_b$ and all nodes are drawn in one line. The problem now is that graph edges are drawn as straight lines which results in the edge from A to C cutting across the node B. In some cases this can make it hard to distinguish graph edges.

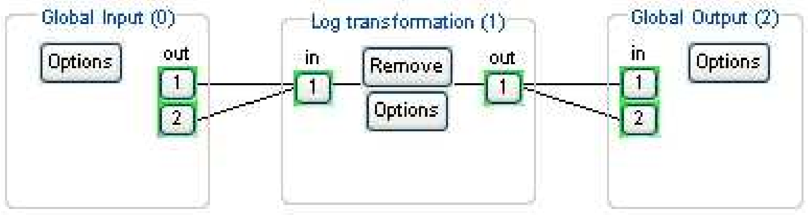


Figure 4.24: A graph where the algorithm fails to produce a perfect layout.

4.4.2 `VisualNode`

The `VisualNode` class creates the GUI elements necessary to represent a `FilterNode` within the Designer window. One can think of the `VisualNode` class as the *presentation logic* corresponding to the *business logic* embodied in the `FilterNode`.

This class is derived from `javax.swing.JPanel` and can as such be incorporated into any existing Swing GUI. The elements it creates are: A node border with the node name and its node index, a button for node removal, a button that opens the corresponding `FilterBase`'s option dialog window and buttons for all input and output slots that can be used to create connections between the nodes of the graph.

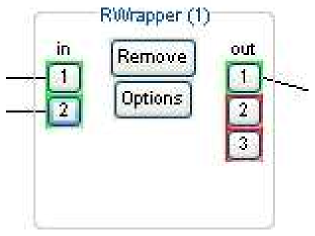


Figure 4.25: A `VisualNode` *GUI element representing an instance of "RWrapper" that has a node index value of* 1 *and offers two input and three output slots. Connected slots are indicated in green, those lacking a connection in red.*

Apart from creating and drawing the graph node, `VisualNode` also takes care of drawing graph connections. Every `VisualNode` instance is responsible for drawing the *outgoing* connections attached to its corresponding `FilterNode` object.

4.4.3 `DesignerPanel`

`DesignerPanel` takes the role of a `javax.swing.JPanel` representing the main area of the `Designer` window where the graph is displayed. While `VisualNode`s, as descendants of `javax.swing.JPanel`, could also be displayed by a normal `JPanel` instance, `DesignerPanel` adds some useful functions.

When the `paintComponent` method of `DesignerPanel` is called, it asks each of the `VisualNodes` it contains to draw the outgoing connections of the `FilterNode` it represents. For this job, the `VisualNode` needs additional position information from the panel as well as a reference to the panel itself because the connection lines extend beyond the boundaries of the `VisualNode` object and are drawn directly onto the panel.

The class also performs a function that very much increases usability of the graph display: When a user changes the graph by creating or removing connections, the graph has to be rebuilt to assign new node indices. This also entails recomputing the graph layout so that it reflects these changes. Even small changes can have extensive effects on the graph layout.

If the change from the old layout to the new layout were done in a fraction of a second, users would always have to spend time to get accustomed to the new layout, e.g. searching for a node that was at a particular position just a moment ago and has now moved somewhere else. Searching for nodes after every change of the graph takes a lot of time and makes working with the Designer tiresome. A solution to this problem has been implemented in the `DesignerPanel` class:

When `Designer.computeLayout()` recomputes the coordinates of all nodes in the graph, the nodes are not moved to the new coordinates at once. Instead, they are moved to their new location slowly enough for the user to follow nodes of interest. Thus the change from one layout to another is not seen as an abrupt interruption and working on the graph gets a more continuous feeling.

The movement is implemented as a separate thread that is created after `computeLayout()` is finished. The thread is destroyed after the nodes have reached their new positions. Nodes are moved at 25fps to create the illusion of smooth movement. The movement process is completed in 30 steps, i.e. 1.2 seconds. To make the effect more appealing, the nodes don't move at constant speed. Their speed follows the sigmoidal function which is incorporated into the calculation of intermediate node locations as follows:

$$\text{Position} = f \cdot \text{Target\_Position} + (1 - f) \cdot \text{Old\_Position}$$

where

$$f = \frac{1}{1 + \exp(5 - 10 \cdot i)} \qquad \text{for } i \in [0, 1]$$

The function f was chosen empirically.

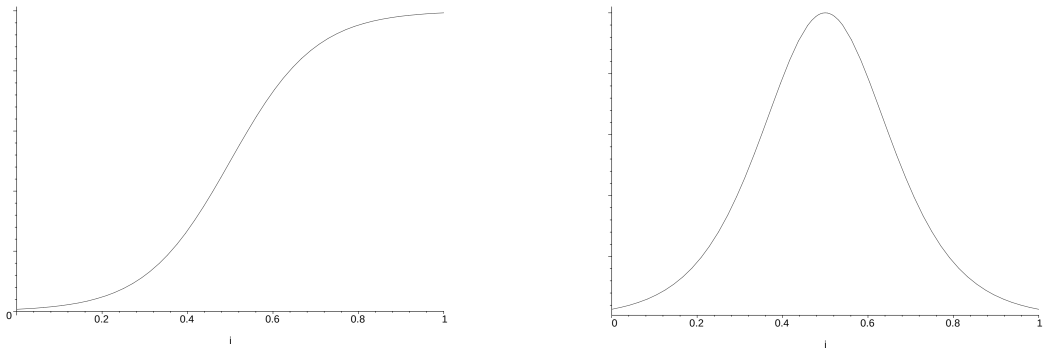
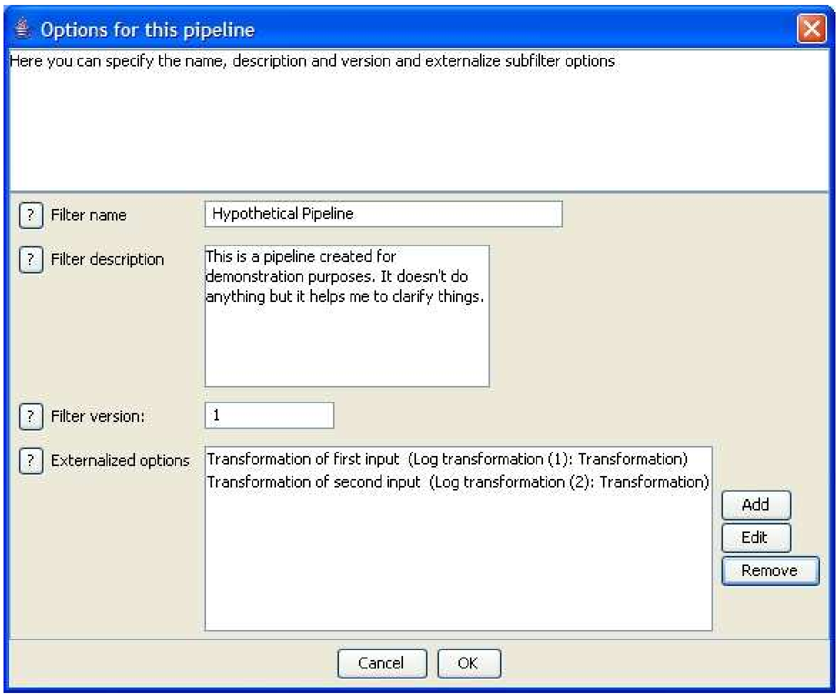


Figure 4.26: The function f influences the location of a node (left) as well as the speed of its movement (first differential, right).

4.4.4 `OptionEditor`

A processing pipeline has several options, some of them derived from the `FilterBase` class, i.e. the *name* of the pipeline, a *description* of what it does and how its results can be influenced and a *version number*. In addition, processing pipelines can externalize options of their submodules to make them available to users of the pipeline.


Options for this pipeline

Here you can specify the name, description and version and externalize subfilter options

Filter name: Hypothetical Pipeline

Filter description: This is a pipeline created for demonstration purposes. It doesn't do anything but it helps me to clarify things.

Filter version: 1

Externalized options:
Transformation of first input (Log transformation (1): Transformation)
Transformation of second input (Log transformation (2): Transformation)

Figure 4.27: The option dialog window for a processing pipeline.

To set these options, the `OptionEditor` class provides a dialog that contains GUI elements to manipulate these options as well as information about every option. All options in the dialog window are objects of types descending from `OptBase`: The name is represented by an instance of `OptString`, the description by an instance of `OptMultiline` and the version number as an instance of `OptInteger`. For the list of externalized options, `OptionEditor` defines an inner class `OptExternalizedList` that is also derived from `OptBase`.
These option objects are added to a new instance of `FilterOptions` that takes care of displaying the GUI dialog and handling user input. This shows how versatile the `FilterOptions`/`OptBase` framework is, allowing even highly complex option objects like `OptExternalizedList`.

4.5 Static classes

Three classes that are used throughout the MPF don't belong to any one part of the plugin and don't fit into the class hierarchy. These are the static classes `Constants`, `ExceptionHandler` and `ModalFrameUtil`.

4.5.1 `Constants`

This class contains some constants that are used in the MPF, such as window titles, the relative path to processing modules as well as the file name extension for graph description files.

4.5.2 `ExceptionHandler`

`ExceptionHandler` is a small class that takes an object of `Throwable` or a derived type and displays a message window with the object's message. It is used in the MPF whereeever a problem has to be communicated to the user.

General notes on Exceptions

The MPF makes a lot of use of exceptions to communicate problems up the calling stack hierarchy, i.e. from called functions to calling functions, because exceptions offer some benefits over return values:

- Using exceptions enhances the readability of application code. By separating processing code that works most of the time from special error handling code that is expected to be called only very rarely, understanding the "everyday" processing code is much easier.
- Exceptions are transparent to the application code. This means that when a function is called that may throw an exception, the caller must not be aware of this fact. It's the choice of the programmer whether the problem needs to be addressed at the current level or whether it's preferable to handle the exception at a higher layer.
  Using return values to indicate exceptions is fundamentally different in that respect. Special values must be defined that indicate a problem and the calling function must be aware of the possible return values and check whether one of the special values was returned. If that is the case, the calling function must deal with the problem right away or delegate that task to its own caller by returning another special value. That way, a problem has to be reported manually through the calling stack until finally one function is ready to deal with it. Apart from the addition in effort and code complexity, this also creates the problem of defining special return values. For some return types it can be hard to find a special value, e.g. consider a function returning an `int` value. All valid integers can be return values of the function, so there's no special unused value that could be employed to indicate a problem.

Within the MPF, exceptions are often thrown accross several levels in the call hierarchy before a decision can be made about how to deal with the problem at hand. For example, `ComplexFilter` throws an exeption if it's loading a pipeline that doesn't validate. If we have a pipeline A that contains, as a submodule, a pipeline B and the filter graph for B is invalid, `Designer` shows a message describing the problem but lets the user work on A while `Applicator` denies using the broken pipeline as a processing module.

4.5.3 `ModalFrameUtil`

Java Swing offers two kinds of GUI windows that come with different capabilities and limitations:

- Windows that descend from `javax.swing.JDialog` are *dialog windows*. Their window title bar has no maximize button. Dialog windows can be *modal* windows, which means that a parent window opening a modal dialog will be disabled as long as the dialog window is open and reenabled as soon as the dialog is closed. In addition to not accepting user input, the parent window thread waits for the dialog to close because calling the `setVisible()` function of the `JDialog` doesn't return as long as the dialog is visible.
- Windows descending from `javax.swing.JFrame` are normal windows. Their title bar has maximize and minimize buttons. Java doesn't support showing normal windows as modal windows.

There is no easy way to make create a GUI window that combines the characteristics of those two classes even though it can be useful to have modal windows that can be resized and maximized by the user. In the case of the MPF I wanted to have a resizable `Designer` window which is opened as a modal dialog thereby blocking the `Applicator` window.

To get "the best of both worlds", I use the `ModalFrameUtil` class. It's basically an adaption of the blocking mechanism used in the `JDialog` class so that this mechanism can be applied to instances of `JFrame`. The class was written by Vikram Mohan and published online [Moh06]. I made some minor modifications to his code. To the best of my knowledge, Mr. Mohan placed no restrictions on the use of his code.

4.6 Modifications to other Mayday classes

`Probes` and `ProbeLists` that are created by the MPF are annotated with MIOs describing the applied module and its options. When a `Probe` is cloned, its MIO complement has to be cloned as well (see section 4.1.7). This required additional functions in the Mayday core.

In order to make the `RWrapper` module work, I had to introduce small changes into some of the `RPlugin`'s classes, most of them were necessary to make the `RPlugin` run without any user intervention.

The following sections will briefly present my changes to classes outside the MPF.

4.6.1 Modifications to `MIOGroup`

Originally, `MIOGroups` were meant to be created, filled with MIOs and then "finished". After finishing a `MIOGroup`, no further MIOs can be added to that group. The MPF, however, needs to be able to add new MIOs to finished groups, e.g. when a Probe has to be cloned along with its MIO complement. To allow this, I created the `MIOGroup.addAfterFinishing()` method. This method tricks the `MIOGroup` into believing that it is in fact *not* finished. The `MIOGroup` then accepts the addition of a new MIO. Finally I have to perform the "finishing" on the one MIO just added.

Secondly, `MIOGroups` didn't allow removing MIOs once placed into the group. As I have shown in section 4.1.7, removal of MIOs is essential when `Probes` and `ProbeLists` created by `MaydayData-Objects` are to be reclaimed by the garbage collector. The `MIOGroup.remove` method was already in the source code but it was commented out, so I only had to remove the comments.

4.6.2 Modifications to the R Interpreter plugin

Modifications to `RPlugin`

Several changes to `RPlugin.runInternal()` were necessary to make the previously untested function work:

1. A `StringBuffer` was not initialized which lead to a `NullPointerException` whenever the function was called.
2. The function tried to return the R function's warning messages by assigning them to a `String` parameter. This didn't work. I changed the `String` to a `StringBuffer`
3. Also, the function tried to return the new `MasterTable` by assigning it to a `MasterTable` parameter. This didn't work. I changed the function so that the new `MasterTable` is returned inside the `RSettings` parameter object.

4. Deletion of temporary input and output files was omitted even though the supplied `RSettings` object asked for it. I added the necessary code to respond to the `deleteInputFiles` and `deleteOutputFiles` flags.

Modifications to `RJob`

The `RJob.run()` method was changed so that jobs created by `RPlugin.runInternal()` don't wait for the user to click on an "OK" button after R has finished. Now, instead of showing the dialog and waiting for input, the function simply waits for the task to finish and then exits.

Modifications to `RProcessStateMonitor`

The `RProcessStateMonitor.run()` method failed to check a temporary variable for `null` which sometimes resulted in an exception.

Modifications to `RResultParser`

I added a small code block in the `RResultParser.parse()` function to make sure that the `RPlugin` only adds a new `DataSetView` to Mayday's `DataSetManagerView` when the input `DataSet` was also represented by a `DataSetView` instance there. This prevents the new `DataSet` from appearing in Mayday and then disappearing again when the `RWrapper` integrates it into `MaydayDataObjects`.

Modifications to `RSettings`

Two changes were necessary to the `RSettings` class: I added a new boolean member variable `silentRunning` which indicates whether the plugin was called by Mayday's `PluginManager` (`silentRunning=false`) or by another plugin. Plugins whishing to suppress all user interaction within the `RPlugin` have to pass an instance of `RSettings` to `RPlugin.runInternal()` where the `silentRunning` member is set to `true`. I also added some code to the `RSettings.createInitializedInstance` function to make sure that the working directory is initialized correctly.

4.7 Class diagram

The full diagram of all classes ends this detailed discussion of the MPF core. The next section will discuss the implementation of MPF modules with the help of three examples.

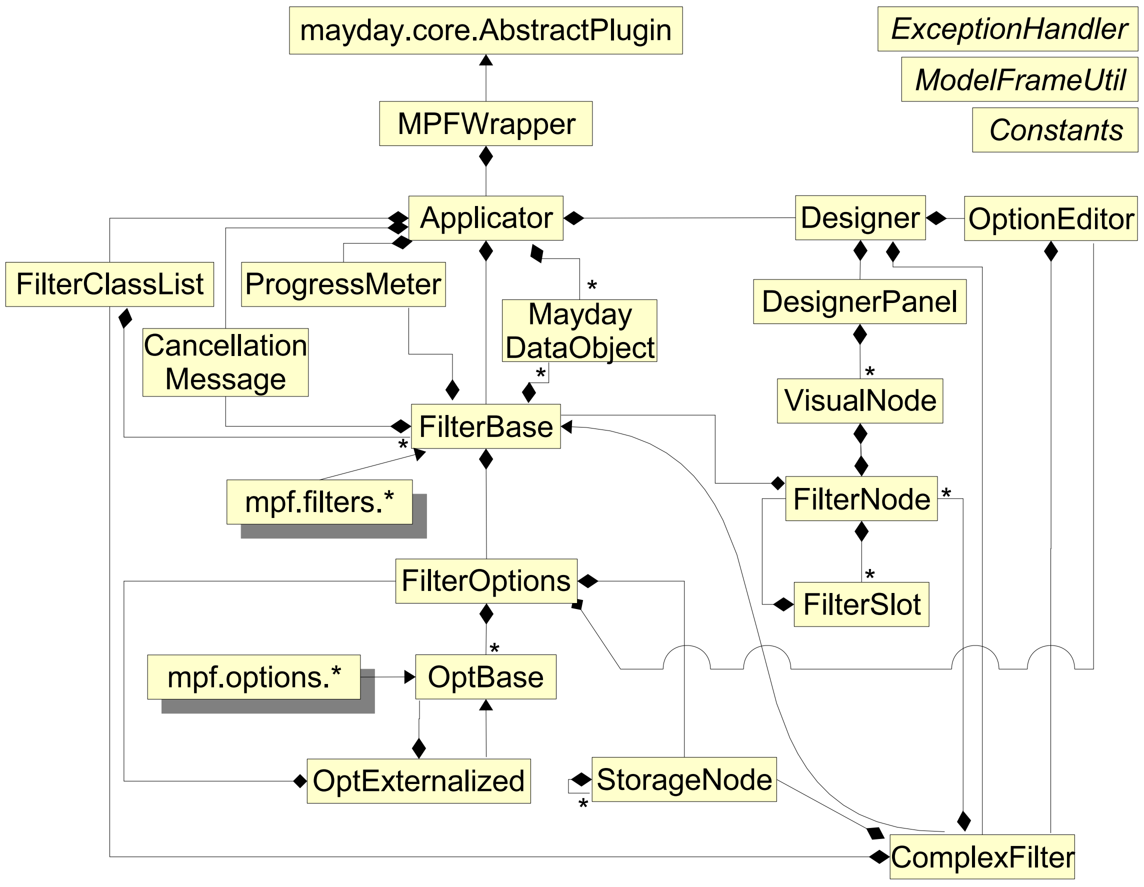


Figure 4.28: Overview over all non-internal classes of the MPF.

5. Implementing processing modules

This chapter will show how processing modules can be implemented. I will present three modules included in the MPF base installation as examples. These examples range from very simple to relatively complicated modules.

5.1 Simple: Expression value corridor

The expression value corridor filter removes all probes whose expression values don't fall into a specified interval. It's obvious that we need options for the upper and lower interval border. In addition, I also added two boolean options, one to answer the question whether missing values are treated as falling inside the interval and another one to let users run the filter in "inverted mode", i.e. remove all probes that have no expression values outside of the corridor.

The first thing to do is to add the correct package declaration and import classes we need. Every module inherits from `FilterBase`, so I need to import that class. I also need to import the `OptBase` descendant classes for my option values as well as `mayday.core.Probe` for working on probes.

```
package mayday.mpf.filters;

import mayday.mpf.FilterBase;
import mayday.mpf.options.OptBoolean;
import mayday.mpf.options.OptDouble;
import mayday.core.Probe;
```

The next step is the class declaration and the definition of the option objects I just described. Note how the `OptDouble` and `OptBoolean` instances are initialized with the option name, its description and a default value.

```
public class ExpValCorridor extends FilterBase {

    OptDouble minVal = new OptDouble(
            "Lower bound",
            "The lowest value still in the corridor",
            0.0);

    OptDouble maxVal = new OptDouble(
            "Upper bound",
            "The highest value still in the corridor",
            100.0);

    OptBoolean invert = new OptBoolean(
            "Inverted mode",
            "Select this option to discard probes that fall inside "
            + "the corridor instead of keeping them",
            false);

    OptBoolean nullInside = new OptBoolean(
            "Consider missing values as inside",
            "Select this option to treat missing values as falling "
            + "into the corridor as opposed to lying outside of it.",
            false);
```

Every processing module must have exactly one constructor that takes no arguments. The first job of the constructor is to call the `FilterBase` superconstructor with the correct number of input and output slots. The constructor has to set the module name and its description. It's also important to add the predefined option objects to the module's option list. Here, no additional work is necessary, so the constructor is complete.

```
    public ExpValCorridor() {
        super(1,1);
        Name="Expression Value Corridor";
        Description="Filters probes based on whether their expression "
                + " values fall within a certain corridor.";
        Options.add(minVal);
        Options.add(maxVal);
        Options.add(nullInside);
        Options.add(invert);
    }
```

The second crucial function for each processing module is its `execute()` function. It must take no arguments and have `void` as its return type. For most modules with one input and one output slot, this function starts with the statement `OutputData[0]=InputData[0];`. This statement transfers the input `MaydayDataObject` (MDO) instance to the output slot of the module. The MPF makes sure that subsequent modifications to the `OutputData` object are done without affecting other MDOs or Mayday's data structures.

Now the module needs to inspect each probe to see whether to keep it or not. This is done in the for-loop iterating over all `Probe`s in the `OutputData` MDO. It is strongly recommended to use the Java 1.5 way of iterating over the MDO because this makes sure that the iteration covers all probes present in the MDO at the start of the iteration. Adding or removing probes does not break the iteration.

Normally, removing an element from a container during iteration breaks the iterator as does adding one. If indices are used instead of iterators, adding and removing an element has to be done with great care to make sure that no element is skipped and the iteration doesn't go beyond the limits of the container. In the case of a `ProbeList` the problem is even greater: `Probe`s are stored in a hashmap, so the programmer can never know whether a newly added probe is inserted before the current iteration index or after it. The iteration method shown here does not have these problems.

In the loop, I simply check whether a probe fulfills the criteria defined in the options and remove it if it doesn't.

```
    public void execute() {
        OutputData[0]=InputData[0];
        for (Probe pb : OutputData[0]) {
            boolean keep = checkCriteria(pb);
            if (invert.Value) keep=!keep;
            if (!keep) OutputData[0].remove(pb);
        }
    }
}
```

The `checkCriteria` function used in the `execute` method iterates over all expression values of the probe and checks whether they fall into the predefined corridor, taking into account the user's decision on how to treat missing values. Note how the value of an option object can simply be accessed by its `Value` member. The `Value` of an `OptDouble` object is of type `Double`, the `Value` of an `OptBoolean` is of type `Boolean` and so forth.

```
    private boolean checkCriteria(Probe pb) {
        boolean isInside = true;
        for (int i=0; i!=pb.getNumberOfExperiments() && isInside; ++i)
        {
            Double d = pb.getValue(i);
            if (d==null)
                isInside &= nullInside.Value;
            else
                isInside &= (d<=maxVal.Value) && (d>=minVal.Value);
        }
        return isInside;
    }
```

5.2 Not so simple: MIO Presence Filter

The MIO Presence Filter module is a more complicated example. It needs to scan the input data for MIO groups it can filter on (this can only be done during execution of the module when actual input data is present). The module also needs to show a slightly modified MIO group selection dialog. The package and import declarations are as follows.

```
package mayday.mpf.filters;

import java.util.ArrayList;
import javax.swing.JCheckBox;
import mayday.mpf.options.OptDropDown;
import mayday.core.DataSet;
import mayday.core.Probe;
import mayday.core.gui.AbstractStandardDialogComponent;
import mayday.core.gui.StandardDialog;
import mayday.core.mi.MIManager;
import mayday.core.mi.MIOGroup;
import mayday.core.mi.MIOType;
import mayday.core.mi.MIStructure;
import mayday.core.mi.gui.MIOGroupSelectionComponent;
import mayday.mpf.FilterBase;
```

This class makes use of an `OptDropDown` option object which is initialized with a name, a description, a list of values that the user can select from and the index of the value that is selected by default. Additional members are a list of `MIOGroup`s that have to be present in a probe for it to pass the filter and a boolean value indicating whether the MIO group(s) selected during the first run of the module should be used for all further runs in batch mode. No special work is required in the constructor.

```
public class MIOPresent extends FilterBase {

    private ArrayList<MIOGroup> selectedGroups = null;
    private boolean reuseGroup = false;

    private OptDropDown matchmode = new OptDropDown(
        "Keep probes that",
        "Select whether to keep matching or non-matching probes.",

        new String[]{
            "contain the selected MIO group(s)",
            "don't contain the selected MIO group(s)"
        },
        0);

    public MIOPresent() {
        super(1,1);
        Name = "MIO Presence Filter";
        Description = "Removes probes based on whether a certain meta "
                    + "information object group is attached to them.\n"
                    + "Requires user interaction during execution "
                    + "(selection of a MIO Group).";
        Options.add(matchmode);
    }
```

The module's execute method has to do several things: First it has to find out which MIO groups the user wants to filter against. In batch mode, the selection dialog is not shown again if the user decided to use his first selection for all following jobs. For logging purposes, `rpCounter` counts how many probes have been removed by the module.

The main loop iterates over all probes to check which ones to keep. For each MIO group selected by the user, the function checks whether the group contains an annotation for the given probe. If that isn't the case, the probe is removed and the counter is incremented. Finally, the module uses its attached `ProgressMeter` instance to report the number of removed probes. It does not, however, show a progress indicator while it is working. In every iteration, the `isCancelled()` function is called to check whether the user asked to cancel processing. If this is the case, the `execute()` method simply returns, and MDO takes care of memory cleanup.

```
    public void execute() throws Exception {
        if (selectedGroups==null || !reuseGroup) {
            getMIOGroups();
        }

        int rpCounter = 0;

        OutputData[0]=InputData[0];

        if (selectedGroups!=null) {

            for (Probe pb : OutputData[0]) {
                if isCancelled() return;
                boolean matching = true;
                for (MIOGroup mg : selectedGroups) {
                    MIStructure dm = mg.getMIO(pb);
                    matching &= (dm!=null);
                }
                if (matchmode.Value==1) matching=!matching;
                if (!matching) {
                    OutputData[0].remove(pb);
                    ++rpCounter;
                }
            }
```

```
                ProgressMeter.writeLogLine
                  (rpCounter + " probes removed (MIOPresence).");
            }
        }
```

A lot of this module's work is done in helper functions. One of them is `getMIOGroups`. This function scans the input probe list for MIO groups and then asks the user which of these groups should be used for filtering. At first, the function has to find all MIO groups. This is done by inspecting the MIO container attached to each input probe. Every MIO group is added exactly once to the list of groups found.

```
        private void getMIOGroups() {

            // collect all MIO groups in input data
            ArrayList<MIOGroup> MIOcollector = new ArrayList<MIOGroup>();
            for (Probe pb : InputData[0]) {
                ArrayList<MIOType> list = pb.getMIContainer().getAllMIOs();
                for (MIOType mis : list) {
                    MIOGroup mg = mis.getGroup();
                    if (!MIOContainerContains(MIOcollector, mg))
                        MIOcollector.add(mg);
                }
            }
```

The next step is to let the user select a subset of the MIO groups found. I use a modified instance of `MIOGroupSelectionDialog` for this purpose. This dialog expects its input to be a `MIManager`, so I have to create one and fill it with the MIO groups. The dialog is invoked and the user's selection is remembered. If no MIO was found in the input data, the module stops with an exception. The MPF then takes care of memory cleanup.

```
            MIManager allMIOs = new MIManager((DataSet)null);
            for (MIOGroup mig : MIOcollector) allMIOs.add(mig);

            // show selection window
            MIOGroupSelectionDialogX seldia
                = new MIOGroupSelectionDialogX(allMIOs, MIOType.class);

            if (MIOcollector.size()>0)
            {
                seldia.setVisible(true);
                selectedGroups=seldia.getSelection();
                reuseGroup=seldia.getUseForAll();
            } else
                throw new RuntimeException("No MIO groups found "
                                          + "to allow selection.");
        }
```

While collecting the MIO groups, the `MIOContainerContains` method is used to check whether a MIO group has already been found before. Here I can't use `ArrayList.contains()` because this function uses the equals() function of the `MIOGroup` class which doesn't compare pointers but MIO group indices. I found that they aren't necessarily unique.

```
    private boolean MIOContainerContains(
        ArrayList<MIOGroup> MIOCollector,
        MIOGroup mg)
    {
        for (MIOGroup mg2 : MIOCollector)
            if (mg2==mg) return true;
        return false;
    }
```

To complete the MIO Presence Filter, the modified `MIOGroupSelectionDialog` has to be included as an inner class. This class is mostly equal to the original class but it includes a `JCheckBox` component so that users can select if they want to use the same selection for all subsequent jobs in batch mode.

5.3 Awfully complicated: R Wrapper

The "Wrapper for R interpreter functions" module (`RWrapper`) is by far the most complicated module included with the MPF. This is due to the problems I found calling the R interpreter [Zsc04]. `RInterpreter` doesn't conform to the `mayday.core.Pluggable` interface and relies on direct manipulation of Mayday's data structures instead. For instance, it allows R functions to return new data sets while the `Pluggable` interface only allows probe lists in the input `DataSet`. Also, R functions can modify input data in any way they like because the `RInterpreter` gives them access to the main `MasterTable`.

Including R [R D05] functions in MPF pipelines is very desirable because R scripts offer a sort of middle road between the limited possibilities of building processing pipelines from modules in the Designer GUI and writing new modules as Java classes. R is extremely versatile when it comes to statistics. But R functions can also be used to circumvent a big problem of the MPF: There is no way to include visualization plugins into MPF pipelines because no common API exists for automating visualization (e.g. data selection, zoom, image export, etc.). Many visualisation tasks can be performed in R. One example is the missing value graph presented in figure 6.1.

To include R functions in MPF pipelines, the `RWrapper` has to "tame" the `RInterpreter`. This means that some R functions will not work with the `RWrapper`, particularly those that return new data sets (though not all of them, as will be shown later). As a consequence of the `RWrapper`'s complexity, I will not include the source code here. Instead I will discuss the ways of the `RInterpreter` and how the `RWrapper` deals with them to make as many R functions as possible compatible with the MPF.

- *R functions declare neither the number of input probe lists they expect nor the number of output probe lists they create.*
  To include R functions in processing pipelines, the MPF has to know how many input and output slots to allocate for the corresponding processing module. The XML declaration supplied with each R function doesn't contain that information and there is no way to automatically detect it. Thus the user needs to supply that information. This, in turn, leads to several problems:

  1. *How does the user know these values?*
     Obviously, users have to know the values to set them correctly. Even though using R functions may be considered a "power-user" task, the `RWrapper` should be as tolerant as possible with respect to incorrect slot numbers.
  2. *Slot numbers can't be changed in Applicator*
     In Designer, users can set the number of input and output slots for `RWrapper` `FilterNodes` in the node's option dialog. They can then connect all slots, save and run the pipeline.

But what about Applicator? Here, the number of slots has to be known to complete the first step of the Applicator dialog before the `RWrapper` option window is shown in the third step. This situation could only be changed by a complete rewrite of the Applicator module.

The problems were addressed as follows: By default, an instance of `RWrapper` has one input and one output slot. This should allow most R functions to be used directly from the Applicator without the need for changing slot numbers. For those functions that require a different number of slots, a processing pipeline has to be created, containing only the `RWrapper` with the desired number of slots.

The `RWrapper` needs two `OptInteger` option objects, one to set the number of input slots and one for the output slots. These options must be present when the option dialog is shown inside the Designer but they may not be changed when it is shown from the Applicator. For this reason, a subclass of `OptInteger` was created. That subclass, `RWrapper.OptIntegerFancy`, knows whether it is being displayed inside Applicator or Designer by checking the `RWrapper`'s `CancellationMessage` member. Applicator assigns its own `CancellationMessage` instance to that variable, Designer doesn't.

The question of tolerance with respect to a wrong number of input slots is outside the MPF's scope: R functions will have to check whether or not they have all the input they need. Concerning the output slots, the `RWrapper` simply checks whether the correct number of probe lists was returned. If probe lists are missing, empty ones are created in their place. If too many probe lists are returned, the surplus lists are ignored. In both cases, a warning message is displayed.

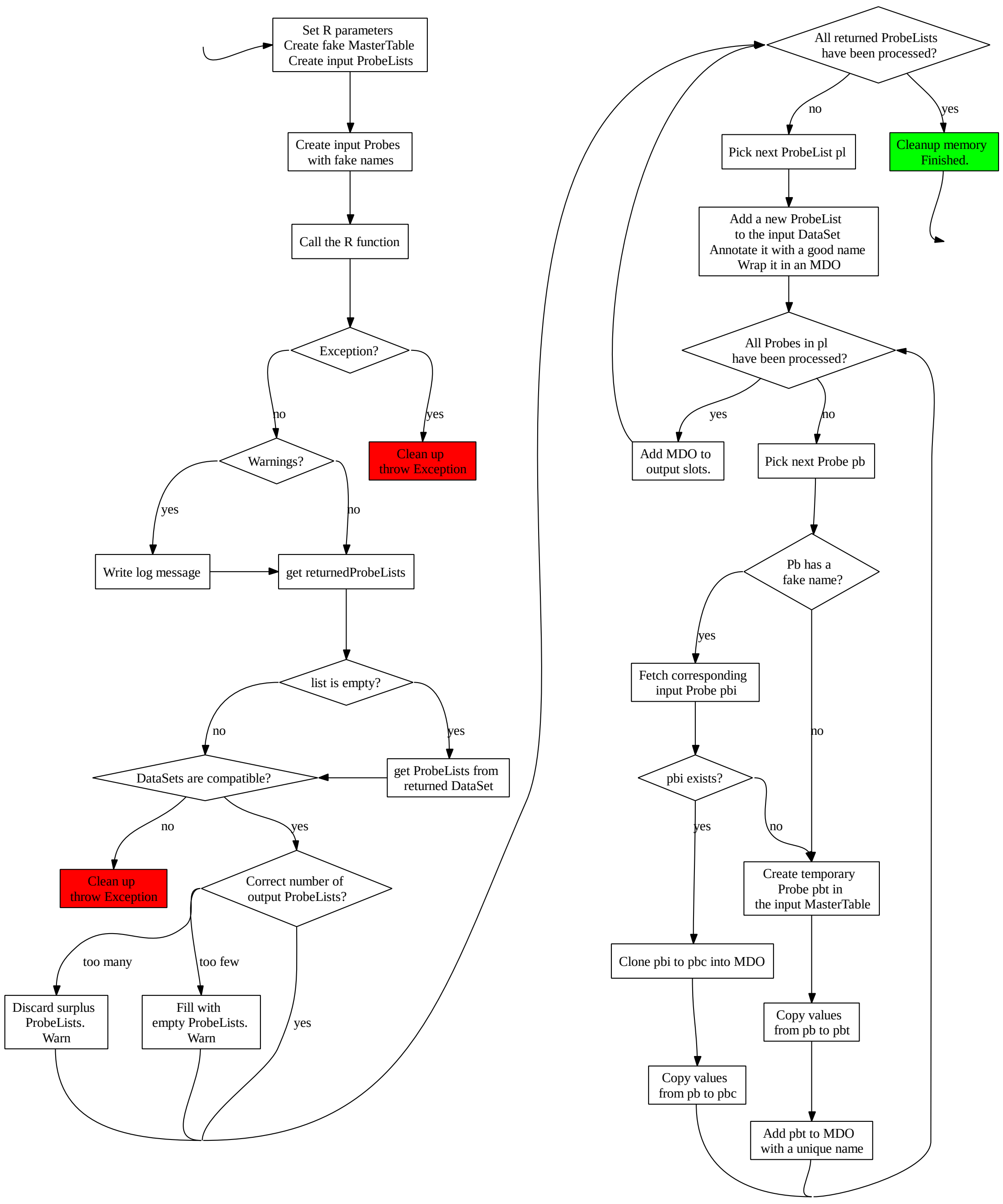
- *R functions can return new* `DataSet`*s.*
  This is the result of a decision made during the design of the `RPlugin` and greatly extends the field of application for R functions. But as a consequence it is much harder to integrate the output of an R function into MPF pipelines. The `Pluggable` interface allows plugins to return `ProbeLists` but R functions are allowed to create entire `DataSets`. These are "returned" by mounting them into Mayday's data structures behind the program's back. Some changes to the `RPlugin` were necessary to suppress this behaviour when R functions are called by the `RWrapper`.
  The integration of probe lists returned inside a new data set is problematic. The `RWrapper` has to make sure that the new data set has the same number of experiments as the input data set. If that isn't the case, the returned data cannot be incorporated into the input `DataSet` and that particular R function can't be used within the MPF. An example would be the "Transpose" function which transforms an $m \times n$ matrix into an $n \times m$ matrix.

- *R functions don't know about* `MaydayDataObject`*.*
  If an R function decides to manipulate its input `MasterTable`, these changes affect the input data of the MPF job. One of the requirements of the MPF is that processing does not affect input data. This means that the data given to R functions must be separate from the original input data given to the `RWrapper`. Instead of the input `MasterTable`, a "fake" one has to be created. There's also a second reason to do this: Some R functions don't limit their processing to the probe lists they have been given but rather manipulate all probes within the associated `MasterTable`. This behaviour is not desirable for processing modules in MPF pipelines. Thus the `RWrapper` restricts the fake `MasterTable` to those probes that are contained in its input probe lists.
  After the R function finishes working on the data, the results are `Probes` in `ProbeLists` (possibly in a new `MasterTable`). Integration into MPF pipelines requires the results to be `Probes` in `MaydayDataObjects`, so the `RWrapper` has to convert all output back to MDOs (as well as convert all input MDOs to normal probe lists before calling the R function). Of course all these additional objects introduce the need for a special memory cleanup function.

- *Some R functions may drop associated MIOs*
  Originally, the `RPlugin` was not designed to process meta information. When MIOs were introduced into Mayday, the `RPlugin` was extended so that R functions could access and create MIOs but there are some R functions that discard attached MIO information. Keeping annotations is crucial for most data analysis efforts (for an example, see 6.1), so the `RWrapper` has to make sure that MIOs associated with input `Probe`s are still present in the processed output `Probe`s. To know which MIOs to attach to output probes, the `RWrapper` must know whether an output probe was created by the R function or if it is, in fact, the result of processing an input probe (regardless of whether the input probe was modified or not).
  The connection between output and input probes is made via the probe name: All probes are "tagged" by adding a unique identifier to the probe name before they are sent into the R function. When the function returns, all output probes are examined. Those that still possess the tag are considered to be derived from the respective input probes and get a clone of their ancestors' MIO complement; the tag is removed from the probe name. Probes without tag are considered to be new probes without MIOs.

- *R functions have differing parameter lists.*
  Every R function comes with an XML file describing the parameters that influence its work. These parameters need to be accessible when the function is used in an `RWrapper` instance. This means that all options must be represented by `OptBase` descendants to integrate them into the `FilterOptions` list of the `RWrapper` object and that the values of all options must be converted back into the `RPlugin`'s internal representation before calling the R function. The `RWrapper` must change its option list whenever the user selects another R function. In addition, creators of processing pipelines should be able to externalize the R function's parameters. Finally, the `OptDropDown` instance used to allow the selection of the R function must never be externalized. The `FilterOptions/OptBase` tandem provides all methods necessary to accomplish this.

Figure 5.1 shows what happens during the `RWrapper`'s `execute()` method. Note how there are only two possible reasons for the `RWrapper` to fail: Either the R function returns a new `DataSet` incompatible with the input `DataSet` and there's nothing the `RWrapper` can do to integrate the results into the current `DataSet`, or the R function itself fails (usually due to a problem with the input data, e.g. the function can't deal with missing values).



Set R parameters
Create fake MasterTable
Create input ProbeLists
Create input Probes with fake names
Call the R function
Exception?
no
yes
Clean up throw Exception
Warnings?
yes
no
Write log message
get returnedProbeLists
list is empty?
no
yes
get ProbeLists from returned DataSet
DataSets are compatible?
no
yes
Clean up throw Exception
Correct number of output ProbeLists?
too many
too few
yes
Discard surplus ProbeLists. Warn
Fill with empty ProbeLists. Warn
All returned ProbeLists have been processed?
no
yes
Pick next ProbeList pl
Cleanup memory Finished.
Add a new ProbeList to the input DataSet Annotate it with a good name Wrap it in an MDO
All Probes in pl have been processed?
yes
no
Add MDO to output slots.
Pick next Probe pb
Pb has a fake name?
yes
no
Fetch corresponding input Probe pbi
pbi exists?
yes
no
Create temporary Probe pbt in the input MasterTable
Clone pbi to pbc into MDO
Copy values from pb to pbt
Copy values from pb to pbc
Add pbt to MDO with a unique name

Figure 5.1: Flow diagram of the RWrapper's `execute()` *method.*

6. Using the MPF

This chapter will use an example to show how the MPF can be used to process data. I use the Spellman cdc15 data set [SSZ+98] as input. This data set was obtained from a time series experiment. More than 6000 genes were tested on microarrays that were manufactured from PCR products of yeast open reading frames (ORFs).

Spellman et al. used a cdc15 temperature sensitive mutant. The cells were grown for several hours at 37°C until they showed the symptoms of cdc15 arrest. They were released from the arrest by shifting the temperature to 23°C. Then, samples were taken every 10 minutes for a total of 300 minutes. The data I use here is a subset of the original data set and contains data for 24 time points (10, 30, 50, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 270, 290 minutes after release from cdc15 arrest).

6.1 Running individual modules

Dealing with missing values

The input data contains $\log_2$ values of all genes tested by Spellman. It also contains a large amount of missing values. I wrote an R function to visualize the missing data. The following diagram shows missing values as black dots with the gene index on the y axis and the time points on the x axis. I only p

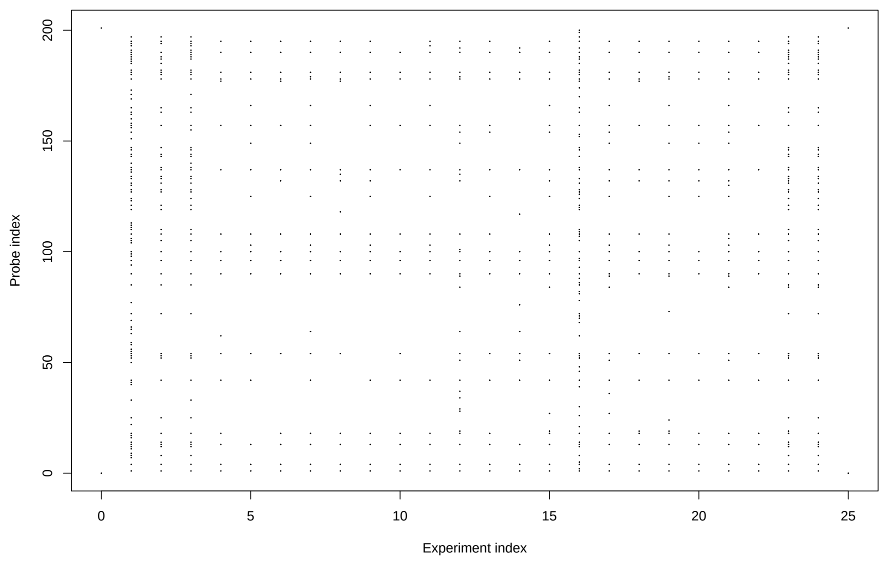
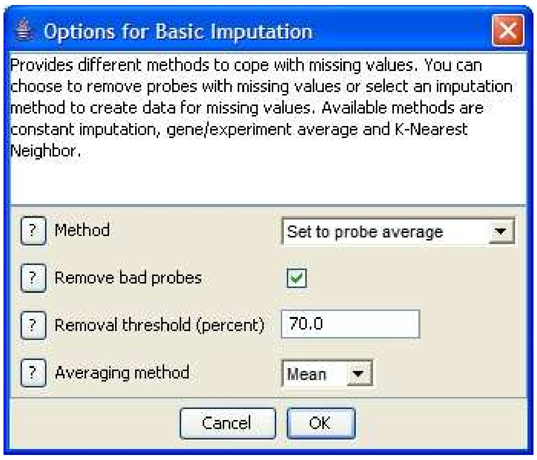


Figure 6.1: Missing value plot of a subset of the Spellman data set. Every dot in this plot represents one missing value.

Most data processing functions require the input data to be complete, i.e. they don't tolerate missing values. So the first step in analyzing the Spellman data is *Imputation*, the removal of missing values. One of the modules included with the MPF is "Basic Imputation" (see figure 6.2). It offers several methods to deal with missing data, ranging from the removal of probes with missing data (a method too harsh for this data set) to a k-nearest neighbor imputation method (that can't be applied here because there are too many missing values). I chose to remove all probes with less than 70% "value coverage", i.e. more than 30% missing values, and to impute the remaining missing values by setting them to the average value of the given probe's existing values.



Options for Basic Imputation

Provides different methods to cope with missing values. You can choose to remove probes with missing values or select an imputation method to create data for missing values. Available methods are constant imputation, gene/experiment average and K-Nearest Neighbor.

Method: Set to probe average

Remove bad probes: ✓

Removal threshold (percent): 70.0

Averaging method: Mean

Figure 6.2: Option dialog for the "Basic Imputation" module.

Normalizing expression values

The next step is data *normalization*. The MPF provides a simple normalizing module, "Basic Normalization", that offers the so-called z-transformation, that is, every probe is transformed so that the mean of its expression values is zero (centering) and the corresponding standard deviation is one (scaling). For the Spellman data set, we only use centering because we're dealing with a time series experiment. Let v_{ij} be the expression value of probe i in experiment j, then the centered expression value v'_{ij} is computed as

$$v'_{ij} = v_{ij} - \overline{v_i}.$$

Questions to the data

One interesting question that we can ask with regard to the Spellman data is whether we can find genes that have similar expression profiles over the course of the cell cycle. If we can find such clusters of genes, the second question would be whether those clusters correspond to our knowledge about the genes. Now we will deal with these questions:

(1) Can we find clusters of genes with similar expression profiles?

(2) Do the clusters correspond to groups of genes of known cell cycle stages?

Filtering on MIOs

Spellman included a cell cycle annotation for a subset of the genes he tested. Of coure we can only answer the second question for the genes contained in the annotation list. Thus the third step of our analysis is removing (filtering out) genes without annotations. The MPF includes several modules for filtering on meta data (MIOs). We will use the "MIO Presence" module.

Mayday imports meta data as key–value pairs where the key is the probe name and the value is the meta information to attach to that probe. Because MPF modules change probe names, we have to import the meta data before doing any processing. The following list shows where we are in processing the data:

1. Open the Spellman cdc15 data set
2. Import string meta data on cell cycle genes
3. Run the "Basic Imputation" module to remove missing values
4. Run the "Basic Normalization" module to normalize data
5. → Run the "MIO presence" module to remove probes without annotation
6. Cluster the data to answer question (1)
7. Try to answer question (2)

After filtering on the presence of meta information, we have a subset of 746 probes (of the original 6178 probes, 657 were removed during imputation. Of the remaining 5521 probes, 4775 did not have attached meta information). The MPF attaches meta information to all probes it modifies as well as to all probe lists it creates. Let's take a look at this information. Our probe list has three associated MIOs that show what we did to obtain this list:

- Processing Pipeline = [Basic Imputation] Method=Set to probe average; Remove bad probes=true; Removal threshold (percent)=70.0; Averaging method=Mean;
- Processing Pipeline (1) = [Basic Normalization] Centering=true; Scaling=false;
- Processing Pipeline (2) = [MIO Presence Filter] Keep probes that=contain the selected MIO group(s);

A randomly chosen probe (YBR073W) in this probe list has the following MIOs:

- A string associated with this probe. = G1
- Processing Pipeline = [Basic Imputation] Method=Set to probe average; Remove bad probes=true; Removal threshold (percent)=70.0; Averaging method=Mean;
- Processing Pipeline (1) = [Basic Normalization] Centering=true; Scaling=false;

Here we can see Spellman's cell cycle annotation for that gene is "G1", that the probe was modified by the imputation module (i.e. it contained missing values) and that the probe's expression values were normalized. We can't see the MIO Presence module here because it didn't modify the probe. Another gene (YBR070C) does not contain meta information about "Basic Imputation" which means that this gene wasn't modified by the imputation module because it didn't contain missing values.

Answering question (1)

Now we are ready to answer question (1). We can use the k-means clustering plugin provided with Mayday. With the default settings we get 9 clusters. As we can see in the following multi profile plot, the answer to our first question is a definite *yes*. We can clearly see clusters of genes with similar expression profiles.

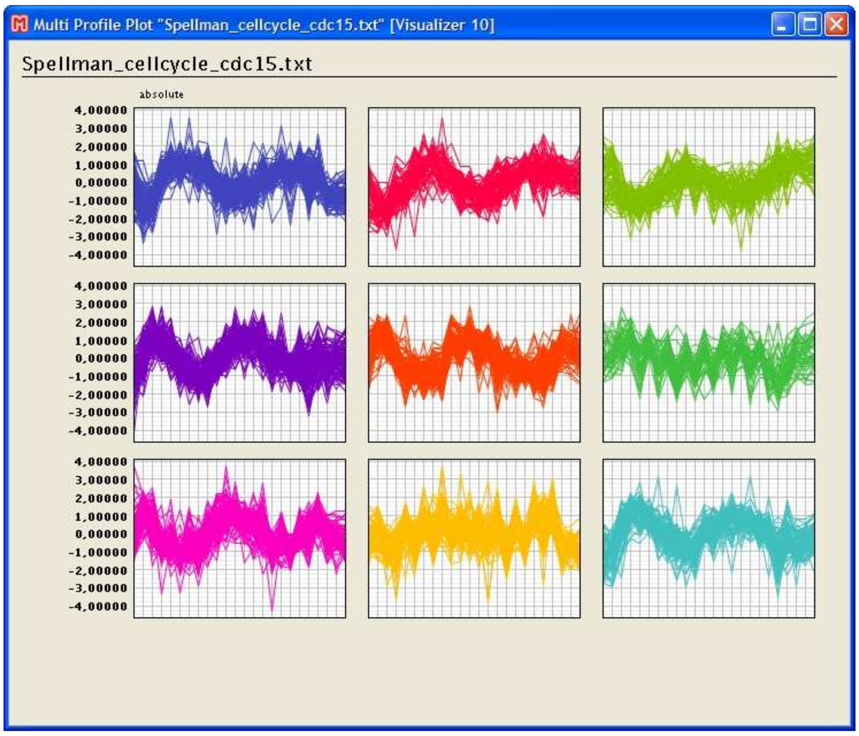
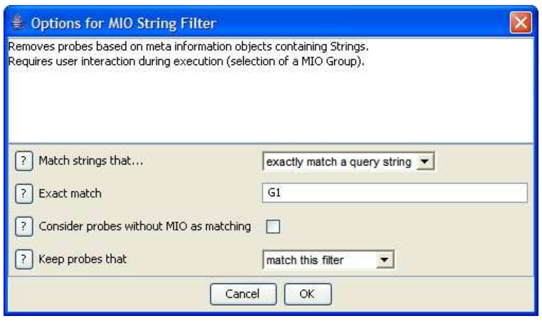


Figure 6.3: A k-means clustering of the processed Spellman data ($k = 9$).

Answering question (2)

To see whether the clustering corresponds to what we know about the tested genes, we can perform the following steps:

1. Use the "MIO String Filter" processing module on the processed data ("global [Basic Imputation] [Basic Normalization] [MIO Presence Filter]") to create a probe list that only contains probes annotated as "G1" by Spellman.



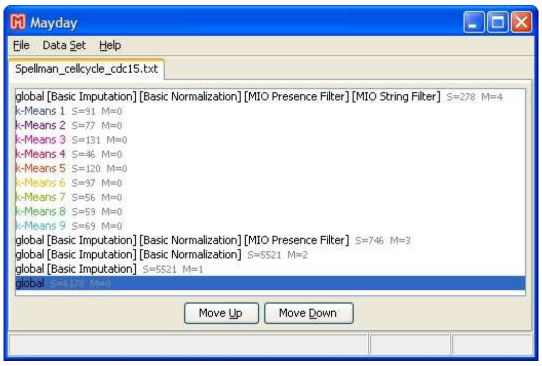
Options for MIO String Filter

Removes probes based on meta information objects containing Strings.
Requires user interaction during execution (selection of a MIO Group).

Match strings that...	exactly match a query string
Exact match	G1
Consider probes without MIO as matching	☐
Keep probes that	match this filter

Figure 6.4: Option dialog for the "MIO String Filter" module.

2. Move the new probe list to the top of the probe list view.
3. Select all probe lists created by k-means along with the one created in step 1.
4. Use the multi profile plot plugin to create a 3×3 plot of all k-means clusters.



global [Basic Imputation] [Basic Normalization] [MIO Presence Filter] [MIO String Filter] S=278 M=4
k-Means 1 S=91 M=0
k-Means 2 S=77 M=0
k-Means 3 S=131 M=0
k-Means 4 S=46 M=0
k-Means 5 S=120 M=0
k-Means 6 S=97 M=0
k-Means 7 S=56 M=0
k-Means 8 S=59 M=0
k-Means 9 S=69 M=0
global [Basic Imputation] [Basic Normalization] [MIO Presence Filter] S=746 M=3
global [Basic Imputation] [Basic Normalization] S=5521 M=2
global [Basic Imputation] S=5521 M=1
global S=6178 M=0

The result of the above steps is the following multi profile plot which clearly shows that most of the genes annotated as belonging to the G1 phase of the yeast cell cycle have been clustered into three of our nine clusters. The clustering isn't perfect: some G1 genes have different profiles and are clustered into different groups. Also, the arbitrary choice of nine clusters splits the G1 group into three clusters.

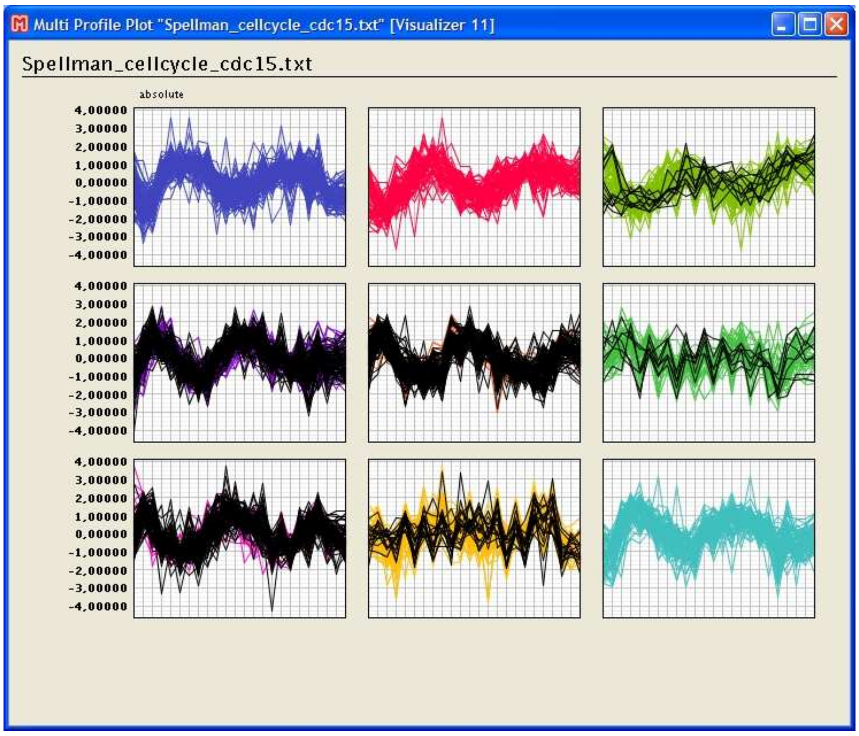


Figure 6.5: The same clustering as shown in figure 6.3. Here, profiles of genes annotated as "G1" by Spellman are drawn in black.

6.2 Creating a pipeline

As the last section showed, performing all analysis steps manually requires quite a lot of work from the user. If we created a processing pipeline, the process could be simplified:

1. Open the Spellman cdc15 data set
2. Import string meta data on cell cycle genes
3. Run the pipeline
4. Cluster the data
5. Create a multi profile plot to answer the questions

The pipeline for step 3 must contain the following steps: Imputation, Normalization, Filtering on MIO presence (this will give our first output probe list) and Filtering on MIO strings (this will be the second output, needed for question (2)).
To create the pipeline, we need the “Basic Imputation”, “Basic Normalization”, “MIO Presence Filter” and “MIO String Filter” processing modules along with a “helper” module, called “Duplicate”. The helper module is needed so that we can return two probe lists. Figure 6.6 shows the resulting pipeline.
Above, we answered question (2) with respect to genes annotated as G1. Yet it would be nice if users of our pipeline could decide which cell cycle phase to use for that test. So we will externalize one option, namely the string to match against in the MIO String Filter module (see figures 6.7 and 6.8).

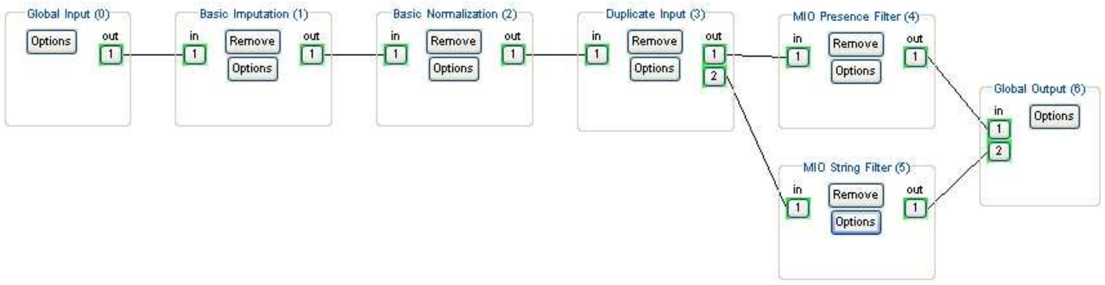
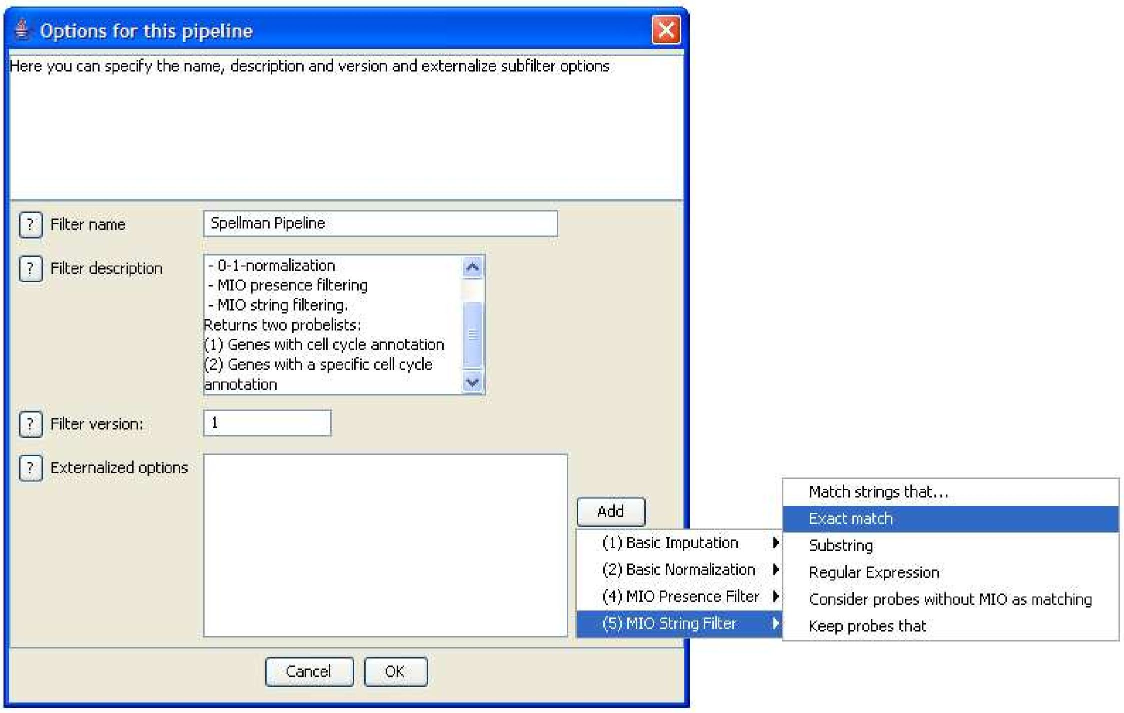


Figure 6.6: The MPF’s rendering of the pipeline used to analyze the Spellman data set.



Options for this pipeline

Here you can specify the name, description and version and externalize subfilter options

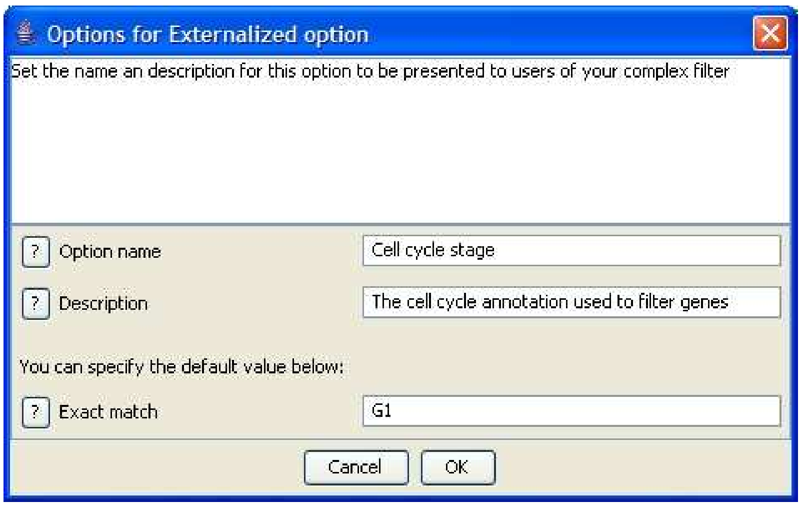
Filter name: Spellman Pipeline

Filter description:
- 0-1-normalization
- MIO presence filtering
- MIO string filtering.
Returns two probelists:
(1) Genes with cell cycle annotation
(2) Genes with a specific cell cycle annotation

Filter version: 1

Externalized options

Figure 6.7: Externalizing an option:
The popup menu lists externalizable options for each submodule of the pipeline.



Options for Externalized option

Set the name an description for this option to be presented to users of your complex filter

Option name: Cell cycle stage

Description: The cell cycle annotation used to filter genes

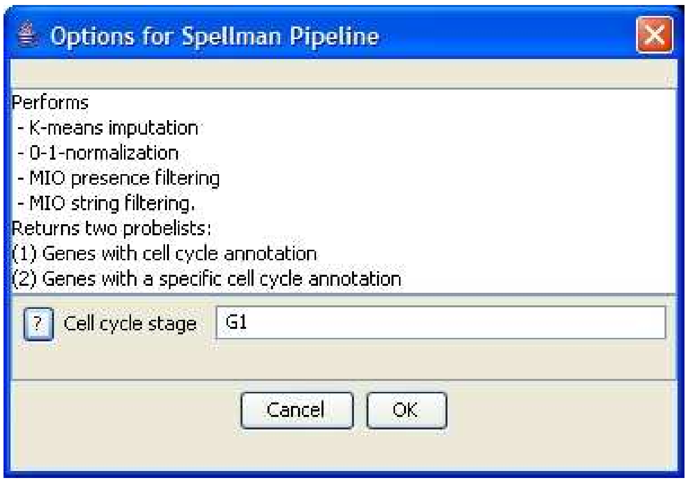
You can specify the default value below:

Exact match: G1

Figure 6.8: The newly externalized option needs a good name and an informative description. The dialog also allows setting a default value.

6.3 Running a pipeline

After loading the data and the meta information, we select the global probe list, start the MPF and choose the "Spellman pipeline" from the list of processing modules.
Before the pipeline is started, we can review the externalized options:

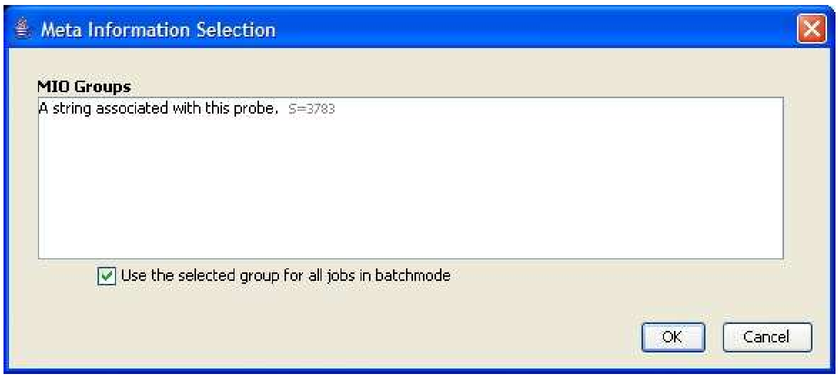


Performs
- K-means imputation
- 0-1-normalization
- MIO presence filtering
- MIO string filtering.
Returns two probelists:
(1) Genes with cell cycle annotation
(2) Genes with a specific cell cycle annotation

Cell cycle stage: G1

Figure 6.9: The option dialog for our pipeline shows the pipeline description as well as the externalized option.

After we click on "Start", the pipeline is executed. Normally, pipeline execution can run without any intervention by the user. Filtering on MIOs is an exception to the rule because the MIO filtering modules needs to know which MIO groups to filter on. This information is not available before the module is asked to process the data, so in this case the user is asked to select one or several MIO group(s) during the execution of the pipeline. For batch jobs, the user can decide whether his initial selection should apply to all jobs.

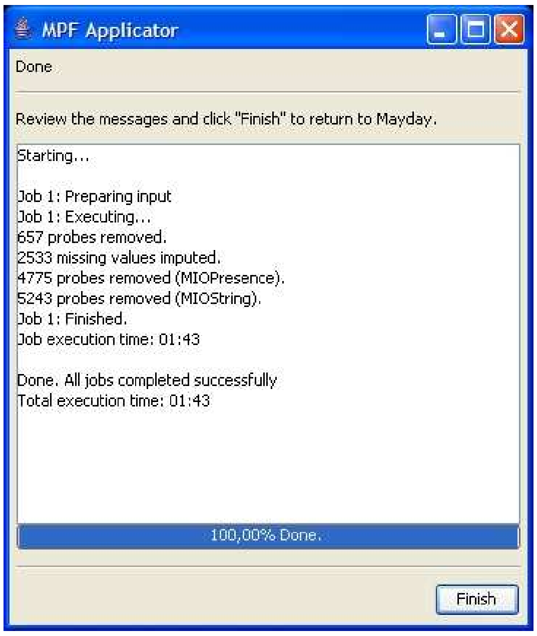


MIO Groups
A string associated with this probe. S=3783

✓ Use the selected group for all jobs in batchmode

Figure 6.10: MIO group selection can only be done during pipeline execution.

When the pipeline has finished processing the input data, it is time to review the log messages it produced (see figure 6.11). We can see that imputation as well as filtering on MIOs removed a large number of genes (the numbers are equal to those presented in the previous section) and, most important, that the pipeline ran without any errors. Now the last thing to do is using Mayday's k-means clustering plugin and creating a multi profile plot just as described before.



```
Starting...

Job 1: Preparing input
Job 1: Executing...
657 probes removed.
2533 missing values imputed.
4775 probes removed (MIOPresence).
5243 probes removed (MIOString).
Job 1: Finished.
Job execution time: 01:43

Done. All jobs completed successfully
Total execution time: 01:43
```

Figure 6.11: The pipeline was executed without errors.

Using the processing pipeline has an additional benefit apart from making life easier for the user The combination of several modules into one pipeline reduces the number of MIOs added to the resulting probes. Probes processed by the Spellman pipeline contain only two meta information objects (instead of four as seen in the previous section):

- A string associated with this probe. = G1
- Processing Pipeline = [Spellman Pipeline] Cell cycle stage (MIO String Filter (5): Exact match)=G1;

7. Outlook

Many things can be done to further improve the MPF but two issues are particulary worthy of attention: Firstly, it would be nice if Designer could perform some kind of *semantic* graph validation apart from testing whether the filter graph is cycle free and contains all needed connections. This semantic validation would for example warn users if the pipeline they created “makes no sense”: Consider a pipeline that processes data with a processing module M before performing certain statistical tests. The results returned by M could violate the assumptions implicitely formulated by the statistical test. This kind of validation obviously requires Designer to know very much about individual processing modules. Considering the fact that processing modules can do virtually anything to the data they are given, it is hard to see how such knowledge could be represented, checked, or even formulated. While an interesting field for debate, implementing semantic validation certainly lies beyond the scope of a three-month student work.

Secondly, it would be nice to integrate more native Mayday plugins into MPF processing pipelines, especially visualization plugins. As long as there is no common interface for calling these plugins without user intervention, this integration can only be done by manually changing every single plugin. Maybe the next version of Mayday (Mayday 3.0) will be a step towards a more standardized API for inter-plugin communication. Until then, the `RWrapper` offers a way of visualizing data from within processing pipelines, with the drawback that every visualization function has to be re-implemented as an R function.

8. Acknowledgements

I wish to thank Dr. Kay Nieselt and Janko Dietzsch for their support as well as Matthias Zschunke for providing me with useful insights into the RPlugin.

References

[DGN06] Janko Dietzsch, Nils Gehlenborg, and Kay Nieselt. Mayday – a microarray data analysis workbench. *Bioinformatics*, 22(8):1010–1012, 2006.

[GDN05] Nils Gehlenborg, Janko Dietzsch, and Kay Nieselt. A framework for visualization of microarray data and integrated meta information. *Information Visualization*, 4(3), 2005.

[Geh04] Nils Gehlenborg. *Visualization in Mayday*, 2004. Studienarbeit.

[Moh06] Vikram Mohan. Workaround for Maximize button on a JDialog. *Published online at http://vikram.blogspot.com/2005/06/workaround-for-maximize-button-on.html*, June 30, 2005 (last checked in April 2006).

[R D05] R Development Core Team. *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[Spe03] Terry Speed. *Statistical Analysis of Gene Expression Microarray Data.* CRC Press Inc.,U.S., 2003.

[SSZ$^+$98] Paul T. Spellman, Gavin Sherlock, Michael Q. Zhang, Vishwanath R. Iyer, Kirk Anders, Michael B. Eisen, Patrick O. Brown, David Botstein, and Bruce Futcher. Comprehensive Identification of Cell Cycle-regulated Genes of the Yeast Saccharomyces cerevisiae by Microarray Hybridization. *Mol. Biol. Cell*, 9(12):3273–3297, 1998.

[Zsc04] Matthias Zschunke. *Connecting R to Mayday*, 2004. Studienarbeit.