Programming Methods Laboratory (LAMP)
School of Computer and Communication Sciences
Swiss Federal Institute of Technology, Lausanne (EPFL)

Extensible Java Profiler

User Manual

Revision 1.00
February 2002

Sebastien Vauclair
Appendix to Diploma Thesis

License

Table of Contents

Table of Illustrations

1 Introducing the Extensible Java Profiler

1.1 What is the Extensible Java Profiler?

The *Extensible Java Profiler* (*EJP*) is a Java™ profiling tool that enables developers to test and improve the performance of their programs running on the JVM. A profiling tool measures and reports the time spent in different parts of a program. Using it allows developers to identify any code making heavy use of the CPU. Pure Java programs as well as programs in any custom language using a JVM backend, such as *Funnel* [1], can be profiled with EJP.

EJP aims at producing exact and precise reports of Java execution times, rather than probabilistic ones, by recording every method call with its precise time duration. Most other profiling tools for Java use a *sampling* method to gain CPU usage information: the current stack trace is being periodically dumped and same instances of it are counted to compute a list of most frequent encountered stack traces. This method is efficient and produces good statistical reports, but developers might need the precise trace and cost of all methods calls. These can be obtained using EJP.

In addition, EJP provides a convenient programming interface that allows choosing what information is presented to the user and the way it is displayed. This is mostly useful for developers profiling applications in custom languages, as they might need to focus on their own code and make abstraction as far as possible of the relying Java implementation.

1.2 Key concepts

The *trace* of a program execution is the report of all the methods that were called, and for each what were the caller and the time spent before the method returned. This report distinguishes time spent inside the method's body (the method's *own* time) and time spent inside its possible callees. Traces are represented as *call trees*, each of whose node corresponds to the call to a specific method and which children nodes are the callees methods.

When presenting such call trees to the user, it might be interesting to hide useless information or highlight important spots. Such transformations can be done by applying a given set of *filters* to it. Filters modify a tree by changing its structure or its visual aspect (like the coloration of the nodes). By filtering information, they allow the user to avoid seen encoded versions of Funnel (or other) language concepts. The user can therefore have a vision at the level of the custom language source code, rather than at the level of its Java translation.

A *view* is the presentation of a call tree on which a specific set of filters has been applied. The meaning of the information displayed in a view is given by its filters.

1.3 Getting started

The basic steps to run the *Extensible Java Profiler* are as follows.

- Create a trace file of the target application. This file contains timings for all method calls of the program and is created by running and monitoring the program execution. This is done by *EJP Tracer*, which is a Java native library that records all method calls.

- Open the trace file in a specific program, *EJP Presenter*, which is a Java application able to create views of the resulting call tree with user-chosen sets of filters.

1.4 Main features

- Important work has been done in EJP Presenter to offer a user-friendly system comparable to other commercial profiling products.
    - A Swing graphical user interface allows easy loading of trace files, creation of new views and browsing of them.
    - Possibly many trace files can be opened at the same time, with possibly many views per trace file.
    - Window management *tile* function is provided, that rearranges all windows on screen to fill available space. Processing can be limited to active windows or to those showing a specific trace file.
- The main feature of EJP Presenter is its focus on extensibility.
    - Filters use a *plug-in* interface allowing users to add new functionalities to the software.
    - User-editable XML files are used to store configuration settings and to register available filters.
- Much effort has been done to improve the software's scalability in both EJP Tracer and Presenter.
    - Trace files are highly compressed: EJP Tracer creates trace files sparingly and applies *zlib* compression [2] on them. EJP Presenter supports loading of both compressed and uncompressed trace files (files can be uncompressed using Unix `gzip` utility).
    - The monitoring (and dumping) of method calls by EJP Tracer can be limited to some specific parts of the code by the user. This is done from the user application through the *Tracer API* (see section 4.6). This way, critical parts of huge programs can be separately profiled.
    - In EJP Presenter, the tree displayed to the user is built dynamically using a lazy application of filters: node's children are computed only when the node is actually expanded. This way, information not required by to the user will never be computed, thus resulting in a possibly significant gain of processing time.

2 Profiling a program

2.1 Configuring EJP

2.1.1 Prerequisites

EJP uses many features of J2SE 1.4 [3]. This (or any later) version is therefore required to run it. EJP was tested successfully with both versions *1.4.0-beta3* and *1.4.0-RC1*.

In addition, if Funnel [4] programs are to be profiled, the language's runtime environment must be present in Java *class path*.

2.1.2 Packaging

EJP is distributed in two distinct packages that can be installed separately, respectively named `ejp-tracer-`*`version`*`.zip` and `ejp-presenter-`*`version`*`.zip` for EJP Tracer and EJP Presenter. They contain both source and binary distributions. It is suggested (but not required) that they are respectively installed in `ejp-tracer` and `ejp-presenter` directories.

2.1.3 Configuring EJP Tracer

The tracer requires its native library to be in the system path. This can be done by manually adding `lib` directory to `PATH` system variable, or by directly running the Java interpreter from the `lib` directory. To use the *Tracer API* (described in section 4.6), the Java library `tracerapi.jar` must also be present in Java class path.

A script is provided (for Windows and Linux) in then `bin` directory, that temporarily adds `lib` to system path and `tracerapi.jar` to Java class path and starts tracing a target application.

2.2 Running a program with EJP Tracer

EJP Tracer is to be run as an extension library of the JVM. The standard[1] way of doing this is to use a specific switch of the Java interpreter. To check that EJP Tracer is configured correctly and view its usage help, use the following command line from the `lib` directory.

```
java -Xruntracer:help
```

To trace a program, use the `run-enabled` script (in `bin` directory) or directly start the Java interpreter from the `lib` directory to run your application.

```
run-enabled application-main-class
java –Xruntracer application-main-class
```

[1] According to the Java Platform documentation, the only way of running a native library is using the `-Xrun` option of the `java` tool, even though this option is part of the non-standard ones and could change without notice. The method described here is the *de facto* "standard non-standard" way to run a library.

By default, tracing of method calls is enabled. To start profiling an application with tracing disabled, set the `enabled` option to `false` (use `–Xruntracer:enabled=false`) or use `run-disabled` script.

Once the target application exits, a trace file named `tracer-output.bin.gz` is available in the current directory.

2.3 Opening the trace file with EJP Presenter

To start EJP Presenter, use the provided `run-jar` script (in `bin` directory), or directly go to the `lib` directory and use the following command line.

```
java –jar ejp-presenter.jar
```

The main window is shown in Figure 2-1. It contains in its top a menu used to issue various commands like loading and closing profiles, creating views and exiting. The bottom of the window contains a status line to which messages concerning the current state of the program are written.

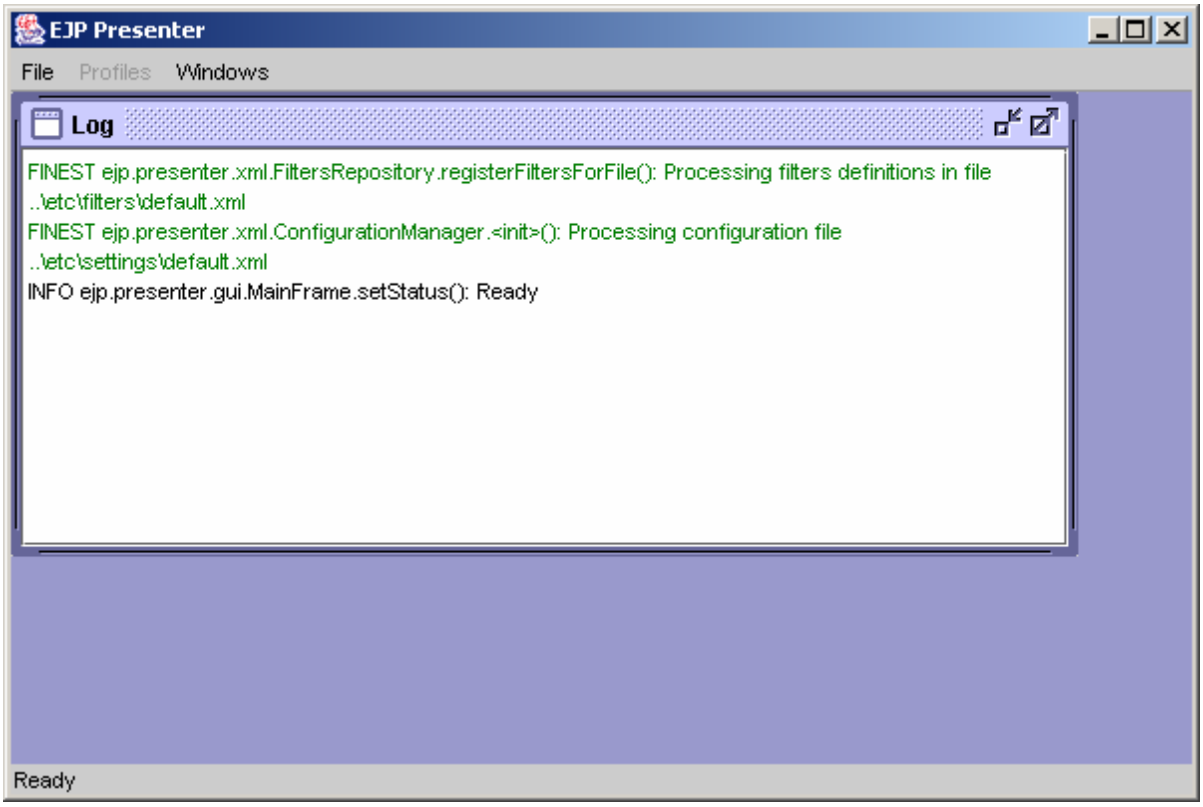

Figure 2-1 – Presenter window and Log frame

An internal frame of the application, called the *Log* frame, recalls all messages written to the status line and displays additional verbose information. Warning messages, if any, are also written to it. All messages are colored depending on their *severity level*, which is displayed in the beginning of every line. These levels are *finest*, *finer*, *fine* (verbose messages, all three displayed in green), *config*, *info* (interesting information, both in black), *warning* (non-fatal error, colored in red) and *severe* (fatal error, in dark red).

At startup, no trace file is loaded. To open one, choose menu *File* → *Open* and select it in the file chooser dialog. Since loading might be a long process, the user can cancel it at any time (see Figure 2-2). However, in that case the resulting profile will not be viewable.

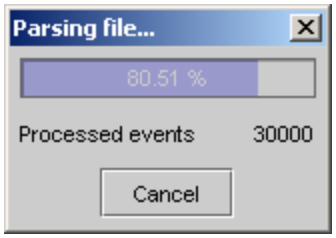

Figure 2-2 – Trace file loading progress dialog

2.3.1 Choosing active filters

When creating a new view, the user can choose the set of active filters and the order in which to apply them. Available filters are organized into a *directory* structure, each filter being associated with a named path. The list of available filters is therefore presented to the user as a hierarchical tree.

New filters can be added or removed from the active filters list (using *Add* command on a directory recursively adds all filters it contains). Every filter may be included only once per view. The user can order the active list by applying move up or down operations to specific filters. The graphical window used to choose active filters is shown in Figure 2-3.

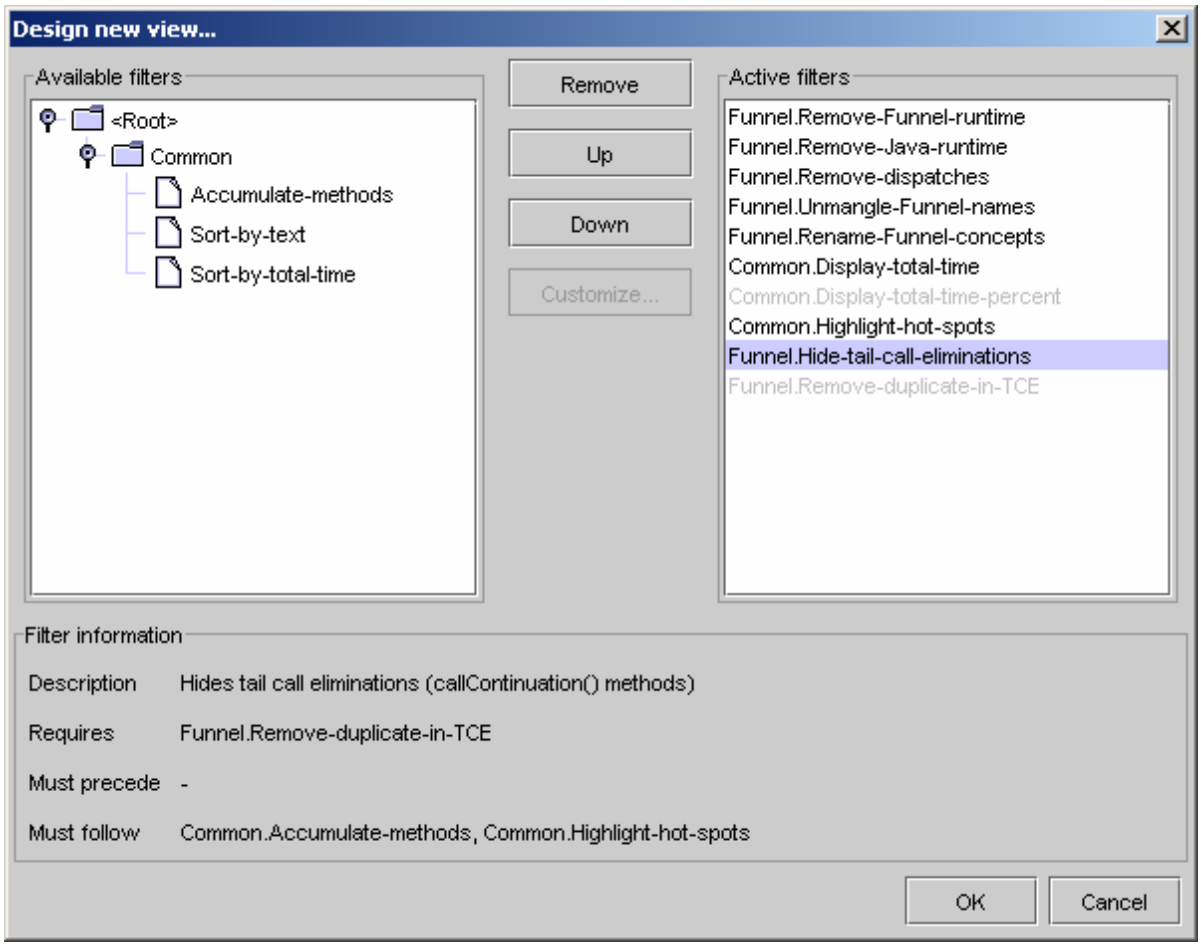

Figure 2-3 – View creation dialog

Some active filters can be customized by the user. For example, a filter which colors method calls depending on their time cost (expressed as a ratio of either the total program time or the caller's time cost) allows the user to choose its color distribution (see Figure 2-4). For any filter, current values can be saved to be automatically restored on next program run.

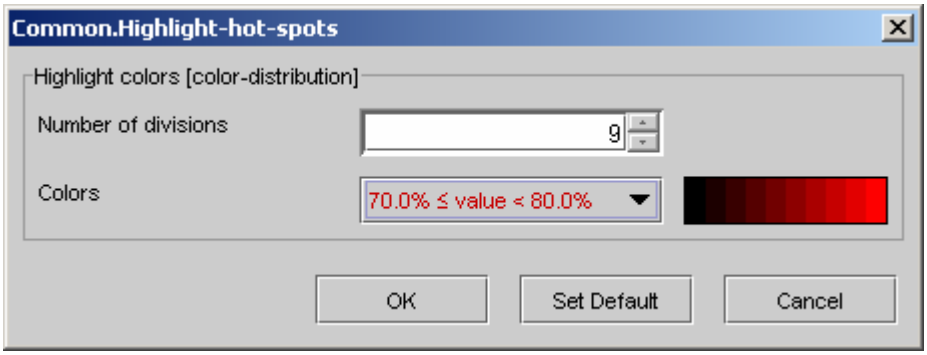

Figure 2-4 – Customization dialog for hotspot highlighter

Filters may have constraints defined between each others. Filters definitions are stored in XML files (see section 2.4).

- One filter may need another one to be also active to work correctly; this relationship is expressed as a *requirement* constraint. Note that requirements do not give any indication on application order of the filters.

$$F_1 \text{ requires } F_2 \overset{def}{\Leftrightarrow} \left(F_1 \text{ is active} \Rightarrow F_2 \text{ is active}\right)$$

- Conflicts may arise between filters working on the same aspect of the information. Some filters may also need to be run before or after other filters. These cases must be solved by defining *precedence rules* between filters. These rules indicate which filter will be applied first, by defining a partial order between filters.

$$F_1 \text{ must precede } F_2 \overset{def}{\Leftrightarrow} F_2 \text{ must follow } F_1$$
$$\overset{def}{\Leftrightarrow} \left(\left(F_1 \text{ is active} \wedge F_2 \text{ is active}\right) \Rightarrow F_1 \text{ is applied before } F_2\right)$$

When inserting a new filter, the program automatically handles all constraints defined on it. Any required filter is automatically inserted *after* the new filter, and the whole list of active filters is ordered to satisfy the maximum number of constraints as possible, while trying to keep user-defined order. If conflicts remain, i.e. cycles exist in filters precedence rules, filters cannot be perfectly ordered and one of the conflicting filters is shown in red in active filters list. Note that provided set of filters does not contain such cycles, but they might happen in user-defined filters. For a more detailed description of the algorithm used to order active filters, see [5].

Active filters list (right part of Figure 2-3) handles requirement constraints as follows:

- filters that are required by other active filters are shown in grey and cannot be removed as long as all requiring filters are active;

- filters that have been automatically activated to satisfy a requirement are also automatically removed when they are no longer needed (if they had not been manually activated before).

2.3.2 Browsing views

Every view recalls in its title line its source trace file between square brackets "`[]`" and the list of active filters between "<>" signs. To reduce the length of this line, trace file name is shortened by removing its path, and filter paths are abbreviated by keeping only the first letter of each path component (for example, a filter which fully qualified name is "`First.Second.Third.My-Filter`" gets abbreviated "`F.S.T.My-Filter`").

Profiling information is presented as a call tree, every method call being represented as a specific sub-tree. Since methods might in turn call other methods, a tree is recursively built up. By default, the tree shows all method calls in the order in which they occur, but filters might hide some calls or order them in a specific way. The time spent inside methods is shown in a *tool tip*, as well as their source file and line (see Figure 2-5).

In order to avoid having to compute (by applying active filters) the whole tree before displaying it to the user, the tree is built dynamically (or *lazily*), i.e. no information is processed before the user requests it. Practically, a node is actually computed only when the user expands it.

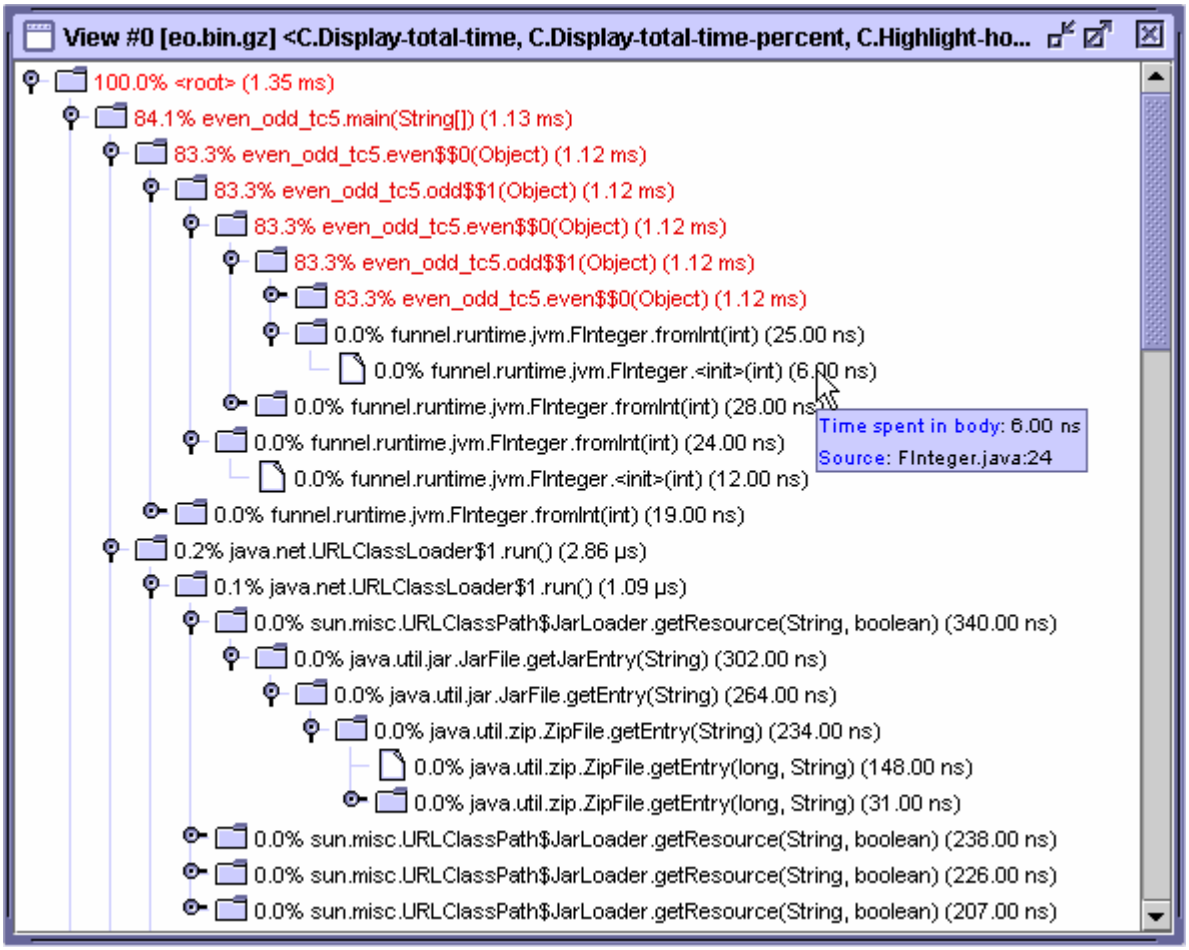

Figure 2-5 – Data presentation with time display and methods coloration

On any view, using the popup trigger (mouse right-click on Windows and Linux) opens a dynamic menu, some of which options are enabled only if a node was selected. The menu allows to:

- view active filters (as in Figure 2-3) for current window, yet without the ability to change them;
- automatically expand the sub-tree rooted at selected node, possibly limiting expansion to a given number of levels;
- run a given program based on selected node information. This feature is mostly intended to browse source code of selected method definition. When activating this option, the dialog shown in Figure 2-6 opens.

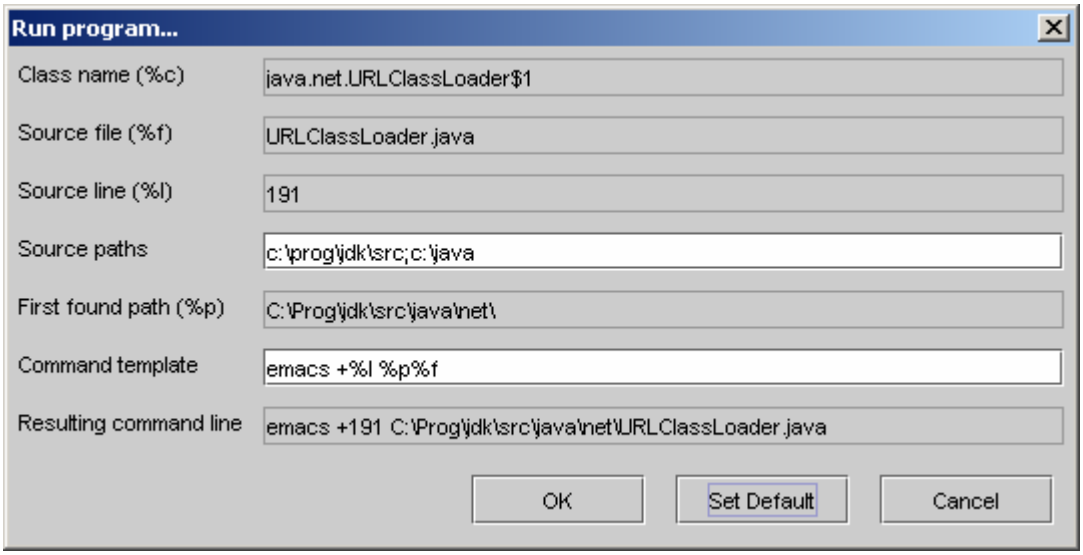

Figure 2-6 – Run Program dialog

This dialog allows the user to enter values (or use saved default values) for two fields:

- the source paths (separated by "`;`" characters) in which to search for the class' source file;
- a command template from which to build the final command line. Some escape sequences may be used, that will be replaced by their value shown in grayed fields of the dialog (see following table for a complete reference). A very useful template is therefore "%p%f" which gets automatically replaced by the absolute path name of a matching source file, if any.

Escape sequence	Replacement
%%	Single "%" character
%c	Fully-qualified class name
%f	Source file (without path)
%l	Line of method definition in source file (`-1` if undefined)
%p	Absolute name of first found path containing source file. It is guaranteed that this name ends with the system-dependant path separator character ("\" on Windows, "/" on Linux).

2.4 Definition files

To be shown in the *New View* dialog, filters must have been introduced in a *filter definition file*. These are XML files describing the filters. Filter definitions may be grouped inside directories, which may in turn hold sub-directories. A filter definition is made of the following fields (only the first two are mandatory):

- its name, a human-readable string uniquely identifying it inside its directory;
- the fully qualified name of its implementation class;
- its description, version number, author name, which are free texts giving information to the user about the filter;
- the names of some other filters strictly required by it. These filters will be automatically activated with the filter;

- order constraints between the filter and any others.

The software first tries to open file `etc/filters/default.xml`, then opens any other file in the same directory (`etc/filters`) which name ends with `.xml`. User-defined filters should therefore be introduced in new definition files.

3 Filters reference

Without any activated filters, trace files are presented as shown in Figure 3-1, i.e. method calls are shown in the order they occur and no special display or processing is done to nodes.

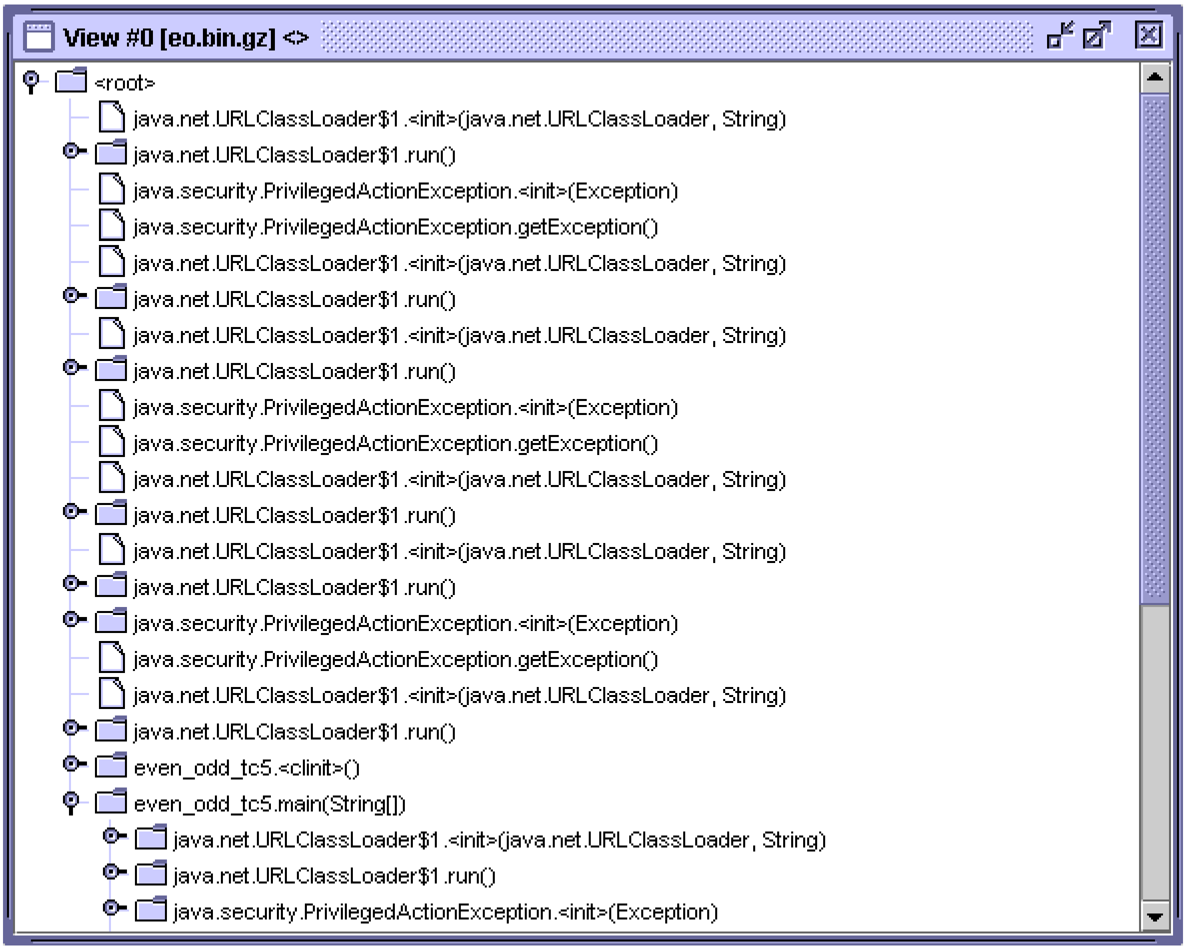

Figure 3-1 – Default trace file presentation with an empty set of filters

All customizable filters can be configured both in the file `etc/settings/default.xml` and through the graphical user interface.

3.1 Common directory

This package contains filters that might be useful for any language targeted to the JVM. These include basic processing such as displaying and highlighting time usage, sorting methods and grouping identical methods together.

3.1.1 Displaying time usage

The filter `Display-total-time` displays for every method call the total time spent in it. Time is by default printed out in nanoseconds, but the user can customize the Boolean parameter `use-best-unit` to render the time in the unit that fits it best (for example, 1000 ns are displayed as 1 µs).

The other display filter is `Display-total-time-percent`, which actually associates a *ratio* with every processed method call, this ratio being equal to the total time spent in the method, divided by the program's total execution time. Using the Boolean parameter `relative-to-parent`, the user can state that ratios should be equal to total time spent in methods divided by their respective caller total time. The ratio computed by this filter can be later reused by filter `Highlight-hot-spots` (see next paragraph).

The result of both time display filters in effect is shown in Figure 3-2.

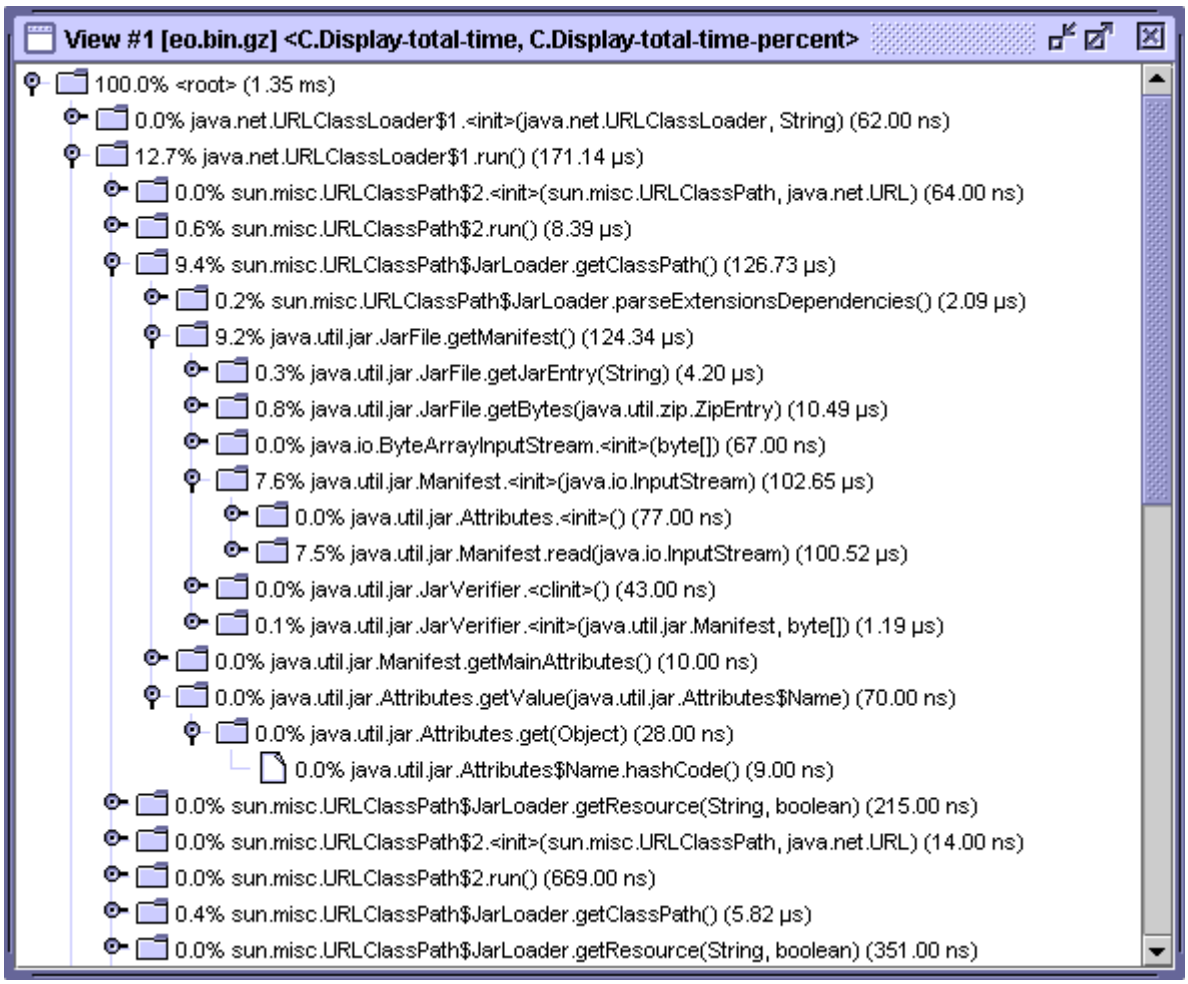

Figure 3-2 – Time display filters

3.1.2 Highlighting time usage

The filter `Highlight-hot-spots` uses ratios computed by `Display-total-time-percent` (which it requires and must follow) to color method calls. The user can customize (through `color-distribution` parameter) what color should be associated to every ratio. This parameter is a list of RGB opaque color values, separated by "`;`" characters, expressed as integers – see the documentation for `java.awt.Color` class for more information [3].

The graphical user interface to this filter customization (see Figure 2-4) shows a bar with all configured colors. Double clicking on this bar with the mouse automatically replaces intermediate colors between the first and the last one by a color gradient.

The Figure 3-3 shows a program trace highlighted using different levels depending on time cost. Notice that filter `Display-total-time-percent` was automatically activated to satisfy `Highlight-hot-spots` requirement constraint.

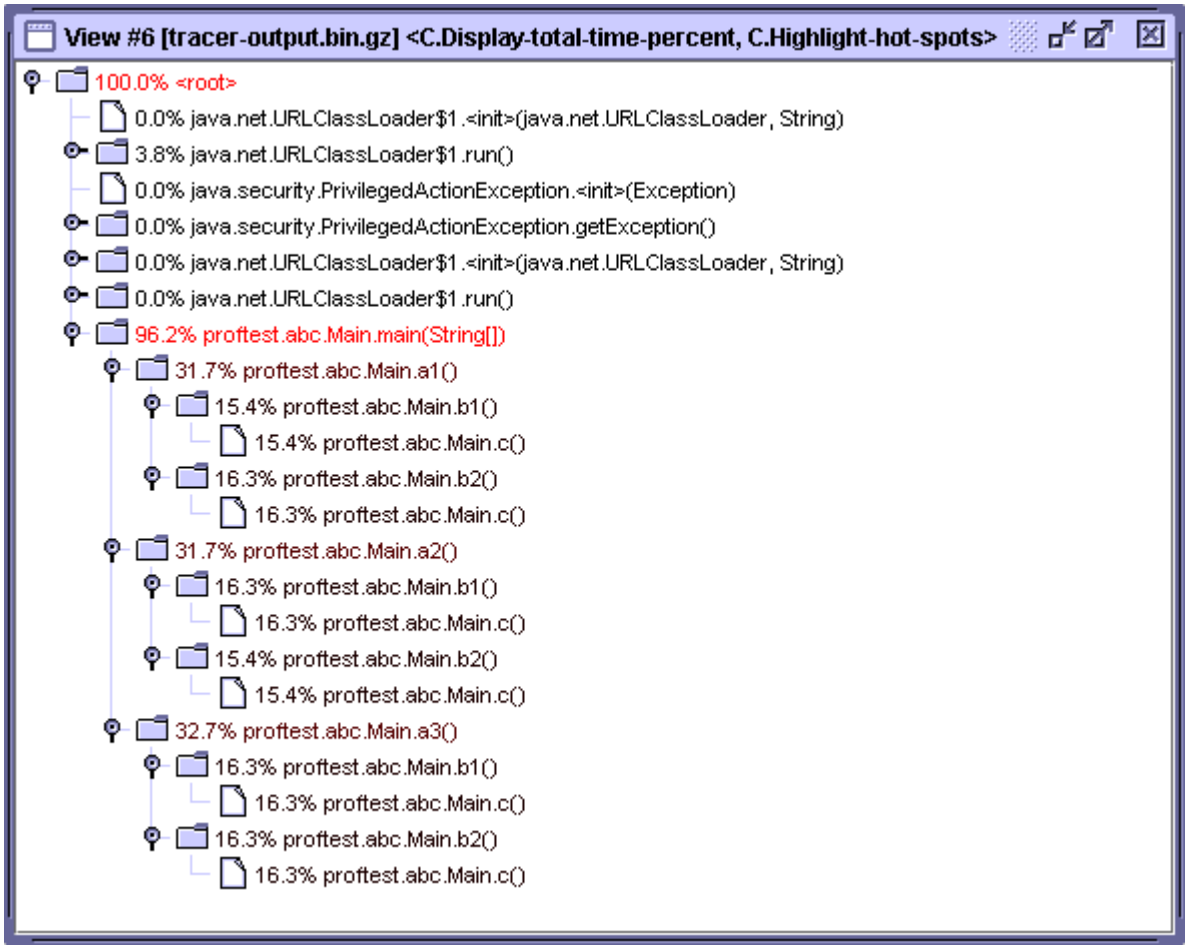

Figure 3-3 – Highlighting important spots

3.1.3 Sorting method calls

Method calls can be sorted respectively by name or by total time spent in them, using respectively `Sort-by-text` and `Sort-by-total-time` filters. Both filters offer a Boolean customization parameter named `descending-sort-order` that allows sorting in reverse order (default is ascending order).

Both sorts are guaranteed to be *stable*: equal elements will not be reordered as a result of the sort. Hence, it makes sense to apply the filters one after the other. First applied filter will set least significant order, while the last applied filter sets main (preeminent) sort order.

Figure 3-4 shows method calls sorted by their text first, then by their time cost. `Display-Total-Time` filter, which is not strictly required, was added to help comprehension.

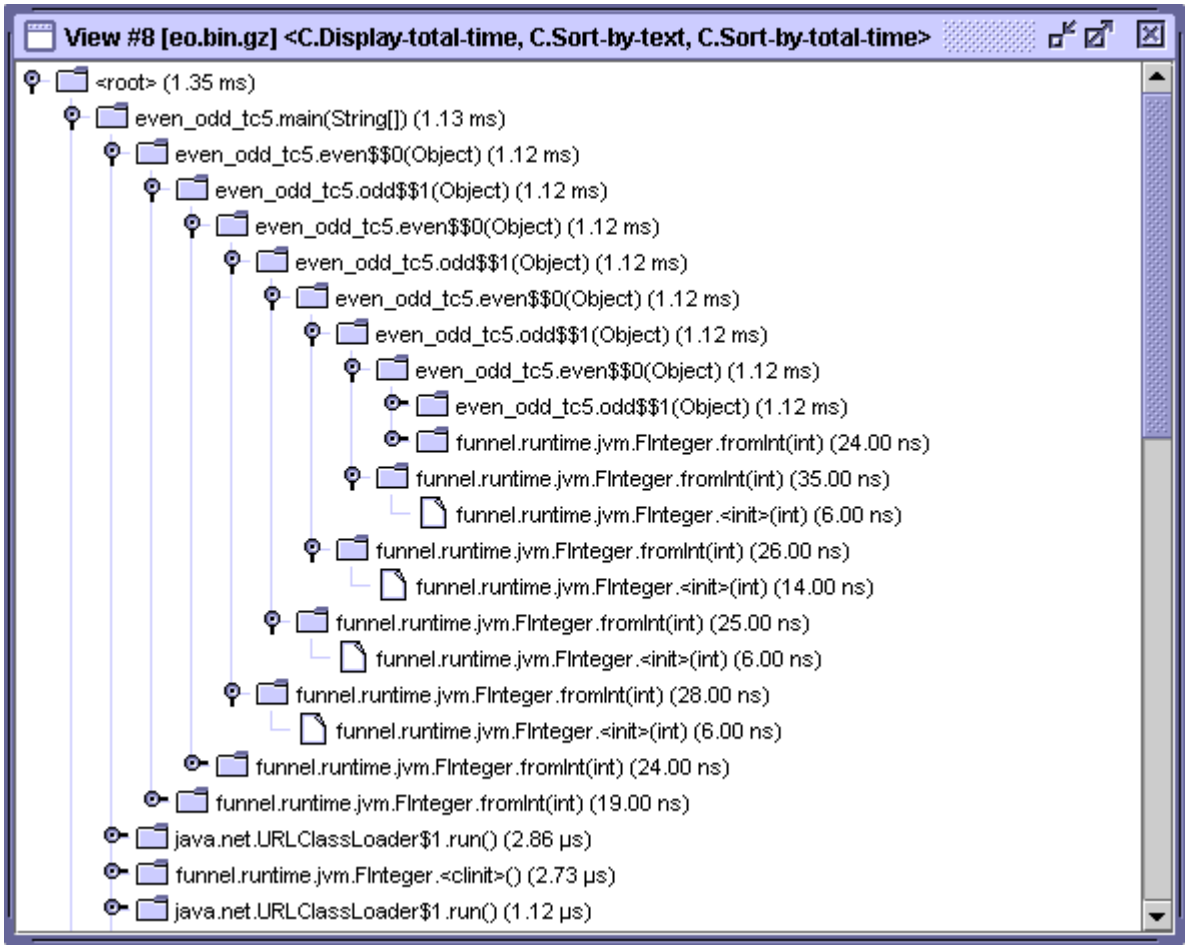

Figure 3-4 – Method calls sorted by time usage and by name

3.1.4 Grouping identical methods together

The filter `AccumulateMethods` completely changes the structure of the tree, by replacing all identical nodes at a given level by a new abstract node that cumulates all time costs and children nodes of the merged nodes.

As this filter replaces nodes, it must be applied *before* any filter that changes nodes' aspect, otherwise their changes would be lost. Consequently, it must precede both `Display-total-time` and `Display-total-time-percent`.

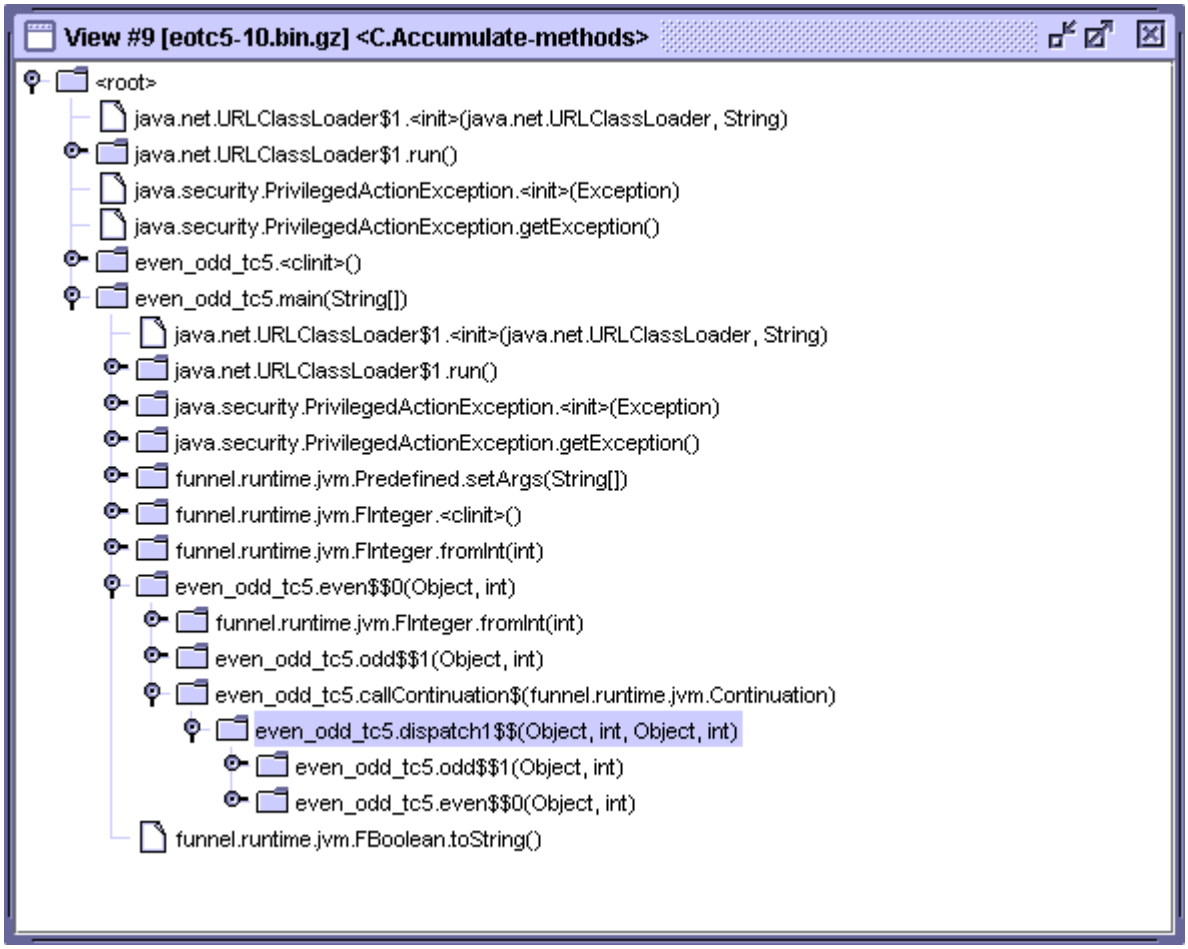

Figure 3-5 – Accumulated method calls

Figure 3-5 shows the result of accumulating identical method calls. The selected line represents two distinct calls to the `dispatch1$$()` method, one of which in turn calls `odd$$1()` method, whereas the other calls the `even$$0()` method. Both get replaced by a single virtual method call that inherits all callee methods.

Note that the view obtained with this filter is that same as the one that some other products like *hprof* (part of [3]) or *OptimizeIt* [6] produce.

3.2 Funnel directory

When profiling a Funnel program on the JVM, the user should be able to see it at the Funnel source code level. This means he should see only Funnel functions he explicitly calls and Funnel modules he uses.

This directory contains various advanced filters that aim at hiding Java encoding of Funnel concepts to the user. These include hiding runtime environments, removing *dispatch* methods widely used in Funnel release 8, completely hiding *tail-continuation elimination* and removing some identifiers that are added when converting function names to method names.

3.2.1 Hiding runtime environments

Two main runtime environments must be hidden to the user, specifically Funnel runtime (all classes located in `funnel.runtime` package) and Java runtime (`java` and `sun` packages). Hiding a runtime environment to the user means hiding all method calls to these classes, but not the possible calls from these methods both to user functions. Hence, filters must be very careful in hiding such information and cannot simply destroy complete sub-trees. For more details on removing nodes from a tree, see section 4.3.3.

The work of removing runtime was logically separated in the two filters `Remove-Funnel-runtime` and `Remove-Java-runtime`. Figure 3-6 shows the effect on a *Funnel* program of hiding method calls to both runtime environements.

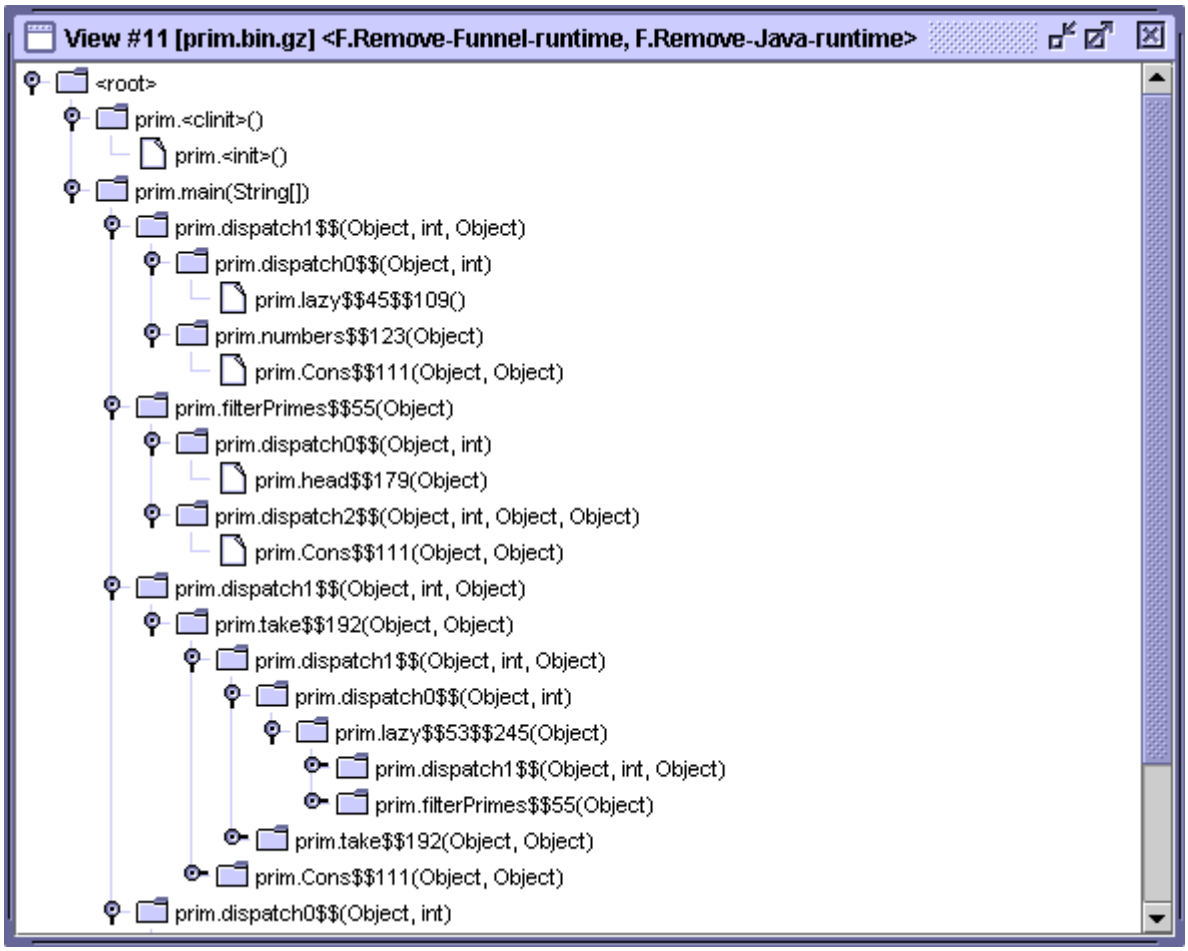

Figure 3-6 – Hiding all runtime environments

3.2.2 Removing dispatch methods

Release 8 of the Funnel compiler makes heavy use of `dispatch()` methods to apply functions that are unknown ate compile time. These method calls must be removed, but the call to the user function they contain must be preserved. In addition, the time that was initially spent in the removed node must be inherited by its parent node.

This precise functionality is handled by `Remove-dispatches` filter, which removes methods (see 4.3.3 for implementation details) with name starting with "`dispatch`".

Figure 3-7 shows the same program as in Figure 3-6 with the additional application of `Remove-dispatches` filter. The result is now structurally very close to a trace execution in the target language.

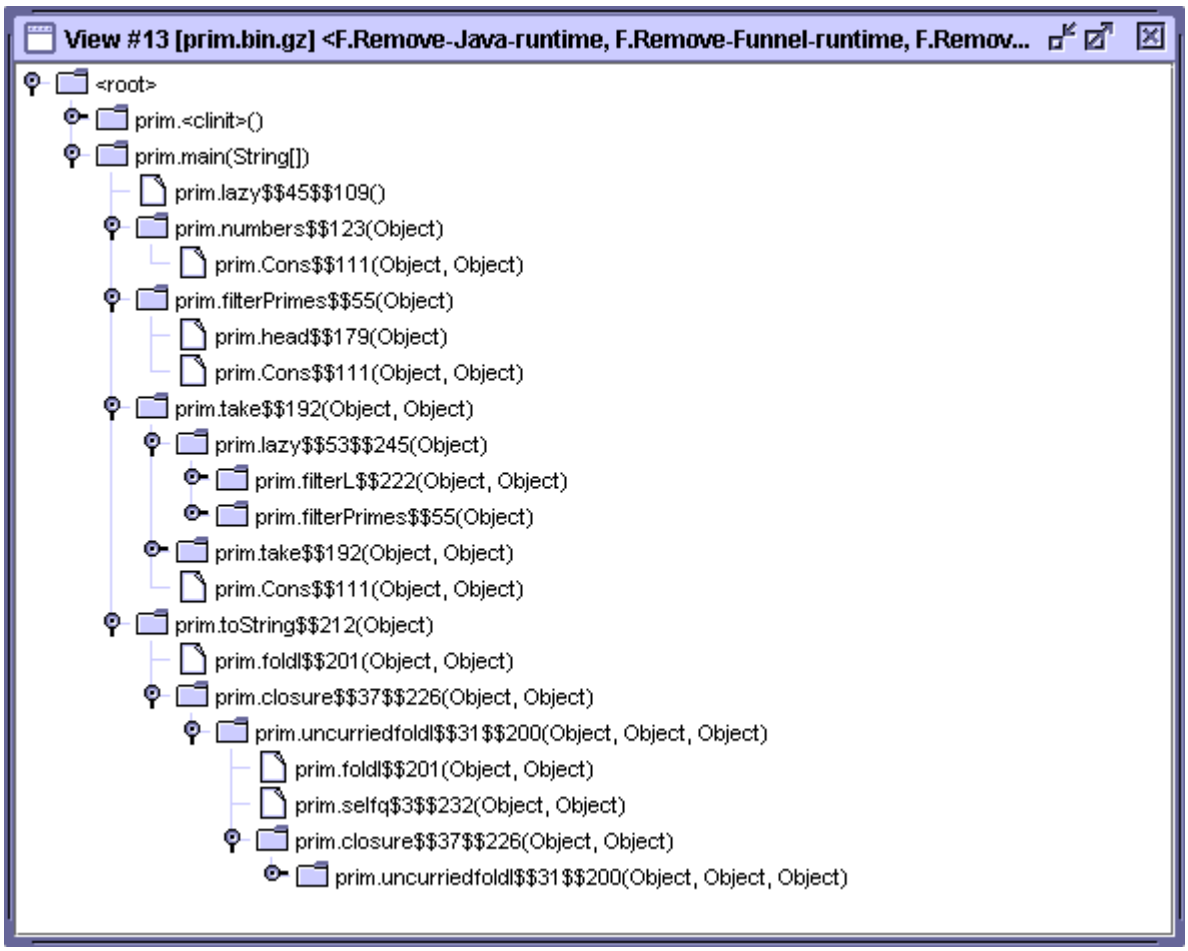

Figure 3-7 – Hiding dispatch methods

3.2.3 Hiding tail-call elimination

Tail call elimination is an optimization used in current and forthcoming *Funnel* compilers which guarantees that loops implemented using recursion execute in constant space [7]. It results in deep changes in the trace tree structure. Figure 3-8 shows reciprocally recursive calls between *Funnel* functions named `even()` and `odd()`, in a program compiled with tail call elimination.

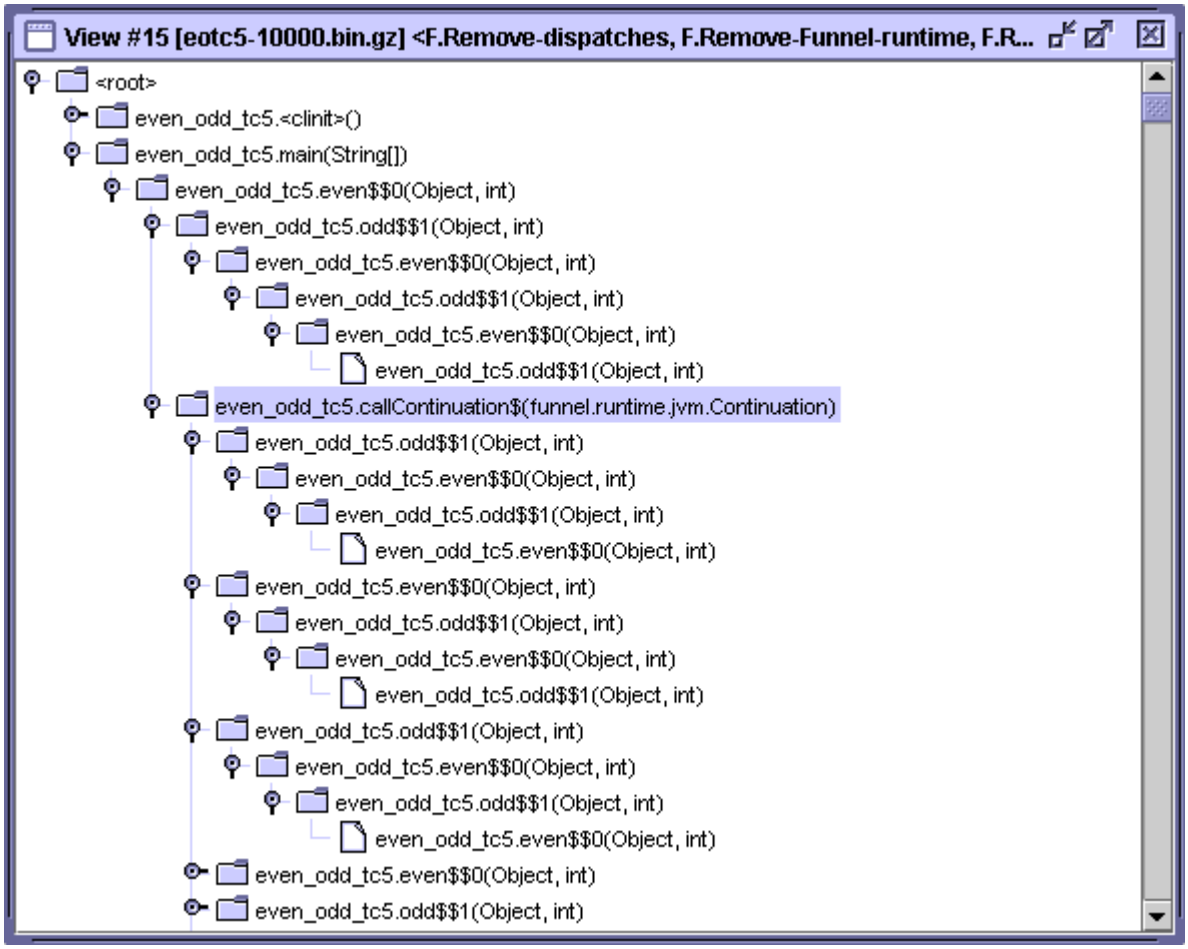

Figure 3-8 – Tail call elimination example

Filter `Hide-tail-call-eliminations` suppresses the call to method `callContinuation()` (highlighted in Figure 3-8), which was introduced by the compiler, and rebuilds the original tree by expanding flattened sub-trees. For more details on this filter, refer to [5].

This filter requires another one to work correctly, since it creates invalid duplicate method calls due to the way the compiler handles `callContinuation()` (notice the duplicate call to `odd()` around it in Figure 3-8). Removal of such duplicates is handled by another filter `Remove-duplicate-in-TCE` which gets automatically activated with `Hide-tail-call-eliminations` (if active alone, this second filter does nothing).

The result of applying both filters is shown in Figure 3-9. Tail call elimination is completely hidden to the user.

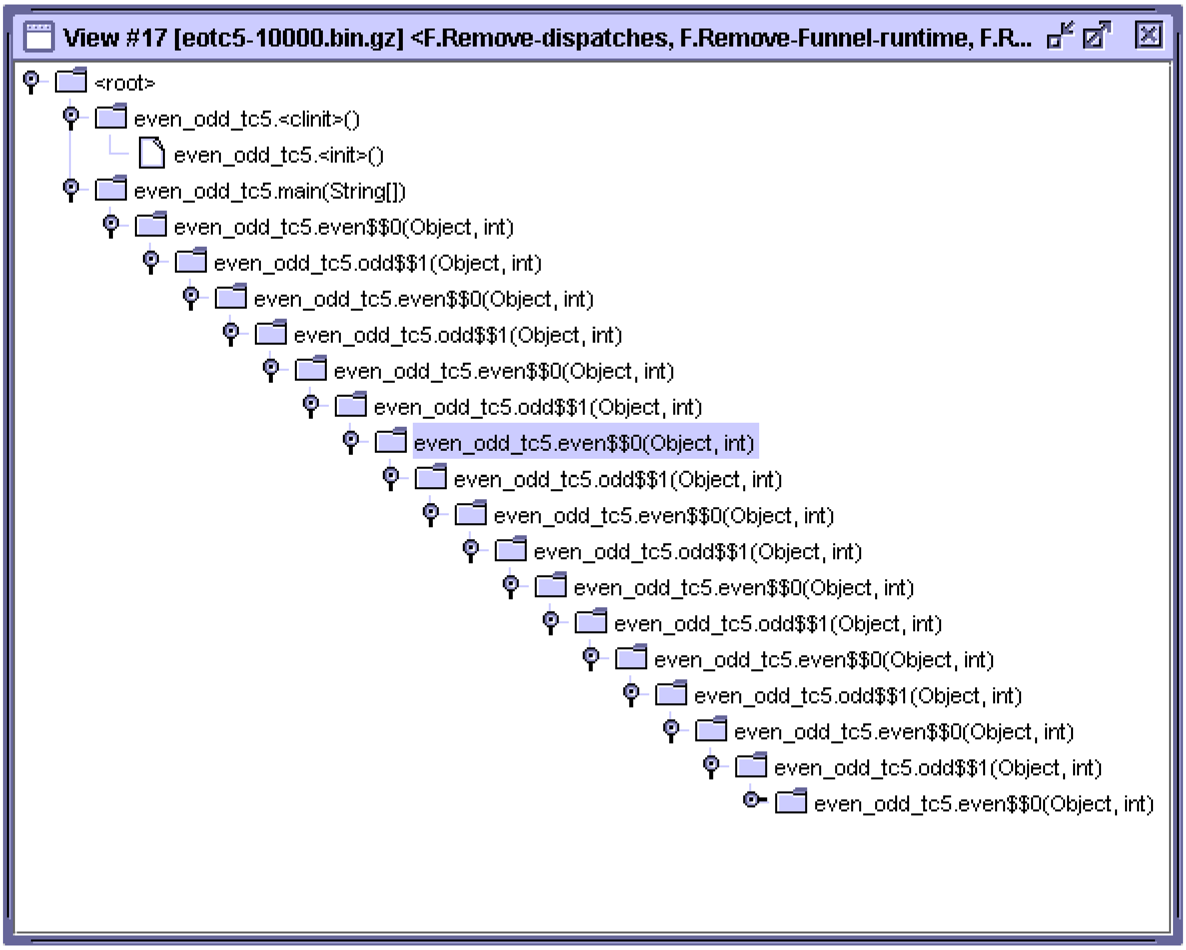

Figure 3-9 – Hidden tail call elimination

3.3 Renaming methods

Two additional filters are provided, namely `Unmangle-Funnel-names` and `Rename-Funnel-concepts`, that rename methods to produce a result which is closer to Funnel source code. The first one removes identifiers added when converting Funnel functions to Java methods (for example, `even_odd_tc5.even$$0()` gets replaced by `even()`). Second filter renames method calls that match a specific Funnel concept with its name. Currently such concepts include anonymous functions application and lazy record evaluation.

Figure 3-10 shows the result of applying both renaming filters, together with all other Funnel and display filters. This can be considered as a complete overview of most filters functionalities.

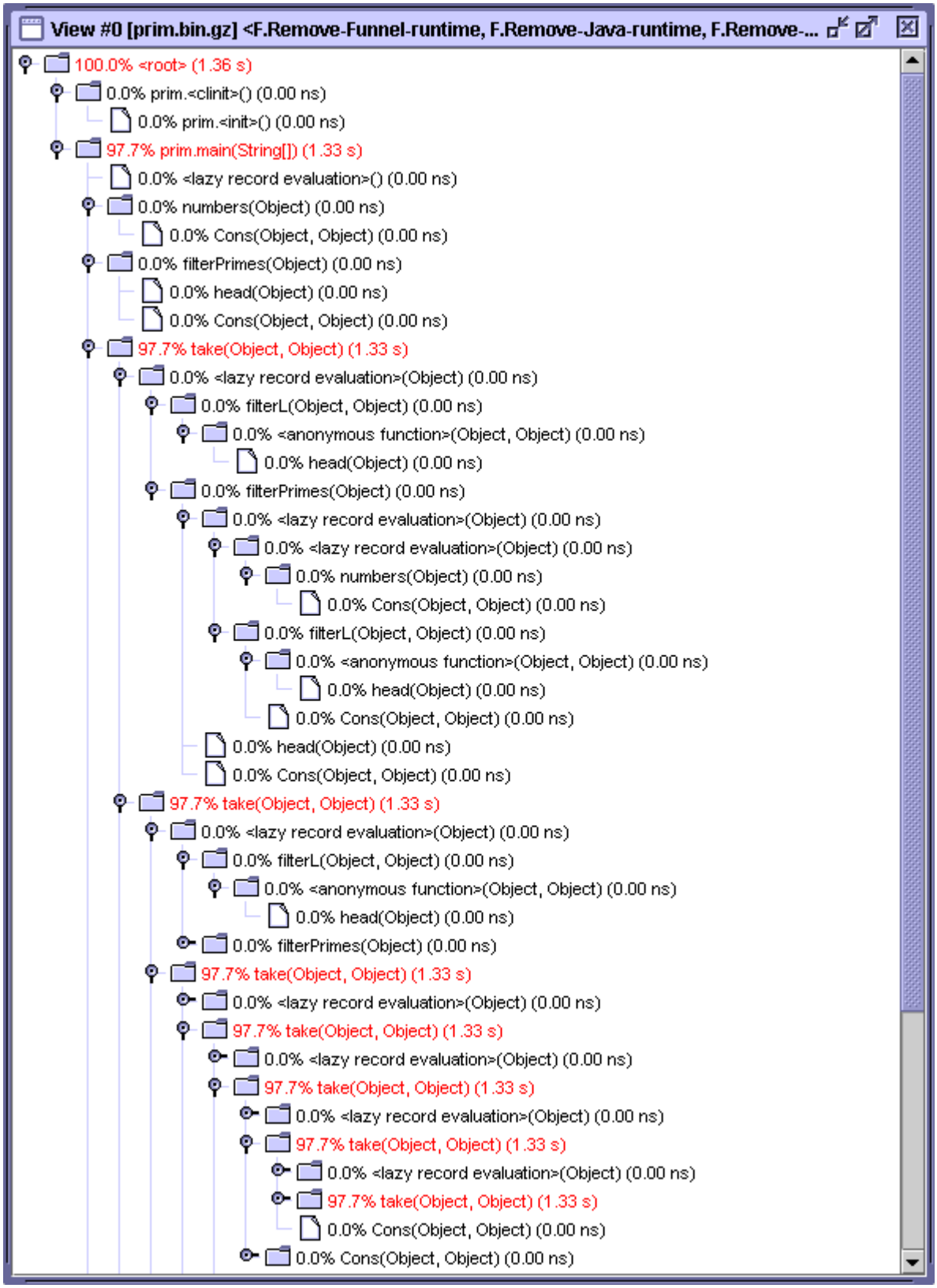

Figure 3-10 – Resulting profile view with all Funnel and many display filters

4 Application Programming Interface

4.1 Design patterns

In object-oriented programming development, Model-View-Presenter (MVP) is the name of a methodology or *design pattern* for successfully and efficiently relating the user interface to underlying data models. EJP Presenter uses this system to handle multiple different views of the same data.

- The model is a constant call tree built from a trace file's content.
- Views are graphical windows that the user can open to show the model in a specific way.
- Presenters are responsible for generating views from the model. They are implemented as a chosen set of transformations to apply to a copy of the model.

4.2 Nodes

When loading a trace file, the parser builds up the *model* tree, whose nodes are instances of the `MethodTimeNode` class. This class implements a *growable-only* tree, i.e. children nodes may only be added to it, and never removed. When the loading is terminated, the built tree is meant to be unchangeable, in order to ensure a read-only model.

Once a trace file is successfully loaded, the user might want to create *views* of the data, using a specific set of filters. Since filters are intended to transform the tree, they cannot directly work on the model tree, but rather have to be applied to another structure that mirrors it.

Furthermore, the nodes of new views must ensure *laziness*, i.e. their children must not be computed before the nodes are opened by the user. So the nodes used for building new views are instances of another class `LazyNode`. Most nodes simply represent a single `MethodTimeNode` instance; these nodes encapsulate it as a private field. More complex nodes (that do not relate to exactly one node of the model) might use their own way to retrieve information such as called method and time cost.

A `LazyNode` must provide methods for:

- accessing (with *get*, *add* and *remove* operations) its children as `LazyNode` instances, while ensuring that they are being actually computed only on first access (laziness property);
- handling its own and total time;
- changing its visual aspect (text, color and tool tip text).

4.3 Filters

A filter is implemented as a class extending `AbstractFilterImpl`.

When used, a filter has to update the children of a given node by applying any required processing to them. The following method is called on the filter with the parent node being processed and its current list of children. Note that this method should not change the parent node in any way since it already has been displayed when the method gets called.

```
public abstract void applyTo(LazyNode aLazyNode, Vector aChildren);
```

Some filters also have a special processing for the *root* node (displayed as "<root>" in the tree), which by definition would otherwise never be handled since it is not a node's child. The processing of the following method is allowed to change the root node, but not to access its children.

```
public void applyToRoot(LazyNode aRoot);
```

When processing child nodes, filters might need to know their own children nodes. Filters are therefore allowed to access a child's children, whose computation results in *recursive filter application*. To ensure correctness of these operations, filters must be:

- *reentrant*: a filter might be recursively applied while an application is in progress at a higher level of the tree;
- *idempotent*: the result of applying a filter once is the same as applying it *n* times.

Different types of general filters are supplied:

- **sorter** filters, which order the children in a specific manner;
- filters that **render nodes**, i.e. that only modify their visual aspect;
- filters able to **remove children nodes** from the tree.

4.3.1 Ordering children nodes

The class `AbstractSorter` provides a convenient way to implement filters that sort children nodes. Such filters only have to implement the following comparator method in order to return a comparison that will be used to sort nodes in *ascending* order.

```
public abstract int compare(LazyNode aNode1, LazyNode aNode2);
```

This method should return respectively a negative number, zero or a positive number when the first node should appear before, next to or after the second node.

4.3.2 Rendering nodes

The different aspects of a node that can be customized by a filter are:

- its label (which by default is the method signature), using `setText()` and `getText()`;
- its color, handled with `setColor()` and `getColor()` methods;
- some additional free text preceding and following the label, using respectively `addPrefix()` and `addSuffix()` methods.

Filters that render nodes should be implemented by extending class `AbstractNodeRenderer`.

4.3.3 Removing children nodes

Some filters need to *hide* specific information to the user in order to allow him to concentrate on important points of his program. This feature was mainly designed to hide Java encodings of Funnel concepts to the user. Removing a node from the tree is not the same process as removing a *sub-tree* from it. Here, removing a specific node means removing only that node from its parent, while preserving the node's children and moving them one level up in the hierarchy. This way, the temporary orphan nodes are immediately inherited by their grand-parent.

The time that was originally spent in the removed node's body should be transferred to its caller, i.e. to its parent in the tree. This is to reflect the fact that this time cost was originated by the caller method and should therefore be counted as its own time.

The class `AbstractNodeRemover` implements an abstract filter able to remove children nodes. A custom filter only has to implement a single method that decides whether a specific node should be removed from its parent.

```
protected abstract boolean isToRemove(LazyNode aChild, LazyNode
aParent);
```

4.4 Parameters

Filters may have any number of parameters that can be customized by the user at the time the filters are activated. These parameters allow greater reusability of filter implementations. A filter uses its `addParameter()` method to register any parameter it will use. Registered parameters will automatically provide graphical customization panels that the user can view and change. These panels offer some more interesting automatic features to filter developers like the saving and loading to an XML file of user's settings.

A parameter type must extend class `AbstractParameter` and provide:

- methods to access (*get* and *set*) the parameter's *value*, as an `Object` (used for *Cancel* feature of the dialogs) or as a `String` (used for persistent XML storage);
- a `setReadOnly()` method, which will be called only once to indicate that the parameter's value may no longer be changed.

Parameters instances automatically build the graphical panel that will be used to display and change their value. A parameter computes its value from possibly many *settings*, each of which is represented on a single *configuration line*. Such a line is composed of a title label, and one or two graphical components (instances of Swing `JComponent` class) that will be placed next to each. All configuration lines are then grouped in a graphical box that represents the parameter. The following method allows a parameter to add configuration lines to its panel (second `JComponent` argument can be left to `null` to only display one component).

```
protected final void addLine(String aTitle, String aToolTipText,
JComponent aComponent1, JComponent aComponent2);
```

For an example of parameter graphical representation, refer to Figure 2-4 on page 10, which shows a filter defining one single parameter that is made of two configuration lines.

Some simple parameters are provided with the API:

- a parameter that represents a `Boolean` value and renders it as a check box;
- a parameter representing a *color distribution*, i.e. a mapping [0,1] → {set of colors}, that associates a color to every real number between 0 and 1. This parameter allows to divide the range by a user-chosen divider, and to assign a color to every sub-range.

Filters register their parameters using the following inherited method. The order in which parameters are added is significant only for their graphical representation, parameters been displayed top down in the order in which they are defined.

```
protected final void addParameter(AbstractParameter aParameter);
```

When creating `AbstractParameter` instances, a filter must associate them names that uniquely identify them inside the filter. Graphical title and tool tip text can also be provided.

```
public AbstractParameter(String aName, String aTitle, String
aToolTipText)
```

4.5 Logging

The API provides a convenient way to log status messages from any class. These messages will be displayed to the user in the *Log* window, using a color code indicating the severity. The class `CustomLogger` uses the Java Logging API and offers a single instance (through its `INSTANCE` field) of the `java.util.logging.Logger` class configured to automatically render messages to the window.

```
public static final Logger INSTANCE;
```

For more information on the `Logger` class, please refer to JDK 1.4 API [3].

4.6 Tracer API

Java package `ejp.tracer` contains one single class `TracerAPI` that allows a profiled program to interact with EJP Tracer. Provided methods allow to know if EJP Tracer encountered any error during its initialization (if not, then it is currently running), and to temporarily enable / disable it.

```
public static Throwable getInitializationError();
public static boolean enableTracing();
public static boolean disableTracing();
```

4.7 XML files

The format of XML can be defined by Document Type Definition (DTD) files. EJP Presenter uses this feature to ensure that processed configuration files are well-formed.

4.7.1 Filters definitions

The format of filters definition files is specified by the file `etc/filters/filters.dtd`. When specifying relationships with other filters (`requires`, `must-follow` and `must-precede` attributes), relative (without dots "`.`") filter names can be used to refer to filters in the same directory, otherwise fully qualified filter names must be used.

```
<!ELEMENT filters (directory | filter)+>

<!ELEMENT directory (directory | filter)+>
<!ATTLIST directory
  name NMTOKEN #REQUIRED
  >

<!ELEMENT filter EMPTY>
<!ATTLIST filter
  name NMTOKEN #REQUIRED
  class-name CDATA #REQUIRED
  description CDATA #IMPLIED
  version CDATA #IMPLIED
```

```
  author CDATA #IMPLIED
  requires CDATA #IMPLIED
  must-follow CDATA #IMPLIED
  must-precede CDATA #IMPLIED
  >
```

For example, the following XML document defines filter `Common.Highlight-hot-spots` and its relationships with two other filters (not shown) in the same directory.

```
<!DOCTYPE filters SYSTEM "filters.dtd">

<filters>

  <directory
    name="Common"
    >

    <filter
      name="Highlight-hot-spots"
      class-name="ejp.presenter.api.filters.HighlightHotSpots"
      description="Highlights nodes depending on their time percent"
      requires="Display-total-time-percent"
      must-follow="Accumulate-methods Display-total-time-percent"
      />
  </directory>

</filters>
```

4.7.2 Application default settings

Application default settings include default values for *Run Program* dialog and for filters parameters. The format of these settings is defined in file `etc/settings/settings.dtd`.

```
<!ELEMENT settings
  (filter-customization*, run-program-dialog?, filter-customization*)>

<!ELEMENT run-program-dialog EMPTY>
<!ATTLIST run-program-dialog
  source-paths CDATA #IMPLIED
  command-template CDATA #IMPLIED
  >

<!ELEMENT filter-customization (parameter+)>
<!ATTLIST filter-customization
  full-name ID #REQUIRED
  >

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
  name CDATA #REQUIRED
  value CDATA #REQUIRED
  >
```

The following example file declares default values for *Run Program* dialog and `relative-to-parent` parameter of `Common.Display-total-time-percent` filter.

```
<!DOCTYPE settings SYSTEM "settings.dtd">

<settings>

  <run-program-dialog
    source-paths="c:\prog\jdk\src;c:\java"
    command-template="emacs +%l %p%f"
    />

  <filter-customization
    full-name="Common.Display-total-time-percent"
    >

    <parameter
      name="relative-to-parent"
      value="false"
      />

  </filter-customization>

</settings>
```

5 References

[1] ODERSKY, M., *An Overview of Functional Nets*. Lecture Notes, APPSEM Summer School, September 2000.

[2] *The zlib Library*. http://www.gzip.org/zlib

[3] *Java 2 SDK Standard Edition version 1.4*. Sun Microsystems, 2002. http://java.sun.com

[4] ZENGER, M., & SCHINZ, M., *Funnel compiler release 8*. http://lampwww.epfl.ch/funnel

[5] VAUCLAIR, S., *Extensible Java Profiler*. Diploma Thesis, EPFL, 2002.

[6] *Optimize It*. VMGEAR. http://www.vmgear.com

[7] SCHINZ, M., & ODERSKY, M., *Tail call elimination on the Java Virtual Machine*. Electronic Notes in Theoretical Computer Science, 59, No. 1, 2001.