Akonadi Filtering Framework

Current State

07.Jul.2009

Szymon Tomasz Stefanek

Location and structure of the sources

Source tree in

`playground/pim/akonadi/filter`

`CMakeLists.txt`

Global compilation rules: builds everything in the project
No particular configuration needed.

Subdirectories:

`akonadi/filter`

The filtering framework libraries.
(The source directory tree emulates the installed one)

`agent`

The filtering agent

`console`

A small demo program

The filtering framework libraries

Two libraries:

libakonadi-filter.so

```
The filtering framework core.
Contains the filter tree model, the Sieve
decoder and encoder and the tools needed
for filter customisation.

Akonadi::Filter::*
Akonadi::Filter::IO::*
```

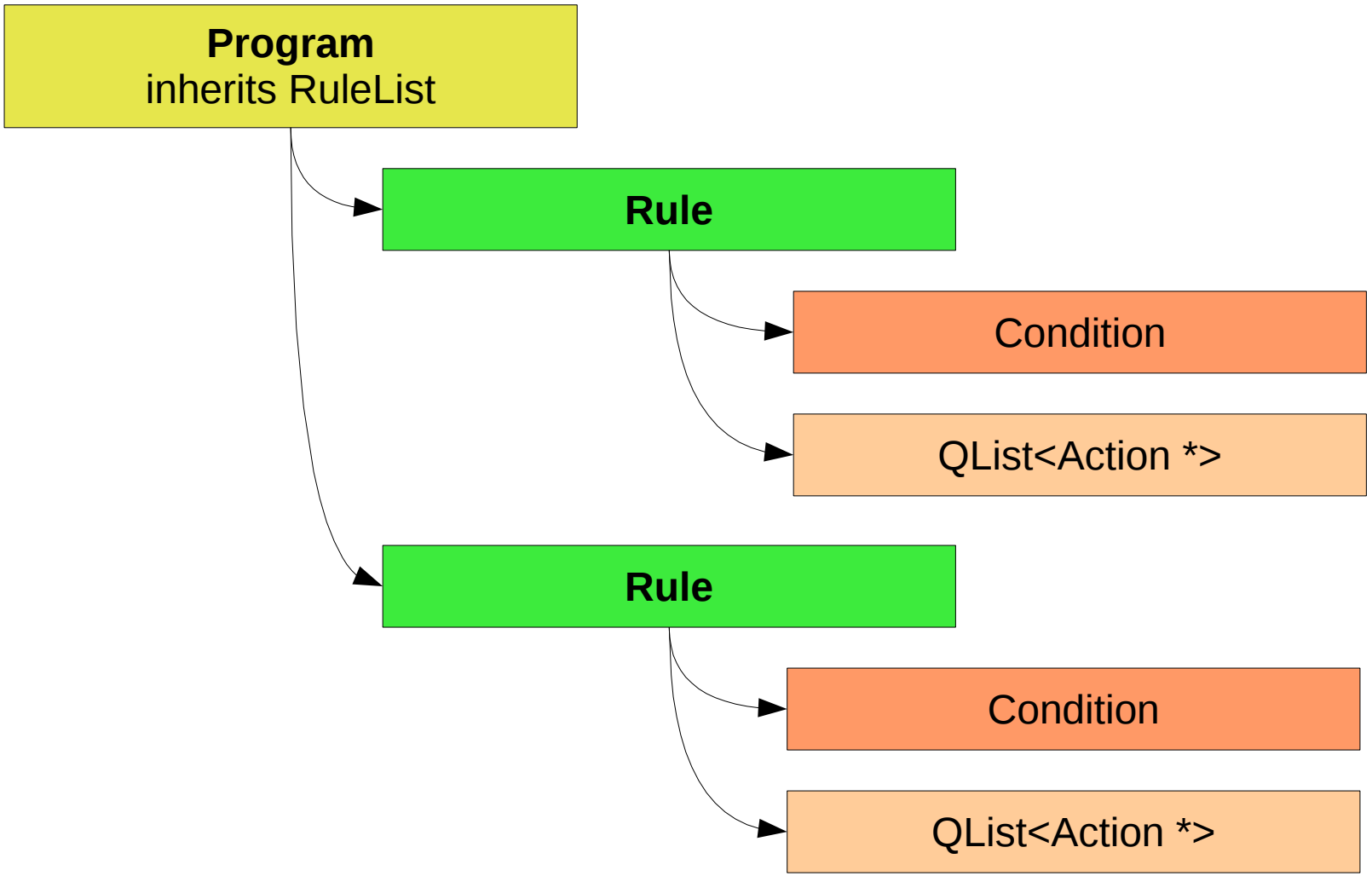
libakonadi-filter-ui.so

```
The user interface for filter editing
And the tools needed for filter customisation.

Akonadi::Filter::UI::*
```

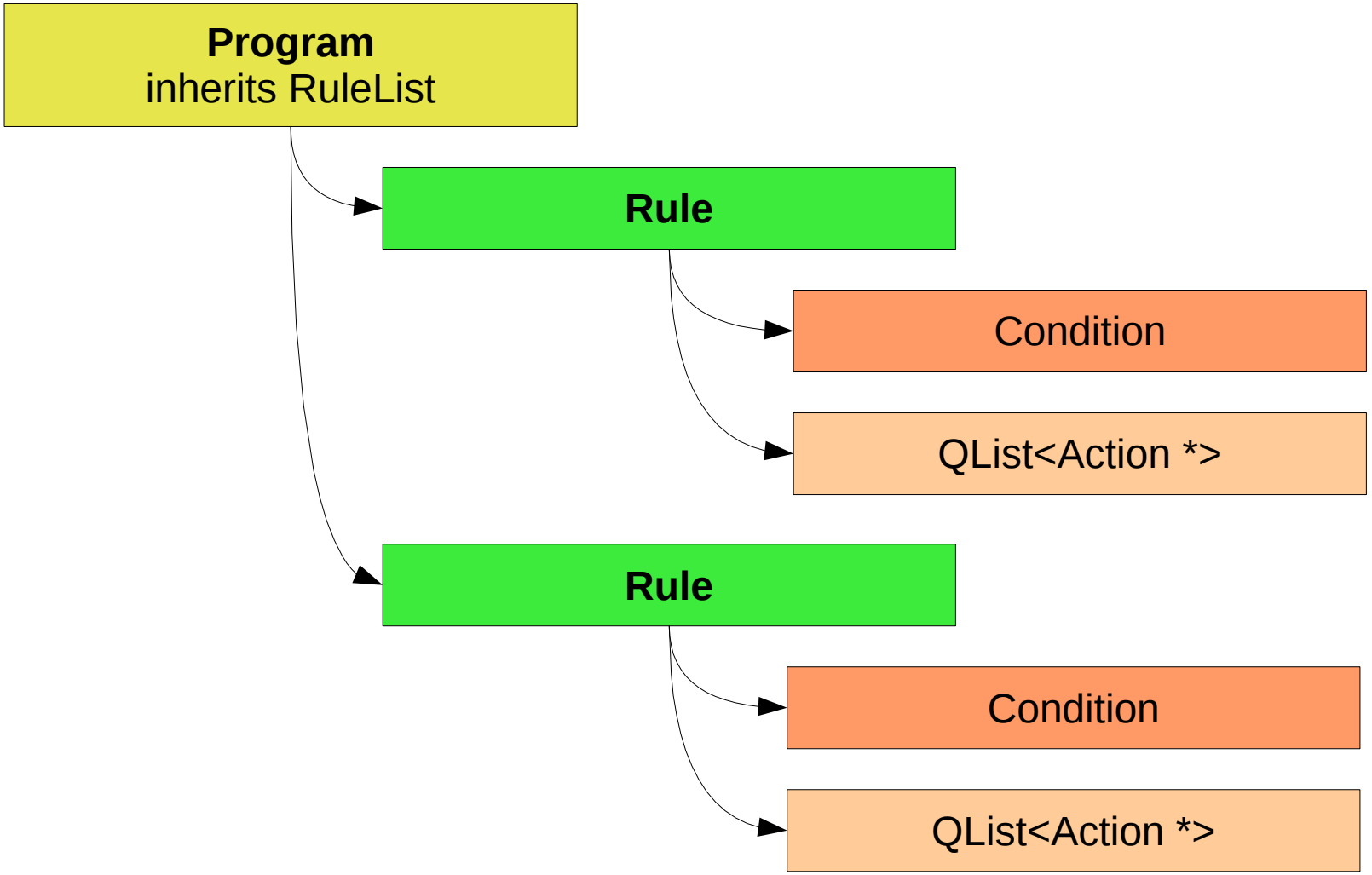
The filter model

A filter is a program.
The memory model is a tree.



The filter model

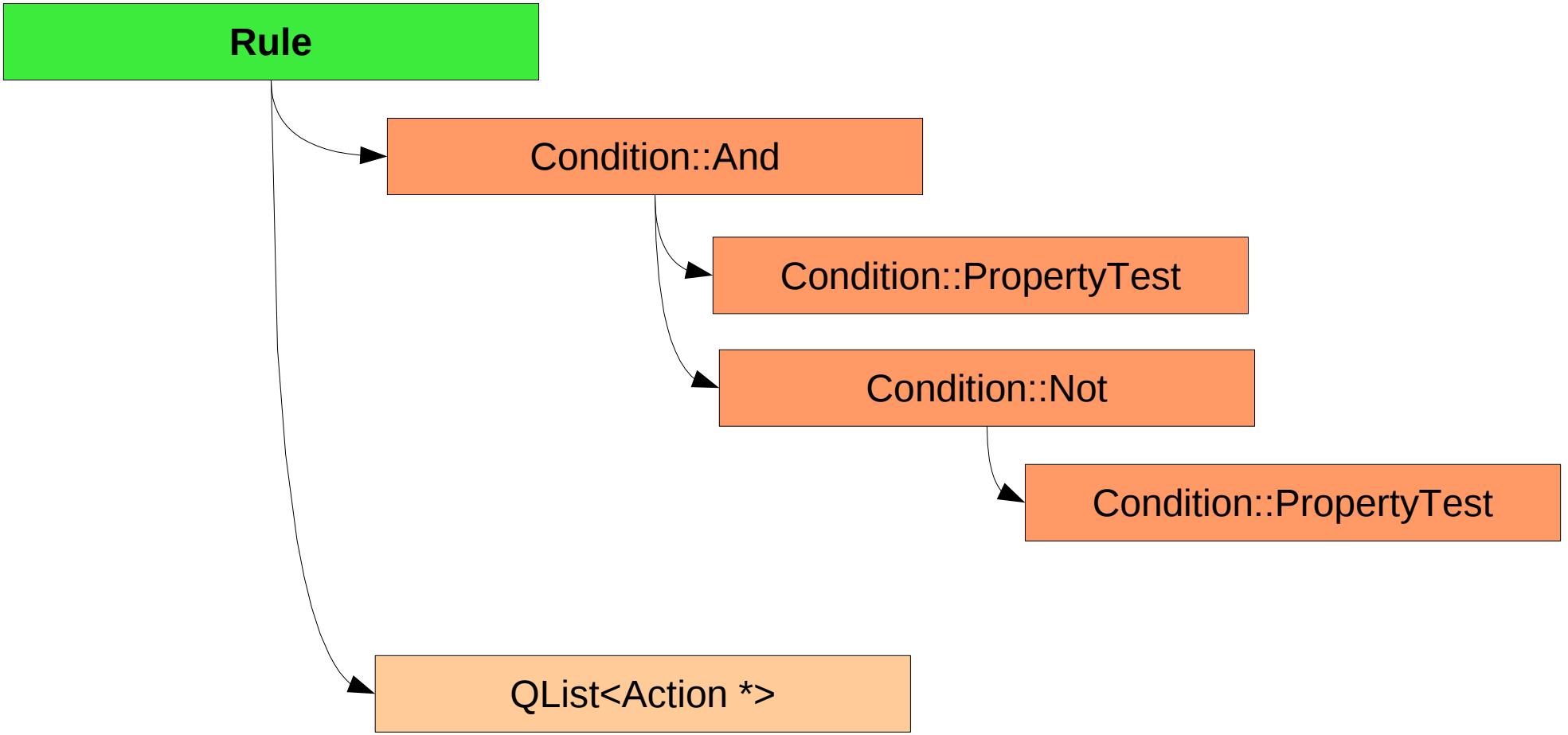
A filtering program is a **list of rules.**
The rules are applied in sequence until a stop condition is encountered or an error occurs.



The filter model

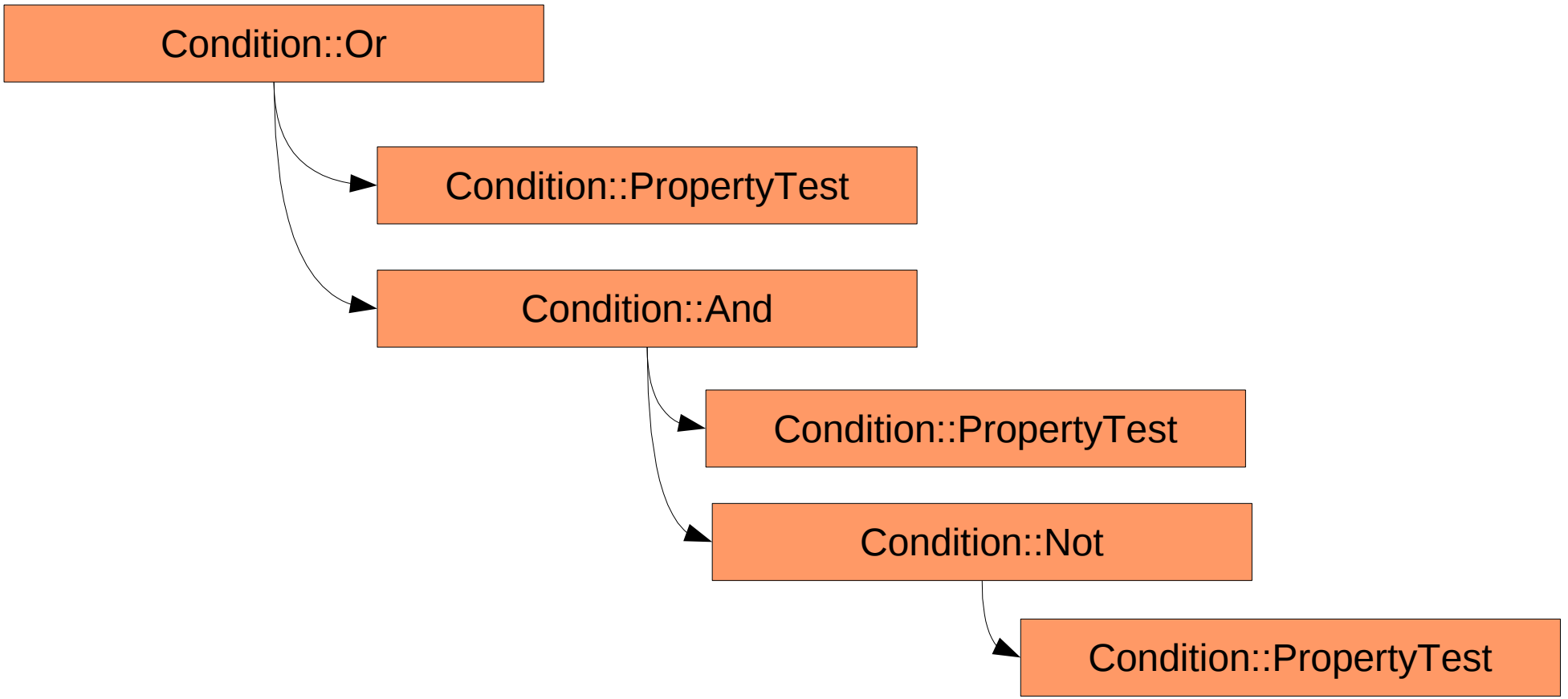
A rule is made of a condition and a list of actions. If the condition matches() then the actions are executed in sequence.

The condition is itself a tree: no limit to nesting.



The filter model

Non leaf condition nodes:
And, Or, Not
(Current) leaf condition nodes:
PropertyTest, True, False



The filter model

The **PropertyTest** condition encodes a test on **Data**
The most general format of a test is

Function(DataMember) Operator ConstantOperand

ValueOfHeader(“From”) Contains “foo@bar.org”

The DataType of the components must agree:

Function must accept the DataType of DataMember,
Operator must accept the result DataType of Function on the left...

The filter model

The available **Function**, **DataMember** and **Operator** objects are stored in the ComponentFactory

The **ComponentFactory** is the primary mean of customisation of filters.

It's responsible of:

- **creating instances** of filter tree nodes
  (so you can provide custom ones)

- providing the description of the available
  Condition and Action types so
  - IO layer knows what/how to decode/encode
  - UI facilities can provide editors

The filter model

In the case of Conditions, for example, you can:

- register your own Function objects
  (say... “sizeof”)
- register your own DataMember objects
  (say... “any attachment”)
- register your own Operator objects
  (say... “matchesWildcard”)

Or even

- provide a fully custom condition which uses a totally different internal test model

“Filters operating on different items (or mimetypes) may provide different DataMember objects”

The filter model

Actions follow a similar cusomization scheme. By now only few actions are implemented.

ActionTypeStop
just stops unconditionally (default)

ActionTypeRuleList
a fully nested sub-filter!

ActionTypeCommand
generic command stored in Sieve format
keep, download, doNotDownload...

moveToFolder will be probably “hidden” here.

The ComponentFactory inside the library provides a set of basic actions which will be registered “on demand”. More advanced actions can be registered “on-the-fly” by the specific filter implementation.

The filter model

The **IO** namespace (io subdirectory) contains filter encoding and decoding classes.

Encoder
- **SieveEncoder**
- **AFLEncoder**
- **WhateverEncoder**

Decoder
- **SieveDecoder**
- **AFLDecoder**
- **WhatewerDecoder**

Actually only Sieve IO is complete. Othe formats can be “plugged in” here. AFL (Akonadi Filtering Language) is something that could be finished before GSoC ends and could end up being “nicer” than Sieve (which has its drawbacks)...

The filter model

Note that until now, nothing really depends on Akonadi yet.

This part of the filtering model could be even used standalone.

The filter model

To execute a filter you “throw” a Data subclass through it.

The Data object wraps the real data being filtered.

This is where the dependency on Akonadi **MAY** be plugged in at agent level.

In the Agent source there will be a DataRfc822 class which wraps an Akonadi::Item with a KMime::Message payload.

The POP3 module might wrap a different data object: it's enough that the Data interface is implemented.

Filter Editing

The akonadi/filter/ui contains the sources for the libakonadi-filter-ui library.

The library is rooted at the

Akonadi::Filter::UI

Namespace.

Obviously:

- every filtering program is bound to use libakonadi-filter.so

- only UI programs will take advantage of libakonadi-filter-ui.so

Filter Editing

Customization is provided via the

EditorFactory

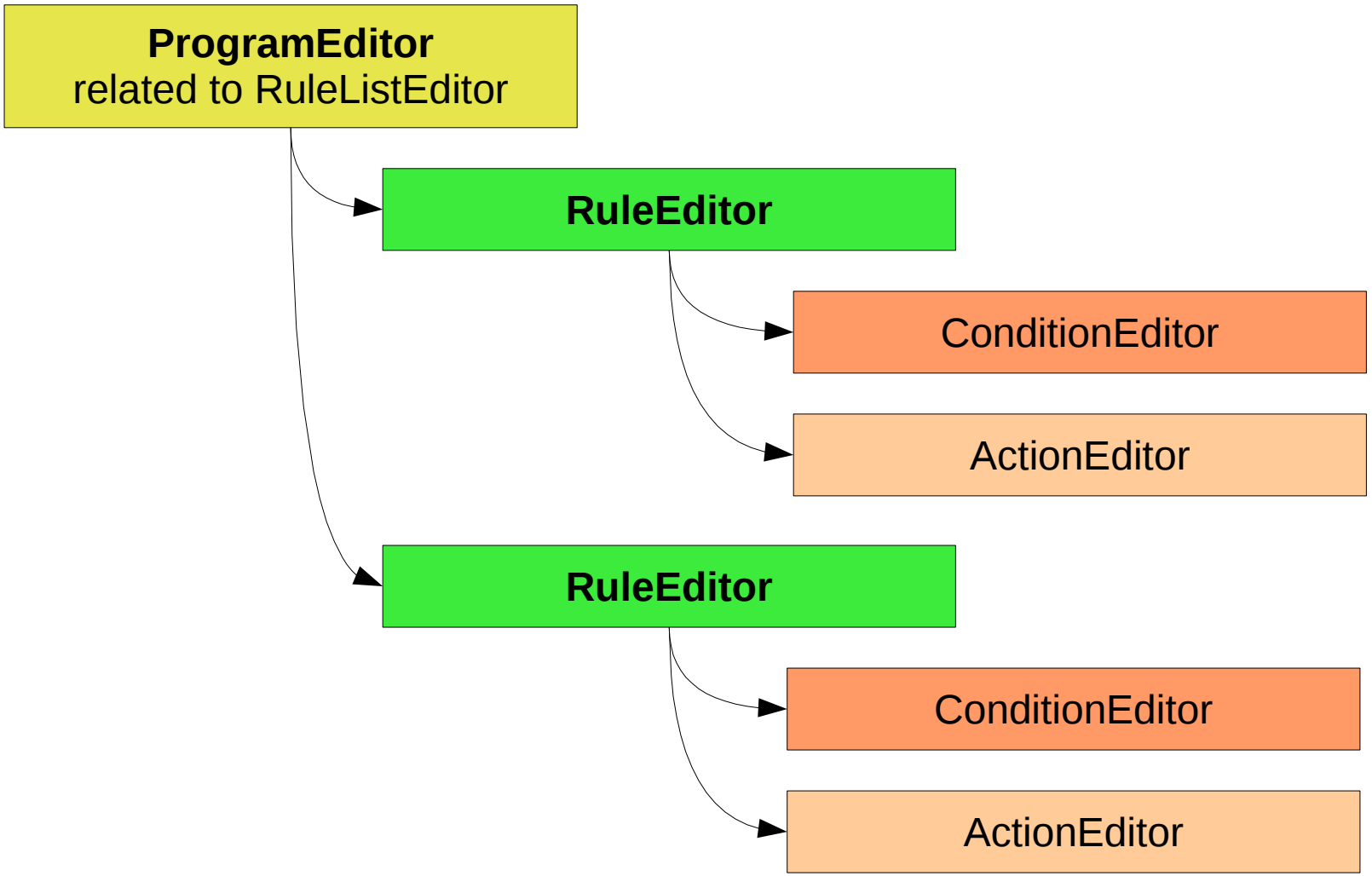
class.

EditorFactory role is very similar to ComponentFactory in the filter model library.

EditorFactory actually depends on a specific ComponentFactory implementation.

Filter Editing

The filter model is “mirrored” in a tree of UI classes.



Filter Editing

The current implementation use a “vertical” editing approach:

- **there is no list box on the left.**
- **the rules appear one after another in an object resembling a QToolBox**

QToolBox wasn't customizable enough: I had to write my own “stacked tools” widget.

Vertical editing:

- makes good use of space
  (thing of nested sub-programs)
- gives the impression of really editing a program = sequence of rules

Filter Editing

You can take a look at the editor by using the demo console.

- **Via akonadiconsole create an instance of the filtering agent**

- **Run akonadi_filter_console**

- **Click on “add filter”**

The editor is on the second tab, more about the first tab later.

Please note that the editor is still “fragile”, especially the constant value boxes in the conditions.

The filtering agent

The main akonadi filtering agent in in the agent/ subdirectory.

filteragent.cpp / filteragent.h:

FilterAgent: public Akonadi::AgentBase

- It exposes the FilterAgent D-Bus control interface

- It hooks on the itemAdded() signal In order to apply the filters.
  (this is the part we need to fix)

The filtering agent

The agent contains multiple **ComponentFactory** instances: one for each mimetype we want to filter.

When you create a filter via akonadi_filter_console you need to specify its mimetype (actually only message/rfc822 works).

Each filter has an **unique identifier** which the D-Bus calls require in order to operate.

The filtering agent

You can look at the D-Bus interface via qdbusviewer.

The methods are commented in filteragent.h

createFilter(<id>,<mime>,<sieveSource>)
deleteFilter(<id>)
attachFilter(<id>,<collectionIds>)
detachFilter(<id>,<collectionIds>)
getFilterProperties(<id>)

The filtering agent

A filter can be “attached” to multiple collections.

Actually this is implemented as

- **enable notifications for collection X**

- **handle itemAdded()**

This has the drawback of items being temporairly visible in a collection and then possibly disappear after the filter has been applied. We possibly need a server modification here.

The filtering agent

The first tab in the akonadi_filter_console contains:

- The specification of the filter id
  Actually you create it manually
  as it's an arbitrary string.
  We might want a naming protocol here

- The specification of the collections
  that the filter is hooked to.

Yeah

That's basically it.

There are many gory implementation details, but this is the general picture of how it works now.

I need your advice to solve:

- **the problem of hooking the filter an item arrival**
- the (easier) problem of manual filtering (Thomas suggests a D-Bus call for this).
- the problem of “no-filtering” for manual item movements (tags?)