Implementing a trace component for Carbide development framework

Esa Karvanen

University of Tampere
Department of Computer Sciences
Computer Science
M.Sc. thesis
Supervisor: Jyrki Nummenmaa
June 2007

University of Tampere
Department of Computer Sciences
Computer Science
Esa Karvanen: Implementing a trace component for Carbide development framework
M.Sc. thesis, 47 pages
June 2007

This thesis introduces a trace component named TraceViewer which is a plug-in for Carbide development framework. TraceViewer is able to receive, manipulate and view traces received from a smartphone. The thesis presents some background information about Carbide.c++ framework. A tracing concept, Open System Trace, used in TraceViewer is also introduced. TraceViewer is described by going through the requirements, architecture and design phases which will give a good overview about the software.

Key words and terms: TraceViewer, Carbide, Trace, Debug, S60, smartphone, Open System Trace.

Contents

Preface

Tracing is a fast and efficient way of debugging software execution. For Symbian development, Carbide.c++ was used. There was no working and easy solution for tracing inside Carbide.c++ which was the motivation to start this project.

The subject of this thesis, TraceViewer, has been designed and implemented in Nokia Tampere in Product Platforms organisation. I would like to thank Erkki Salonen, my team manager, for giving me the opportunity to write this thesis.

I would also like to thank my whole team at work and my parents for trying to motivate me to finish this thesis. Also, special thanks go to Rami Törmä and Tuomas Vuori for making my studying times such a joy.

Esa Karvanen

11 June 2007, Tampere

1. Introduction

This thesis introduces a component called TraceViewer. TraceViewer is used to receive and manipulate traces from a trace source which in this case is a smartphone. TraceViewer is a plug-in for a development framework called Carbide.c++ which is built above a popular open source software framework called Eclipse.

TraceViewer is part of a bigger tracing concept called Open System Trace which enables tracing from a smartphone to the view of TraceViewer. Using TraceViewer the user can receive traces from a smartphone and use many kinds of trace line manipulations to find relevant information more quickly and easily. The main features consist of, for example, filtering, coloring, counting and searching of traces and investigating variables inside traces.

This thesis starts by introducing Eclipse and Carbide.c++ frameworks and continues by explaining the concept of tracing. In this context also a tracing model, Open System Trace, used in TraceViewer and other related tools will be introduced. The next two chapters will describe the requirements, architecture and design of TraceViewer plug-in. The thesis will not go to source code level of the design, but will give an overview of the component and its structure and how the most problematic design issues were solved. The final chapters are used in evaluation of the TraceViewer, summary and the conclusions about the success of the project.

2. Eclipse and Carbide.c++ frameworks

2.1. About Eclipse

Eclipse is a platform independent open-source software framework originally developed by IBM in 2001. It is now managed by Eclipse Foundation which is a non-profit corporation. The Eclipse framework was claimed to have 2.25 million users worldwide in November 2006 [CNET News, 2006].

Eclipse is mainly known as a Java development environment although in principle the platform itself has no support for it. Eclipse was built to a universal IDE (Integrated Development Environment) where the user can handle project workspaces and build, launch and debug applications. It is designed to be highly extendable with so called plug-ins. The most famous plug-in for Eclipse is probably the JDT (Java Development Tools) which will make Eclipse a Java IDE. JDT, as many other plug-ins, comes with Eclipse SDK as a default which is why many people do not even realize that those are not fixed parts of Eclipse. [Eclipsepedia, 2007]

Eclipse Java Development Tools (JDT) provides many plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java perspective to Eclipse Workbench as well as a number of views, editors, wizards, builders, code merging and refactoring tools. [JDT, 2007] Refactoring tools are one reason why Eclipse is popular. These convenience tools allow the user for example to rename Java elements, move classes and packages, create interfaces from concrete classes, turn nested classes into top-level classes, and extract a method from a section of code. Using these tools is a good way to improve productivity and keep your code more maintainable.

While the Eclipse platform is designed to be an open tools platform, it is architectured so that its components can be used to build any kind of client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform (RCP). The Rich Client Platform consists of the following components:

- Eclipse Runtime providing the support for plug-ins, extension points and extensions.

- Standard Widget Toolset (SWT) which provides a native GUI look and feel to applications by accessing native GUI libraries of the operating system.
- JFace – A UI framework layered on the top of SWT which handles many common UI tasks and provides easy access to SWT features.
- Workbench which is built over Runtime, SWT and JFace and provides a multi-window environment for managing views, editors, perspectives, actions and many more.
- Other plug-ins included in the RCP provide support for XML language, commands and help core content model.

There are also many other Eclipse components that can be used in constructing a RCP application. Quite a few companies are using the opportunity to make a multi-platform application easily without having to build everything from scratch by using Eclipse RCP as a base for the software. One of these applications is Carbide.c++.

2.2. Carbide.c++

Nokia's mobile development tools product set is called Carbide. It is designed to integrate Nokia's wide range of development tools into a common framework. There are three families of tools within Carbide: Carbide.c++ tools for Symbian OS development, Carbide.j for Java development and Carbide.ui for customization.

Carbide.c++ is built on the top of the Eclipse framework and more specifically, on the top of C/C++ Development Tools (CDT) Project. CDT project provides an IDE for C and C++ software development. The function of the Carbide.c++ is to replace Metrowerks CodeWarrior as the primary development environment for Symbian OS.

There are four different versions of Carbide.c++. The express version is the most elementary containing basic tools for non-commercial development. The developer version of Carbide.c++ is targeted to commercial software development which also has some additional features such as the UI Designer for rapid UI creation and application-level on-device debugging. Carbide.c++ Professional includes system-level on-device debugging for complete access to all system threads and memory and Carbide Performance Investigator for performance optimization. Because these tools enable application development on early prototype hardware, Professional is mainly targeted to Symbian OS phone manufacturers, their partners and application/middleware vendors. The

latest Carbide.c++ version, OEM, is targeted to device creation users with features such as Stop-mode debug (JTAG support) and Device creation features (TBA). Carbide.c++ OEM v1.2 is available as 2007. [Forum Nokia, 2007]

3. The concept of tracing

3.1. Basics of tracing

Tracing is defined as monitoring program execution and interesting variables in run-time. The difference between debugging and tracing is that in debugging the user normally stops the program execution to investigate, for example, the value of a variable when in tracing, program execution is not stopped but the value of the variable is, for example, printed to the screen. Tracing is often used to follow the application run path by instrumenting traces to the start and end of every function or in every *if* / *else* / *switch* case. This way the user can easily see where the execution ceased and can then instrument more traces to that function to check variable values or the exact code line that caused the error.

The most famous trace in software development is "Hello world" which is the output from the first software for many programmers. The trace indicates that the software functioned properly. The principle is the same also for bigger applications. The developer writes traces to the source code, compiles, executes and analyses the output to ensure that the application functioned properly [MSDN, 2007]. When tracing, iteration is often needed to find the exact problem that is causing the malfunction of the application. When starting the debugging of the software by using traces, traces are often added to start and end of the functions the developer is interested in. After the first run and bug occurrence, developer should know the exact function where the execution ceased or went wrong. Then it is a time to add more traces into this specific function to fully see what is going on there. Usually this means outputting variable values. In the next run, the developer should be able to pinpoint the exact place where something went wrong. If this is not the case, more traces are added to other methods and places until the bug is found.

Traces used in software tracing are normally quite unique because they need to tell the developer where the program execution is going and the values of interesting variables. Also, because the traces are mainly meant for the developer, traces are very rarely localized. Some traces are used like assertions, they should be never seen. The developer can, for example, write a trace inside a null check block and they see from the output if the null check fails. If this happens, traces outputted before the null check trace are used to pinpoint the execution path leading to the false situation.

In most cases traces are printed inside some console on the computer screen while implementing an application to computers. Often in the case of a smartphone, the screen is very small and not very good at showing possibly large amounts of data. Another very commonly used option is to forward traces directly to a file which can then be read later from another place with a bigger screen.

Many programs can write some kind of log file, for example, by using some command line argument when starting the program. These log files can be then analysed by programmers, system administrators or technical support to diagnose problems with the software. Writing a log file can then also be called tracing as long as it is not clearly done as an *event logging*. Event logging normally means logging of very high level information when tracing logs low level information. Writing a log file is often used on debugging server side applications. Servers are supposed to be running for months without interruption and therefore normal debugging means can not be used. Often the defects on server side applications happen in same specific combination of actions which is very hard to find by debugging. Using tracing into a log file, the developer can check latest actions from the log file after the server has crashed or ceased functioning properly.

Tracing has a performance impact on the application execution. An application outputting multiple string variable traces will suffer from slowed down performance because processing strings is very CPU intensive. Depending on the time criticality of the application, the performance impact might be ignored. To avoid losing a lot of performance, traces can be decoded as integers which are very fast to process. Each trace will be mapped as an integer value and the mapping is saved to a decode file. The phone then only outputs integers and the receiving end of the tracing system uses the generated decode file to map those integers back to traces.

3.2. Open System Trace

Open System Trace (OST) is a definition for a tracing concept used in TraceViewer and other components related to it. OST defines what kind of traces can be used in the source code of an application, what kind of a decode file is created from the traces and how TraceViewer (or some other application) uses the decode information when decoding the traces received.

The main ideas behind the Open System Trace are to minimize the performance impact of tracing by using decode files and to enable run-time activation of

traces. Each trace belongs to some component and to some group. Every unique trace inside one group has a unique trace ID. This means that every trace has a component ID, group ID and a trace ID which will all be sent from the phone to the receiving end, in this case to the TraceViewer. Using these three ID numbers and the decode file generated during the instrumentation of the traces into the source code, TraceViewer can display traces as the user wanted with minimal performance impact. The reason why we do not only use trace IDs but also component and group IDs is because it enables the possibility to activate or deactivate traces from a specific component and/or group. For example, the user has instrumented traces using two different groups, "debug" and "normal". Activating only traces from the group "normal" he gets, for example, only function entry and exit traces. Because the traces from the group "debug" are not activated, they will not be sent out of the phone. If there is a problem, the user can also activate the "debug" group and get traces printing out variable values and other debug information. The good thing is that the user does not have to make changes to the source code but only activate traces he is interested in. Traces in the source code which are not activated will cause virtually no performance impact because they are dropped immediately in the phone by the Trace handler.

Because of the decoding and activation, each component ID must really represent only one component. This is why OST will use Symbian UIDs as a component ID. Symbian UIDs are only shared by Symbian and therefore every component has a different UID. While testing and at early development stages users use UIDs from the development range which is:

- **0x01000000 to 0x0FFFFFFF** [Symbian, 2006].

Care must be taken to avoid clashes with other components. If two components have the same UID it may stop a program from loading correctly and typically leading to *Not Found* errors.

When outputting variable values, component, group and trace IDs are not enough. Considering a case where the user wants to output an integer variable value in a loop that has 10000 iterations. The idea is to get 10000 traces each outputting a trace such as "Integer value: n" where n varies from 1 to 10000. If this was done so that each trace is unique, it would take 10000 trace IDs and 10000 decode information lines to the decode file. The number of iterations could also be, say, 10 million. This is why all variables are also outputted as

binary from the phone to the TraceViewer. The decode file specifies that a trace with specified component, group and trace IDs will also have more data after the trace ID. In this case, the decode file could specify that after the trace ID, there are 32 bits of data consisting of an unsigned integer value. This way, TraceViewer can read the integer value and attach it to the trace. Only one trace is then needed to the decode file to output the 10000 (or 10 million) traces.

The Abstract Open System Trace architecture can be seen from Figure 1. The process starts so that the developer uses the Trace Builder tool (described in chapter 3.3) to instrument traces into component source code. In the process, Trace Builder will generate a Trace Definition file. After the application has been rebuilt and installed to the phone, the tester will have to activate the traces by using a Trace Activator. The Trace Activator will get the activation information from the same Trace Definition file that is used to decode traces. The Trace Activator sends the activation information defined by the tester to the Trace Handler located in the S60 smartphone through some supported connection. When the instrumented component is run in the phone, the traces are sent to the Trace Handler. The Trace Handler will delegate the traces marked as activated to TraceViewer. TraceViewer sends undecoded traces to the Trace Decoder which will use the previously generated Trace Definition file to decode traces. Decoded traces are then sent back to the TraceViewer where the tester can read them as the developer planned. The tester can then activate or deactivate more traces using the Trace Activator.

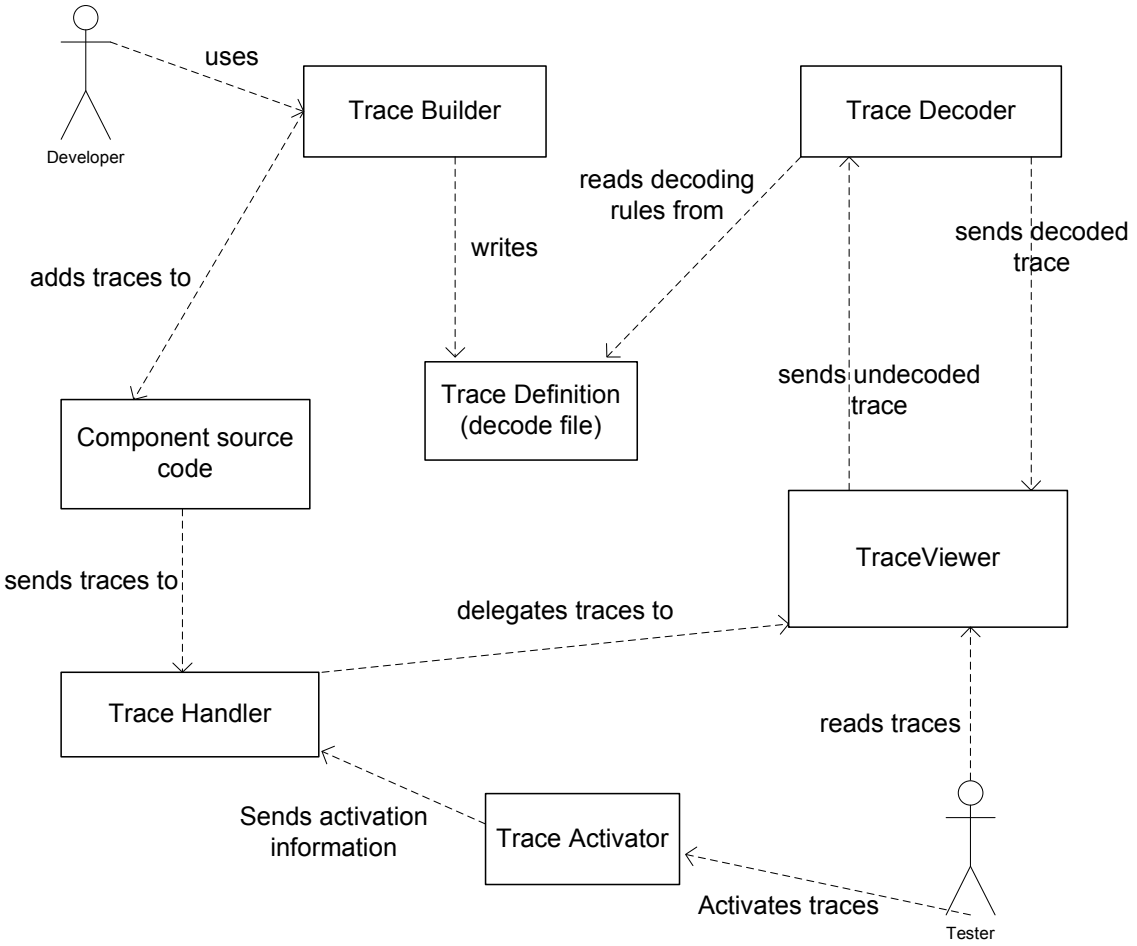

Figure 1. Abstract Open System Trace architecture

3.3. Trace Builder

Trace Builder is a tool being developed by Nokia. It is used to easily instrument traces into the source code of Symbian S60 applications. Instead of having to write traces by hand, the user can easily use Trace Builder's graphical user interface to click traces to his source code.

Trace Builder is an essential piece of the Open System Trace concept. The main reason for this is that even if the user could write OST traces to his source code manually, generating and especially maintaining the OST decode file would be very hard and sensitive for errors. Trace Builder handles generation and updating of the decode file whenever the user adds new, removes old or updates previous traces. The user must realize that if changes are made to the decode file manually, they will be erased when the trace project is opened again to the Trace Builder.

Trace Builder has a feature called code instrumenter. The user can select a template to instrument and files affected by it. Code instrumenter will then

instrument traces based on the template to all user defined files. For example, Trace Builder has a template for adding function entry traces to all functions. The user also has an option to add corresponding exit traces to functions. Using this feature, the user can instrument entry and exit traces to all functions in his application.

4. Requirements for TraceViewer

The TraceViewer project is using a slightly modified waterfall model as a software process model. In the waterfall model the project starts with gathering system and software requirements and then analysing them. The next phase is the design of the program. After the design, application is implemented. The last phases are testing the software and finally delivery and support/maintain. [Royce, 1970] In the TraceViewer project, testing the implemented features was done simultaneously with the implementation. That way we could ensure that the new features functioned properly and they did not break the old features. After consulting the customer, the following requirements were settled:

General:

1. UI layout as view

   TraceViewer is a view. Viewer commands are buttons in the view's client area. The view can be resized. There can be separate views for special purposes.

2. Carbide v1.x support

   TraceViewer must be usable in Carbide v1.x and later.

3. Operating instructions

   There must be a user manual or other operating instructions to the user to consult when needed.

4. Installation package

   TraceViewer must have an easy-to-use installation package for installation.

Functionality:

5. Display received traces in viewer display

   Received traces are displayed in the text viewer of TraceViewer. If some traces are dropped, the user must be informed about it.

6. Trace activation

   The user can activate certain trace groups. Groups are listed in dialog. The user selects the groups to be activated. An activation message is sent when user clicks ‘activate’.

7. Trace deactivation

   The user can deactivate certain trace groups. The groups are listed in dialog. The user selects the groups to be deactivated. A deactivation message is sent when user clicks ‘deactivate’.

8. Clicking of trace opens the corresponding source code & line where trace is defined

   The user double clicks a trace line in the viewer window. The source code editor is invoked and a source code line is opened where the corresponding trace is defined.

9. Trace filtering

   The user can set filtering rules for traces. The filter can be either inclusive or exclusive.

10. Pause

    The display update can be paused.

11. Trace line searching

    There is a search dialog for searching traces from the received traces. Regular expression searching is possible.

12. Trace line colouring

    The user can set colouring rules for traces. Every time when a matching trace line is received, it is coloured.

13. Trace line counting

    The user can set counting rules for traces. Every time when a matching trace line is received, its instance counter is updated.

14. Trigger

    User can set a specific trigger rule. The trigger can be either a start or a stop trigger. When a trace containing a start trigger rule is received, TraceViewer starts showing traces in the view. Traces received before the start trigger are dropped. When a trace containing stop trigger rule is received, TraceViewer stops the view.

15. Connecting/disconnecting to/from trace source

    The user is able to connect to the trace source. The user is able to disconnect from the trace source.

16. Storing data to log plain text log

    The user can specify a logfile where trace information is stored. The log contains a decoded trace and a timestamp.

17. Storing data to binary log

    The user can specify a logfile where trace information is stored. The log contains traces as binary format.

18. Opening of plain text or binary form log files to the TraceViewer

    The user can open previously saved log files to TraceViewer.

19. Trace ‘variables’

    The user can set rules to trace variable values. Every time a trace containing a variable value is received, the value of the variable is updated in the view.

20. Open System Trace support

TraceViewer is able to decode Open System Trace data format. TraceViewer is able to open Open System Trace decode files. TraceViewer is able to activate Open System Trace components and groups.

Based on these requirements, the architecturing work was started.

5. Architecture and design of TraceViewer

5.1. Architecture overview

The designing of the architecture was started based on the settled requirements. The main thing affecting the big picture of the architecture was the fact that TraceViewer is going to be a plug-in for Carbide.c++ framework which is based on the Eclipse framework.

Figure 2 shows the architecture overview of the TraceViewer. TraceViewer is divided into Engine and View. Eclipse has a Plug-in API that is implemented by TraceViewer Engine. Eclipse UI then creates the plug-in and the views specified in the plugin.xml of TraceViewer. The TraceViewer Engine is used to select the correct OST decode file to be opened and the Engine then passes the file reference to the OSTModelCreator which creates the OSTDecodeModel. When TraceViewer Engine gets trace data from the TraceSource which in this context basically means the S60 smartphone, it passes the traces through a list of DataProcessors. Each DataProcessor uses the trace somehow. The DataProcessors are explained below. When the trace arrives to the OSTDecoder DataProcessor, it is decoded by using the OSTDecodeModel created when opening the decode file. After decoding, the trace is forwarded to the next DataProcessor. In the end it arrives to the TraceViewer View which will then view the trace in its viewer.

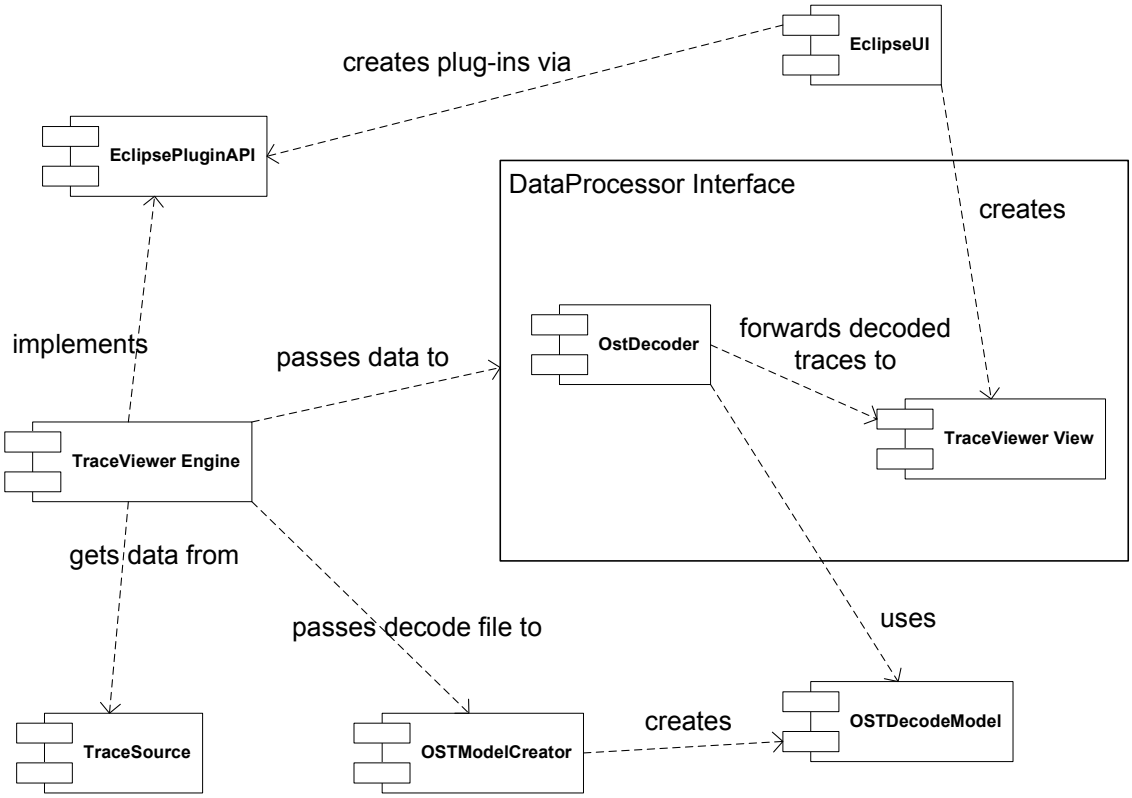

Figure 2. Architecture overview of TraceViewer

Based on the requirements, there is a need, for example, to decode traces, write a log, filter, count and colorize traces, trace variables inside traces and display them in the view. All these operations need to have an access to the trace so that they can modify it (decode, color), try to find rules from it (count, trace variable) or use the trace as it is (log, display). To enable this and easy extensibility, each trace is pushed through a list of DataProcessors. The DataProcessor interface defines a processTrace method that has to be implemented in every class implementing this interface. The list of DataProcessors is maintained in the Engine of TraceViewer. Every DataProcessor will get the trace in turn as it is from the previous DataProcessor and can modify or use it the way it wants. After the processing of a single DataProcessor has ended, the trace will be given to the next one until the whole list has been iterated through.

Figure 3 describes an initial data flow between components in TraceViewer. Engine makes a connection to the smartphone using a connection interface. The connection here is abstract. The phone will then delegate traces to the DataWriter. Because of the fact that there can be millions of traces to be viewed, traces cannot obviously be stored in the memory. This is why all traces are driven straight to a binary file in the hard disk. The DataWriter will create this

binary file. The DataReader reads the binary file and pushes traces to the Engine one by one. The Engine will then iterate the trace through the list of DataProcessors. Here the order of DataProcessors is Decoder, TriggerProcessor, FilterProcessor, Logger, LineCountProcessor, VariableTracingProcessor, ColorProcessor and finally the TraceViewerView. The order can be easily modified just by inserting the DataProcessors in a different order. Also, the same DataProcessor can be inserted multiple times. For example Filter might be used before Decoder to filter traces still in binary format and then again after Decoder to filter using decoded trace. Note that not all DataProcessors are visible in Figure 3 and some of the names are abbreviated. All DataProcessors can be found in subsection 5.2.2 on page 20.

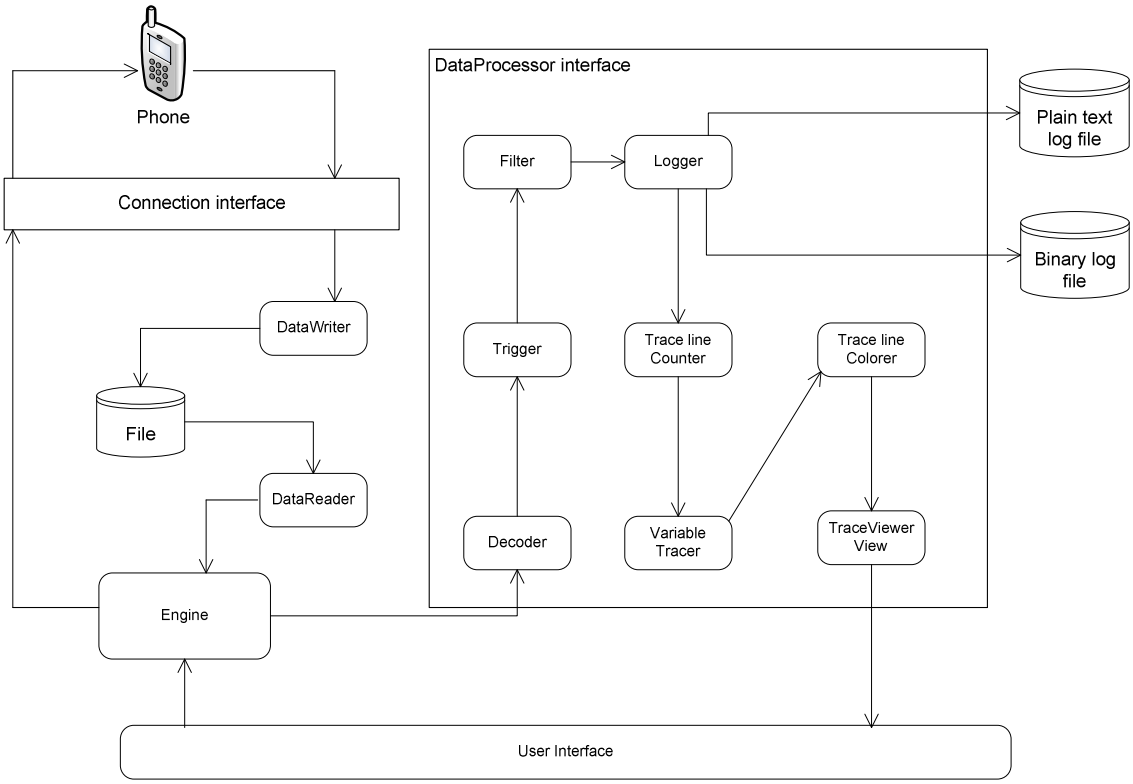

Figure 3. Data flow diagram of TraceViewer

5.2. Class design

5.2.1. TraceViewer Engine

The TraceViewer Engine package contains the central logic of TraceViewer. The Engine receives the traces from the trace source, writes them to a file, reads the same file, processes traces and finally shows them in the view. Figure 4 introduces the main classes of the TraceViewer Engine. Figure 4 is divided into two parts, the Core part and the OST part. The Core part implements common functionality for the application containing mostly interfaces that have to be

implemented in order to actually be able to use the software. The OST part implements those interfaces and registers itself to the TraceViewer Engine. The OST part is actually a separate plug-in and can be easily replaced with another plug-in without making changes to the Core part if the OST format is changed to something else.

The Core part of TraceViewer Engine (Figure 4) contains the classes (c) and interfaces (i) described in Table 1.

Name	Description
ConnectionProperties (c)	User defined properties used when connection to trace source.
TraceViewer (c)	Main class of the application. Holds the list of DataProcessors and contains references for DecodeProvider, Connection and DataReader.
DecodeProvider (i)	DecodeProvider interface contains methods for creating decode model and using it to decode binary traces.
Connection (i)	Connection interface contains methods for handling connection and sending data to trace source and setting a media filter for the connection.
ConnectionImpl (c)	Implementation of the Connection interface. Connects the trace source. The connection method here is unspecified.
ConnectionInit (i)	The ConnectionInit interface contains methods for initializing the connection and wrapping activation message to a specific format before sending it to the trace source.
MediaFilter (i)	MediaFilter interface provides methods for setting the filter source and target channels and filtering the source.
DataReader (i)	DataReader interface contains methods for starting, pausing and shutting down reading of traces.
DataProcessor (i)	DataProcessor interface provides a method to process a trace.
DataProcessorImpl (c)	Implementation of the DataProcessor interface. Different kinds of data processors using and modifying the trace.

TraceProperties (c)	Represents a trace. Contains trace attributes such as timestamp, trace string and a trace location in data buffer.

Table 1. Classes and interfaces in Core part of TraceViewer Engine

The OST part of the TraceViewer Engine is described in Table 2.

Name	Description
OstReader (c)	Implementation of the DataReader interface. Implements methods for reading traces from the binary file. Uses OstMessageProcessor to split data into TraceProperties class representing a trace.
OstMessageProcessor (c)	Processes (large) data buffers and returns messages containing only one trace until the buffer is read.
OstMessageFilter (c)	Implementation of the MediaFilter interface. Filters data coming from the trace source and only writes interesting data to the binary file. Uses OstMessageProcessor to split data buffer into single messages.
OstConnectionInit (c)	Implementation of the ConnectionInit interface. Initialises connection to the trace source and wraps activation messages to the correct format.

Table 2. Classes in OST part of TraceViewer Engine

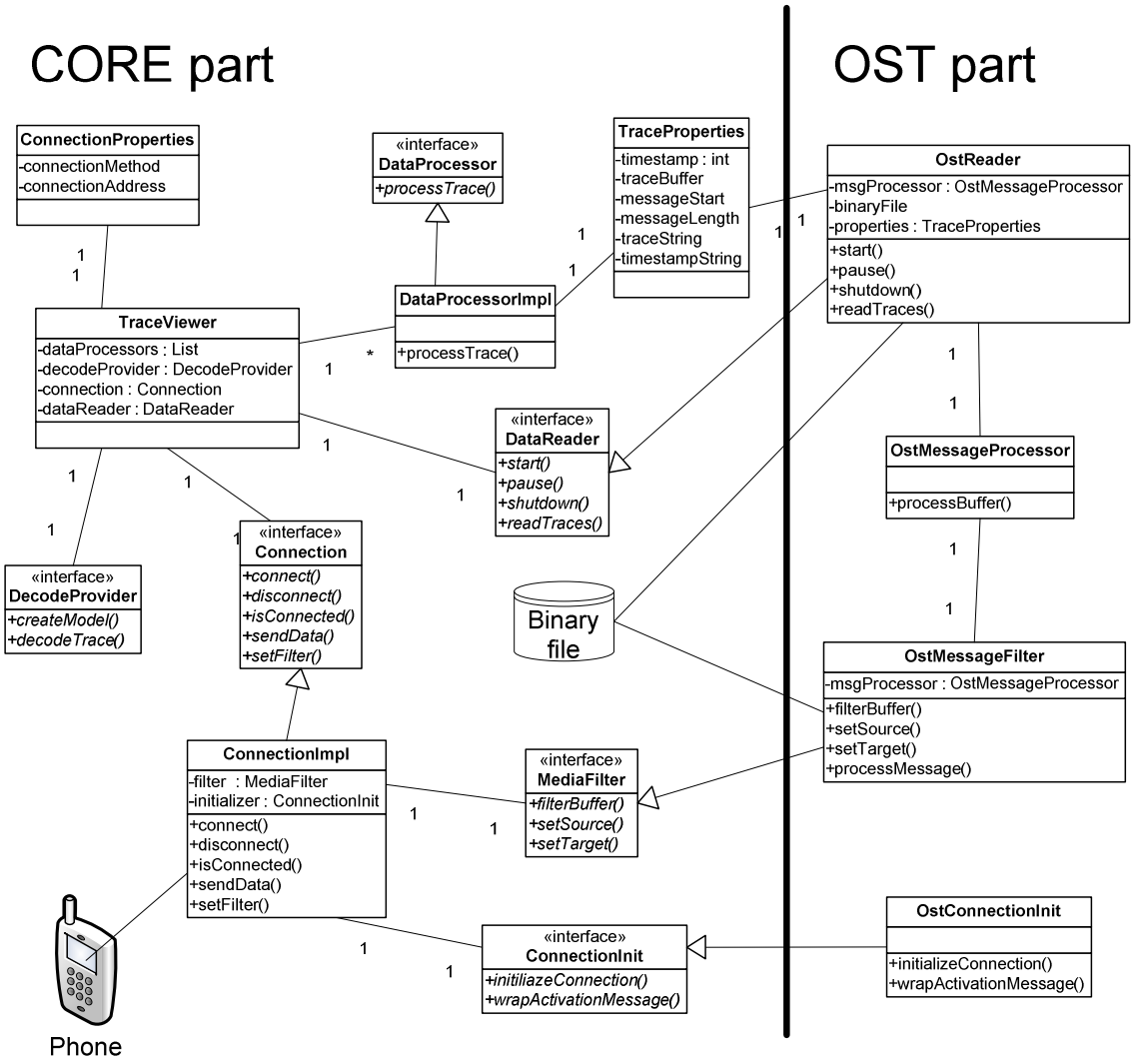

Figure 4. Engine overview

5.2.2. DataProcessors

The DataProcessors process traces read from the file. The Engine has a list of the DataProcessors and the order in which they are executed. The DataProcessors can either change the trace data or use it in some specific purpose. All views that show data to the user are also DataProcessors. The class diagram is illustrated in Figure 5.

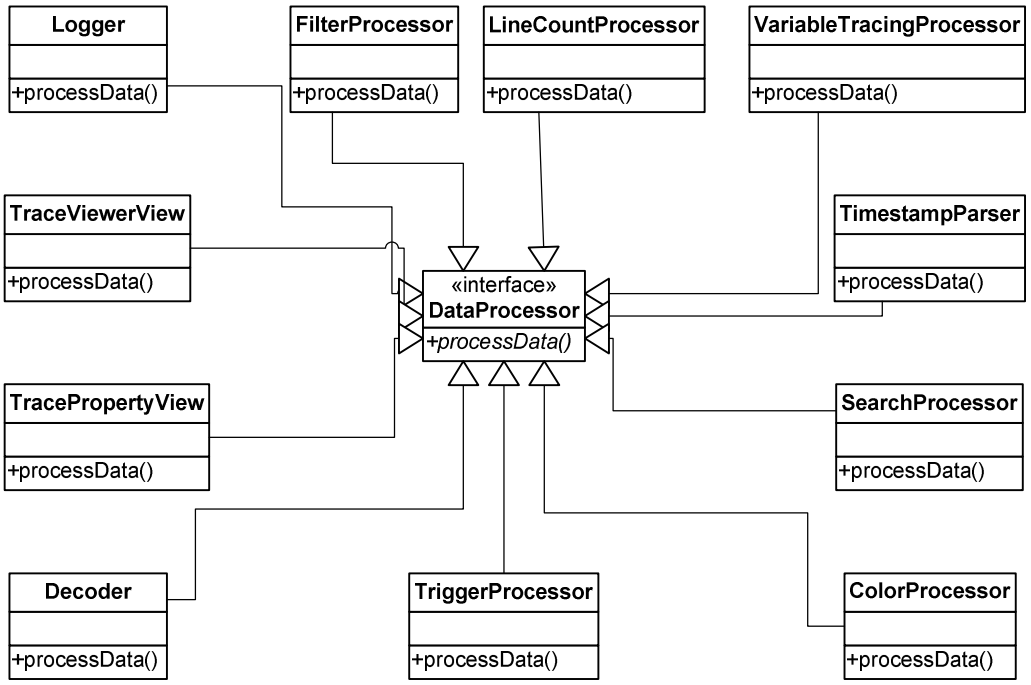

Figure 5. The class diagram of DataProcessors

Every DataProcessor implements the DataProcessor interface which enables inserting them into the same processing list. All of them implement a processTrace method which is called from the Engine for every DataProcessor in the list. A more detailed description of the DataProcessors can be found in Table 3.

Name	Description
Logger	Handles writing traces to a file. The file can be either a plain text file or a binary file. A binary file contains all the information available from the trace when a plain text file only saves text visible in the view.
FilterProcessor	Used to filter traces received from the trace source with user specified rules.
LineCountProcessor	Used to count occurrences of user specified words in the received traces.
VariableTracingProcessor	Used to monitor a state of a variable included in the received traces.
TimestampParser	Parses timestamp from the binary traces to user readable form.
SearchProcessor	Searches through the view and the file for traces containing user specified search criteria.

ColorProcessor	Colours traces with different colours according to user specified rules.
TriggerProcessor	Checks traces for user specified start and stop trigger rules. When a start trigger hits, traces are started to show up in the view. When a stop trigger hits, the trace flow to the view is stopped.
Decoder	Decoder uses the OST decode model to decode binary traces. Decode also attaches metadata information specified in the decode file to the decoded trace.
TracePropertyView	Updates information about Line count and Variable tracing rules in two separate tables displayed in this view.
TraceViewerView	Updates newly processed traces to the text viewer of this view.

Table 3. Description of DataProcessors

5.3. User interface design

5.3.1. Overview

Unlike the rest of the TraceViewer project, all user interface components are developed with a rapid prototyping method. Rapid prototyping is a process of quickly putting together a working model to gather user feedback [UsabilityNet, 2006]. The model is then changed according to the feedback. Depending on the number of user feedbacks of the previous comment round, the changed model is given back to a new comment round. After some number of rounds, the model is ready. Rapid prototyping is often used when developing user interfaces. Dialogs were created using the Eclipse Visual Editor which enables easy building of graphical user interfaces without coding a single source code line [VEP, 2007].

All TraceViewer dialogs are accessed through Actions. Actions are commands which can be triggered from the user interface. An Action can be attached to a menu bar, a toolbar or to a button. Figure 6 shows a toolbar containing several Actions. Actions are defined in the JFace package of Eclipse.

Figure 6. Toolbar containing actions

5.3.2. Base user interface

Figure 7 shows the Eclipse user interface with the additions contributed by the TraceViewer plug-in. Figure 7 contains the TraceViewerView and the TracePropertyView. The TraceViewerView contains a menu which is available from the white arrow within the toolbar. Actions can be launched from the buttons in the views toolbar and in the menu.

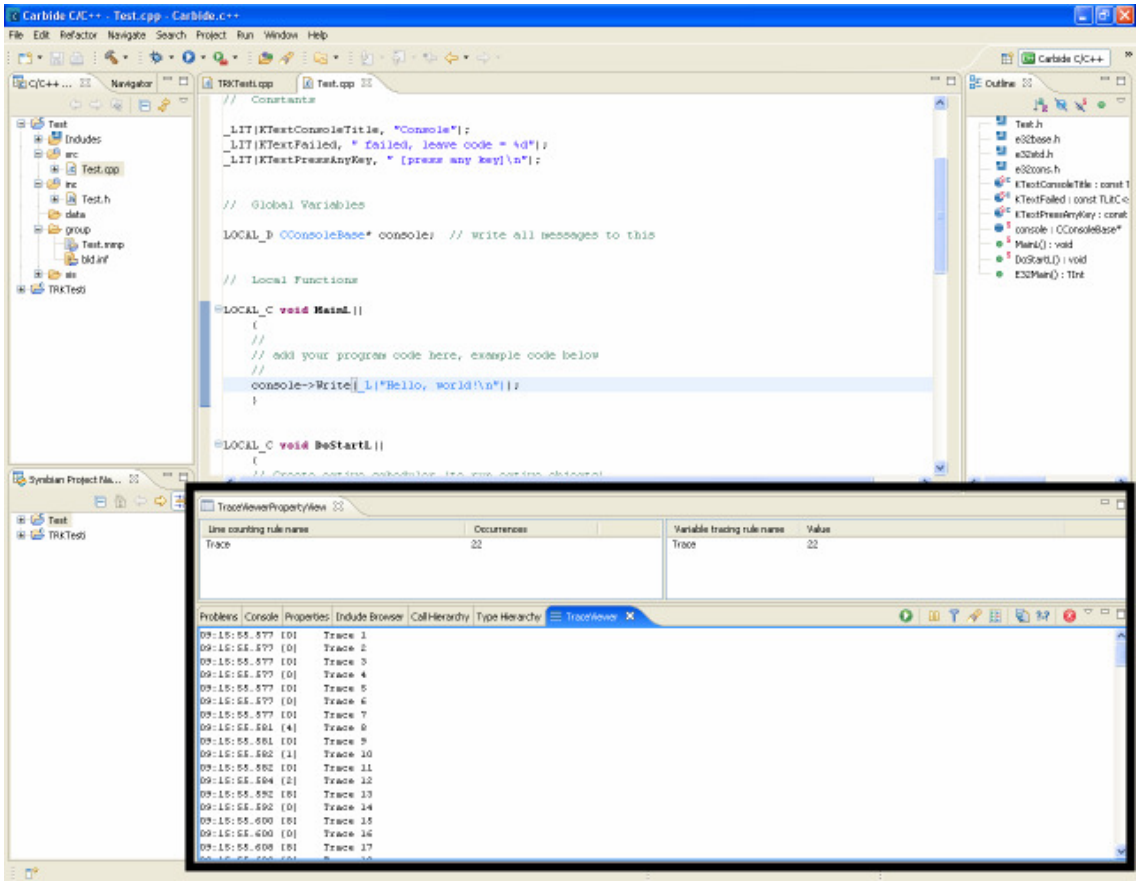

Figure 7. TraceViewer Base UI

5.3.3. DataProcessor dialogs

The DataProcessors can be divided here into three groups. First to those where the user can define rules and then apply them to the application, second to those where the user can see the information and make changes to the behaviour of the DataProcessor, and third to those not needing any user interface. The first group consists of FilterProcessor, TriggerProcessor, ColorProcessor, LineCountProcessor and VariableTracingProcessor. Logger, SearchProcessor, TraceViewerView and TracePropertyView belong to the second group and Decoder, TimestampParser to the third not needing any user interface.

The DataProcessors from the first group need very similar user interfaces. There has to be a list of the user defined rules and a way to apply some of them. Also a dialog creating new rules and editing old ones is needed. Figure 8 shows the Filter Rules dialog having a tree view where the user defined rules are listed. A tree enables an easy grouping of rules and fast enabling/disabling of a whole group at a time. The dialog has toolbar buttons for creating new Group, Add rule, Edit rule, Remove rule and Clear all rules. The same actions can be found in a context menu of the tree items normally by right clicking them with the mouse. The user can move the rules to groups or to different places in the tree by dragging them with the mouse. All the rules the user creates are saved to a XML file which is then automatically imported when TraceViewer starts. This way the rules are always there. The configuration file can be also exported and imported manually.

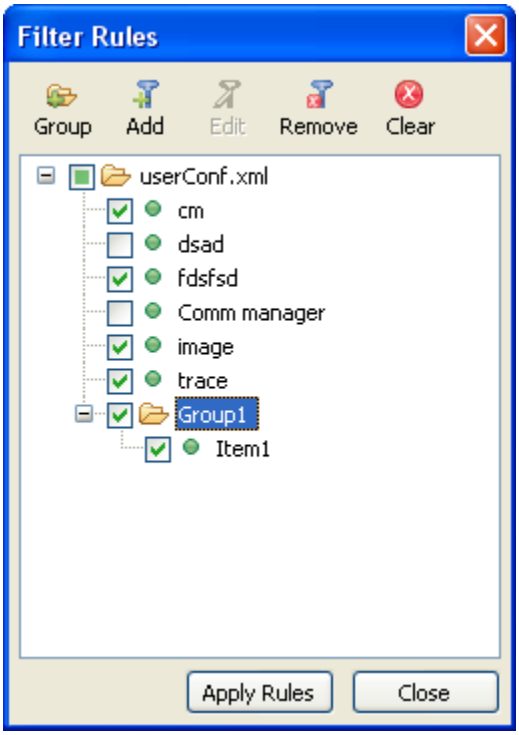

Figure 8. Filter Rules dialog

The rule adding/editing dialog is shown in Figure 9. This is an example of Color rule add/edit dialog and it has two fields that other add dialogs do not have: foreground color and background color fields. Every rule has a name which is shown in the DataProcessor's tree dialog. On the left is a list of available rule types for this DataProcessor which here are Text rule and Component/Group rule. When a rule is changed from a list, the middle right composite changes to

show needed fields for the selected rule and the bottom right composite shows information about the selected rule.

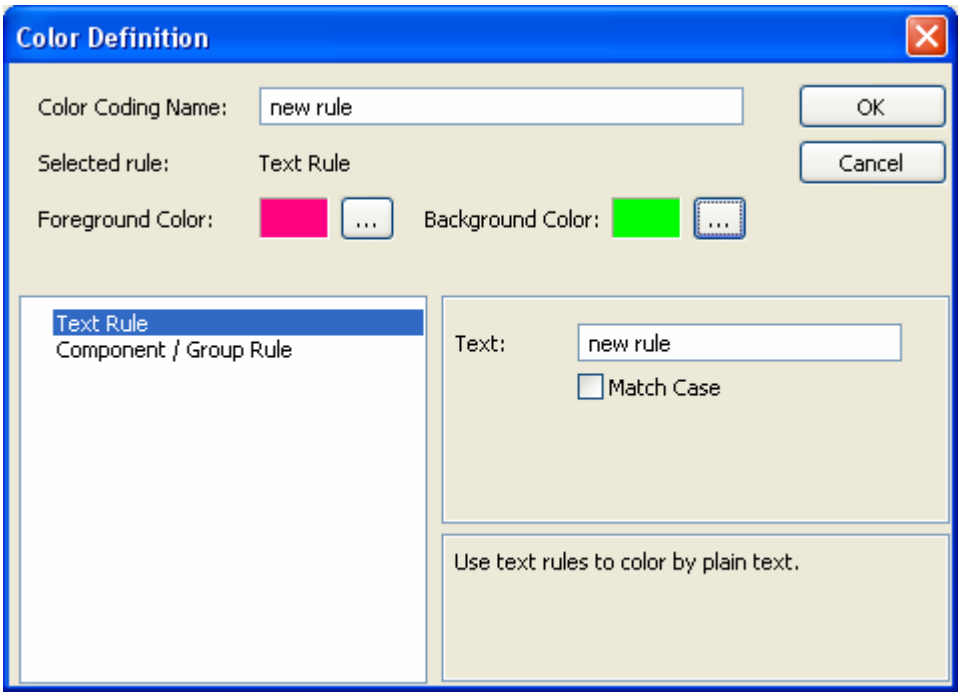

Figure 9. Color Adding / Editing dialog

From the second group of DataProcessors, TraceViewerView and TracePropertyView user interfaces were already shown in Figure 7. TraceViewerView consists of a SWT TextViewer with a scrollbar and a toolbar with buttons for the Actions. TracePropertyView has two tables separated with a sash line which can be used to change the size of the tables. The table on the left shows the information on Line count rules and the one on the right on Variable Tracing rules. The Logger DataProcessor user interface is shown in Figure 10. It enables the use cases for writing plain text or binary log files and opening a plain text or binary log file to the viewer. The timestamp can be omitted from the plain text log file for easier comparation between two log files.

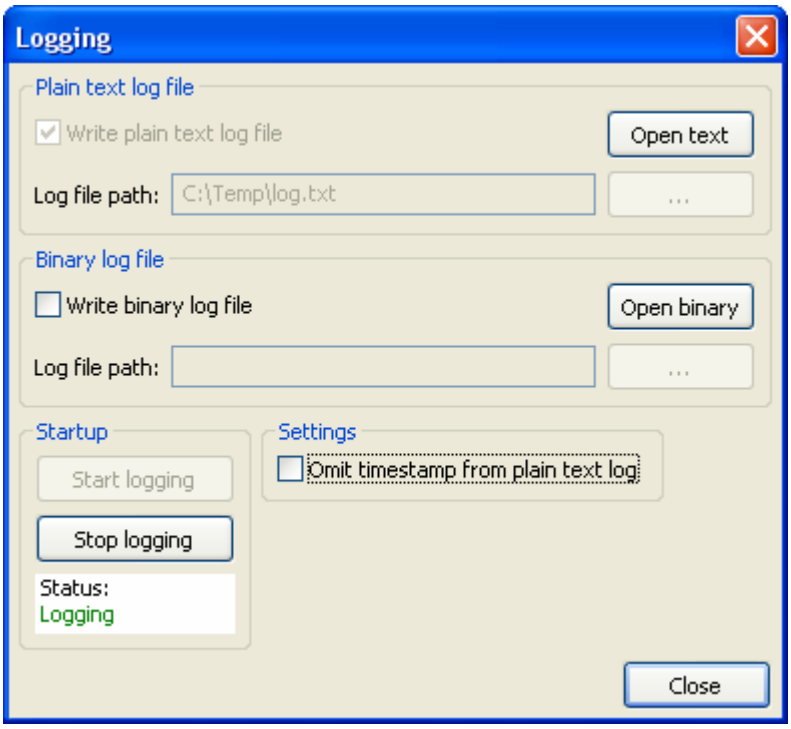

Figure 10. Logging dialog

SearchProcessor, too, needs its own user interface for enabling the user to search from traces. The Search dialog shown in Figure 11 contains basic search functions such as "Match whole word", "Match case", "Regular expression search", search direction and stopping of the search. In the case of TraceViewer, the stop button is extremely important because the data set can be millions of traces. The Search dialog also has a progressbar informing the user visually in which line the search is running.

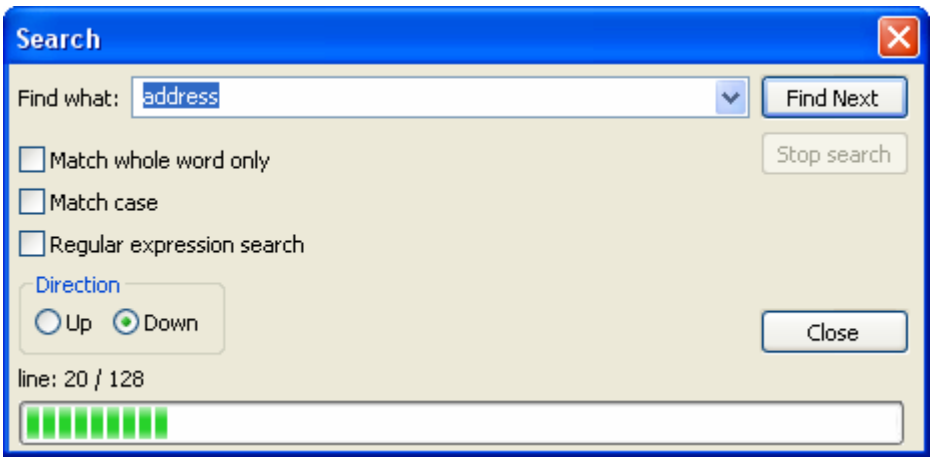

Figure 11. Search dialog

5.3.4. Trace activation dialog

Trace activation information is read from the decode model stored in the OSTDecoder Plug-in. Trace activation dialog in Figure 12 contains a component table and a group table. The user can choose one or more components from the component table and then the groups from the selected components are updated to the groups table. The groups can be activated by selecting one or more and clicking the Activate button. Double-clicking changes the state of the group as well. The components and groups can be also selected using the filter field under the tables. Writing on the filter field will select all components or groups containing the written string. Wildcard * can be used to select all. The Apply or OK button will send the changes to the smartphone. Activation configurations can be saved to a XML configuration file and then loaded when needed again.

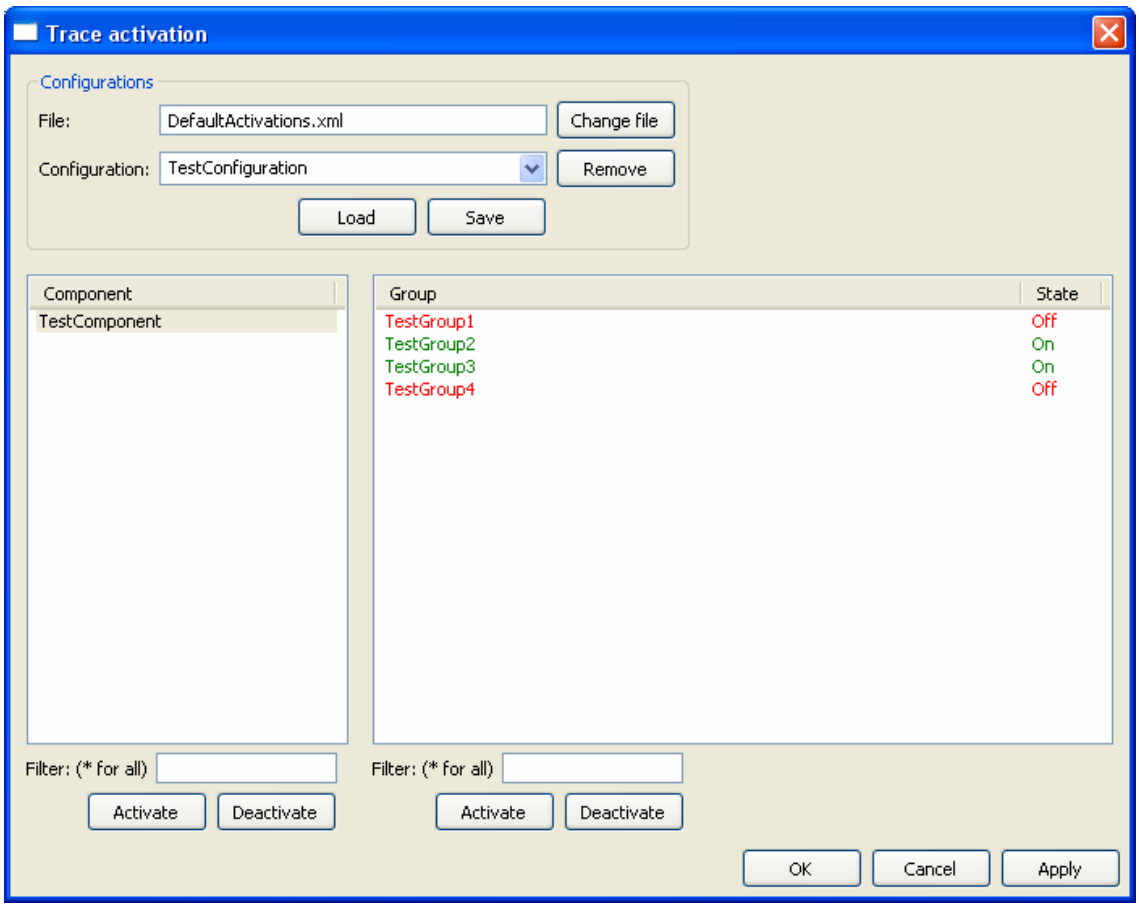

Figure 12. Trace activation UI

5.4. A behavioral view

5.4.1. The big picture

Figure 13 demonstrates a normal behaviour when TraceViewer is receiving traces and processing them through DataProcessors. All classes are not visible in the figure to keep it easier to read. TraceViewerView (the first object in the picture) is also an implementation of the DataProcessor interface (the last object in the picture) but it's separated here to represent the user interface which is used and viewed by the user. The normal behaviour consists of the following phases:

1. A user interface is used to launch a connect command. The connect call goes to the Engine which uses connection preferences to create a specific type of Connection which will forward the call to the Media.
2. The Media returns an indication of the connection state back to Engine. Engine notifies the connection and informs TraceViewerView to indicate it.
3. Media starts sending data (traces) to the Connection.
4. Connection uses MessageFilter (not visible in picture) to filter received data and then writes the result to the File.
5. DataReader reads the data from the File and generates traces out of it. Traces are then sent one at a time to the Engine.
6. Engine passes the trace to all DataProcessors in turn. DataProcessors can modify the trace or use it otherwise.
7. At some point, TraceViewerView will get the trace and show it in the user interface.
8. A user interface is used to launch a disconnect command. The disconnect call goes to the Engine which forwards the call to the currently open Connection. The Connection again forwards the call to the Media.
9. The Media returns an indication of the connection state back to Engine. Engine notifies the disconnection and informs TraceViewerView to indicate it.

Phases from 3 to 7 are repeated as long as data is received from the Media or user invokes a disconnect command.

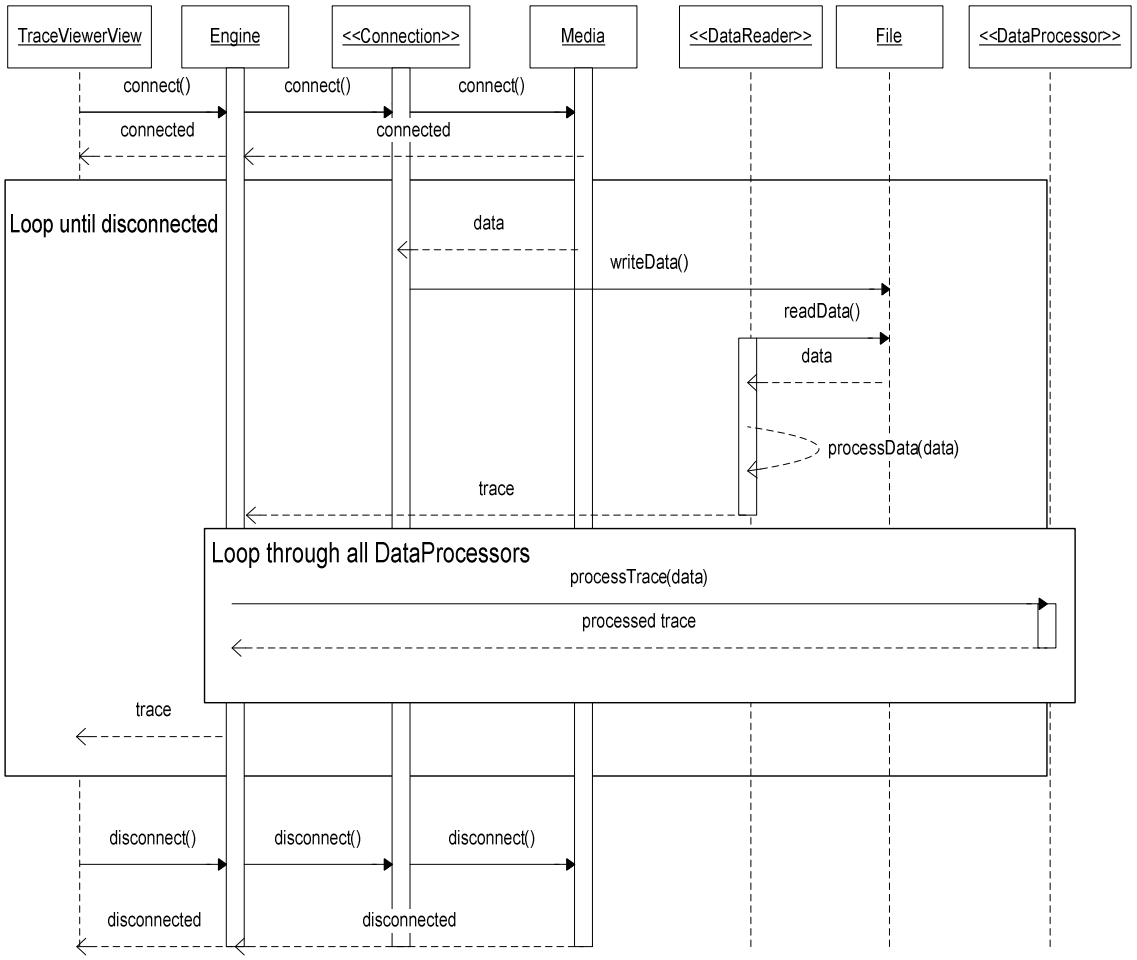

Figure 13. Normal TraceViewer behaviour

5.4.2. Trace displaying

A natural choice for user interface components for TraceViewer was Standard Widget Toolset (SWT) and JFace Viewer on top of it. In JFace viewers, there was a couple of options about the structure to be used to display the traces to the user. Because the number of traces received from the trace source is unknown and can thus be unlimited, the structure must be able to handle for example 100 million traces.

After reading the help pages on JFace Viewers from Eclipse SDK [2007], the following candidates were found:

- TreeViewer
- ListViewer
- TableViewer
- TextViewer.

TreeViewer was abandoned immediately because I could not think a clever way

to show traces in some kind of a tree view. Traces contain a timestamp field and an actual trace field so two columns are needed. ListViewer can only display one string at a line and can not display tabulator character ('\t') as tabulator so it was also dropped from the candidate list. TableViewer was a good candidate because the number of columns can be easily set and every column can contain its own string. Also, reordering of columns could be done just by dragging them. The biggest advantage of the TableViewer was that the underlying structure, SWT Table, supports VIRTUAL attribute. Normally every row in SWT Table represents a TableItem object. When VIRTUAL flag is on, only the items visible in the screen will be created and therefore no extra effort or memory footprint is spent creating items that are never viewed. This sounds very nice considering that we might have 100 millions traces and all of them are most probably not ever viewed. TableViewer was briefly tested but a problem occurred. Even though most of the actual TableItem objects are never created, the SWT Table still holds a data structure containing those TableItems even if they are *null*. When the table gets really big, the structure itself takes up so much memory that Eclipse throws *Out of memory* error. This happened when the size of the Table exceeded 6 million traces and as said, there can be no limit for the number of traces. TableViewer was dropped and the last choice was TextViewer.

TextViewer can handle displaying a tabulator character so it is not an issue. The problem with TextViewer is of course that when there are a lot of traces, the memory consumption will get higher and higher if every trace is added to the underlying text widget in TextViewer. This issue was solved by including only a few hundred traces in the widget at a time. The following paragraphs explain how the displaying of traces is done and how the user is still able to scroll all the traces received.

In TraceViewer, there is a global variable called the *blockSize*. It defines how many traces make up one data block. When reading traces from the trace source, TraceViewerView will take care that the amount of traces in the TextViewer when the view is updated does not exceed two data blocks. If TextViewer contains more traces than two data blocks, TraceViewerView will cut the data from the beginning so that after cutting there is at least one block and at most two blocks of traces in the view. While cutting, TraceViewerView keeps note about the number of traces cut away in a variable called *showingTracesFrom*.

As an example, suppose that there are 2500 traces coming from the trace source and saved to the File. The blockSize variable is 200. Let us assume that the view will update after 1000 traces are processed (normally there is a time interval how often the view is updated). After 1000 traces are processed and the update for the user is invoked, the TextViewer widget contains 1000 traces. The view will cut 600 traces from the beginning so that there will be 400 traces (2 blocks) left in the view. The view updates the showingTracesFrom variable to 600 because that is the amount of traces cut from the view. The view containing 400 traces will be shown to the user. Figure 14 illustrates this situation.

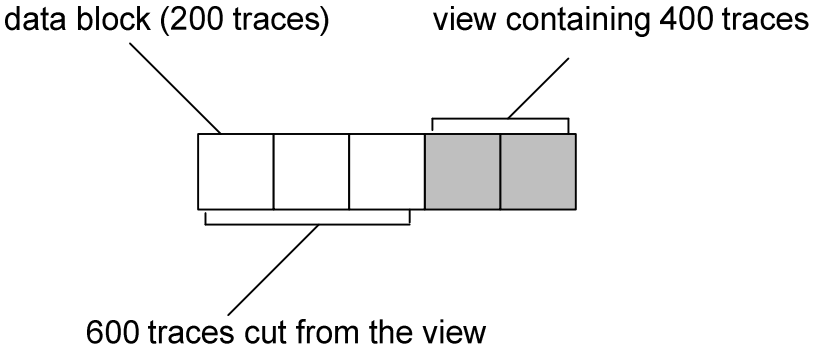

Figure 14. View containing only part of all traces

The next update occurs when 2000 traces are processed. TextViewer will now contain 1400 traces. Now 1000 traces are cut and 400 are left to be shown to the user. The showingTracesFrom variable is updated to be 600 + 1000 = 1600. The last 500 traces are processed and the view will update because the DataReader informs the view when it reaches the end of the file. TextViewer now has 900 traces which are cut again. After the cut, 300 traces are left in the view. The showingTracesFrom variable is updated to 2200.

Let us now assume that the user wants to scroll up the view to trace number 1250. There are a few options as to how this is done.

1. The user will scroll “gently” from 2500 to 1250 so that every time he hits the end of the previous block, the view will get one block more and attach it to the block the user is seeing right now. One block is then removed from the other end of the data in the TextViewer so that the total amount of traces stays between 201 and 400. The first time that the view will get more data is when the user crosses point 2200 which is the value from the showingTracesFrom variable. The view gets one block of traces (traces from 2000 to 2200) and inserts

them to the beginning of the current data. Then traces 2400 - 2500 are cut from the end of the data, and we have 400 traces again in the view. Now the view is showing traces from 2000 to 2400 and showingTracesFrom variable is naturally set to 2000. Finally, when the user gets to line 1250, showingTracesFrom is 1200 and the view contains traces from 1200 to 1600.

2. The user will take the scrollbar and quickly drag it to the desired point (in this case 1250) before the view notices that more traces are needed. If the traces that the user wants are not attached to the current data in the view, two blocks of traces are asked instead of one and the view is generated from them. So if the scrollbar is dragged to point 1250, the view will get traces from 1200 to 1600 and update showingTracesFrom to 1200.
3. Any combination of options 1 and 2.

The data fetching is done as follows:
The view will fetch 1 or 2 blocks of data depending on the way of scrolling. Firstly the view determinates if we need data to the beginning or to the end of the current view. If the user is scrolling up, data is needed before the current view. If the user is scrolling down, data is needed after the current view. In both cases, the view will first tell the TraceViewer engine the first trace it needs and the number of blocks (1 or 2). Engine has a map where it stores file positions to the first traces in every block. In other words, Engine knows where to find every trace block. Engine will first set up the current file position to the start of the trace block and then either create or wake up a *ScrollReader* which is used to get blocks of data from the file. ScrollReader is notified how many traces it should get. ScrollReader will mark all read traces as "scrolledTrace" so that DataProcessors and mainly TraceViewerView will know not to append these traces normally to the end of the current view but use them to generate block to be attached to either the beginning or to the end of current view. After the desired amount of traces is read, ScrollReader will go to sleep. The views will then insert the traces to the correct place, update the view to the user and update the showingTracesFrom variable. Figure 15 demonstrates the sequence of the scrolling procedure.

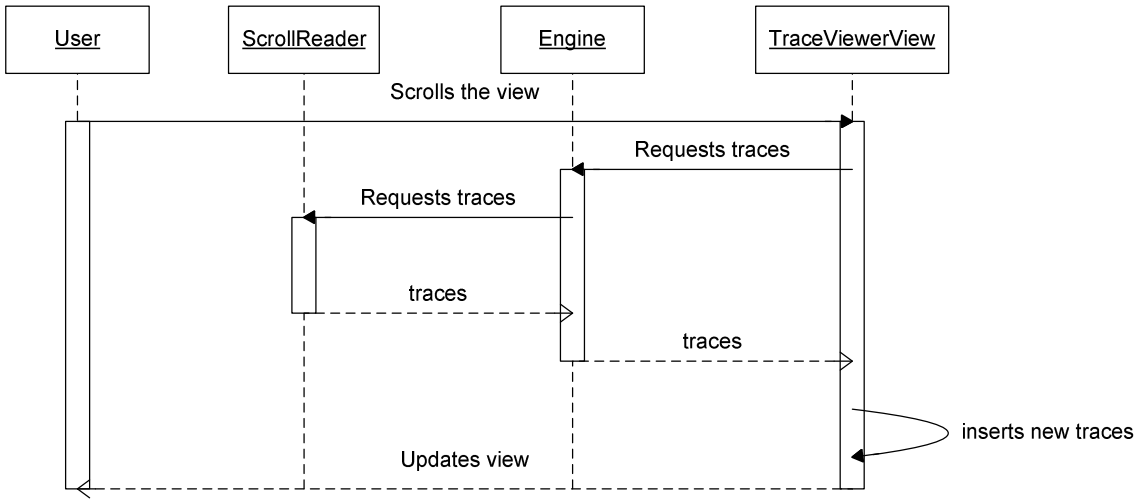

Figure 15. Scrolling sequence diagram

Because TextViewer contains a few hundred traces only, the user can not use the scrollbar that is built inside TextViewer. The original scrollbar was disabled and a separate scrollbar was developed to replace it. The ScrollBar is attached to the TextViewer just like the original would be so the user does not know that there is something strange happening while he is using it. The maximum value of the ScrollBar is set to be the number of traces in the file. A listener is implemented to react to changes in the ScrollBar value. When the value goes out of the interval the view currently contains (for example 2200-2600), the view is notified to get more traces. The value of the ScrollBar tells the view what trace blocks should be read from the file. Other listeners have also been implemented to keep the value of the ScrollBar correct if the user scrolls by some other means such as using the page up/page down or up/down keys or the mouse wheel.

6. Evaluation of the software and improvement propositions

6.1. Performance evaluation

To get a rough estimation about the maximum trace output ratio in TraceViewer, a binary file containing 5 million OST traces was opened and the time spent was calculated. The traces contained 0 to 6 parameters to be decoded. The test computer was Pentium 4 3,60Ghz with 2 GB of RAM. Table 4 shows the results. The first test was done without any extra DataProcessors active by just decoding the timestamp and the trace itself and then showing them to the view. Outputting 5 million traces took 3 minutes and 19 seconds and the output was slightly over 25000 traces per second. In the second test, ColorProcessor, LineCountProcessor and VariableTracingProcessor were added with two text rules for each so that both rules hit at least half of the traces. The time taken increased by 1 minute and 14 seconds and the output decreased to 18000 traces per second. Next, plain text logging was added to the previous test which means that every trace is written to a file in the hard disk in human readable form. Adding the logging increased the time to 5 minutes and 21 seconds while the output dropped to 15500 traces per second. The last test was almost the worst case scenario including also FilterProcessor with two text rules and also a binary logging. Because FilterProcessor writes filtered results to another file, in this use case there are 4 files being written and 2 read at the same time. This really gives the hard disk some tough times and it can be seen from the results. Outputting 5 million traces took 11 minutes and 20 seconds. The output dropped to 7300 traces per second which is 29% of the output got without any extra DataProcessors. Anyway, even the output of 7300 traces per second is more than enough for normal debugging purposes.

DataProcessors	Time taken	Output
TimestampParser, Decoder, TraceViewerView	3 min 19 sec	25126 traces / s
All above, ColorProcessor, LineCountProcessor, VariableTracingProcessor	4 min 33 sec	18315 traces / s
All above, Logger with plain text logging	5 min 21 sec	15576 traces / s
All above, FilterProcessor, Logger with plain text and binary logging	11 min 20 sec	7353 traces / s

Table 4. Time taken to read and display 5 million traces

To find the bottlenecks of the system, TraceViewer was tested with a Java Profiler called JProbe Suite. JProbe provides performance, memory and code coverage analysis for Java software [JProbe Suite, 2007]. The performance test was done with all possible DataProcessors active with a same amount of rules (2) to test which ones take the most CPU time. The data was again the 5 million OST traces with 0 to 6 parameters to be decoded. Figure 16 shows the CPU times spent by different DataProcessors. From the Figure 16 we see that the DataProcessors together take 1375 CPU time units. The time unit here is not important but the comparison of different DataProcessors. TraceViewerView and TracePropertyView do not update the user interface in the processData method, otherwise the times would be much higher. From Figure 16 we see that FilterProcessor, TimestampParser and Decoder take most of the time. FilterProcessors time is explained by the fact that it also has to write traces passing the filter rules to the file. TimestampParser must decode a big Long number to a human readable form and must deal with StringBuffer and String. Decoder took 641 time units which is almost half of the whole time spent in all DataProcessors. What takes time is that Decoder must find the right decode parameter information by first finding the right component and the right group from the lists. Then a StringBuffer, DataBuffer and an offset number are passed to every decode parameter class as a parameter and each parameter class decodes its own data reading from the DataBuffer using the given offset. Finally the trace is fully decoded and can be returned to be given to the next DataProcessor.

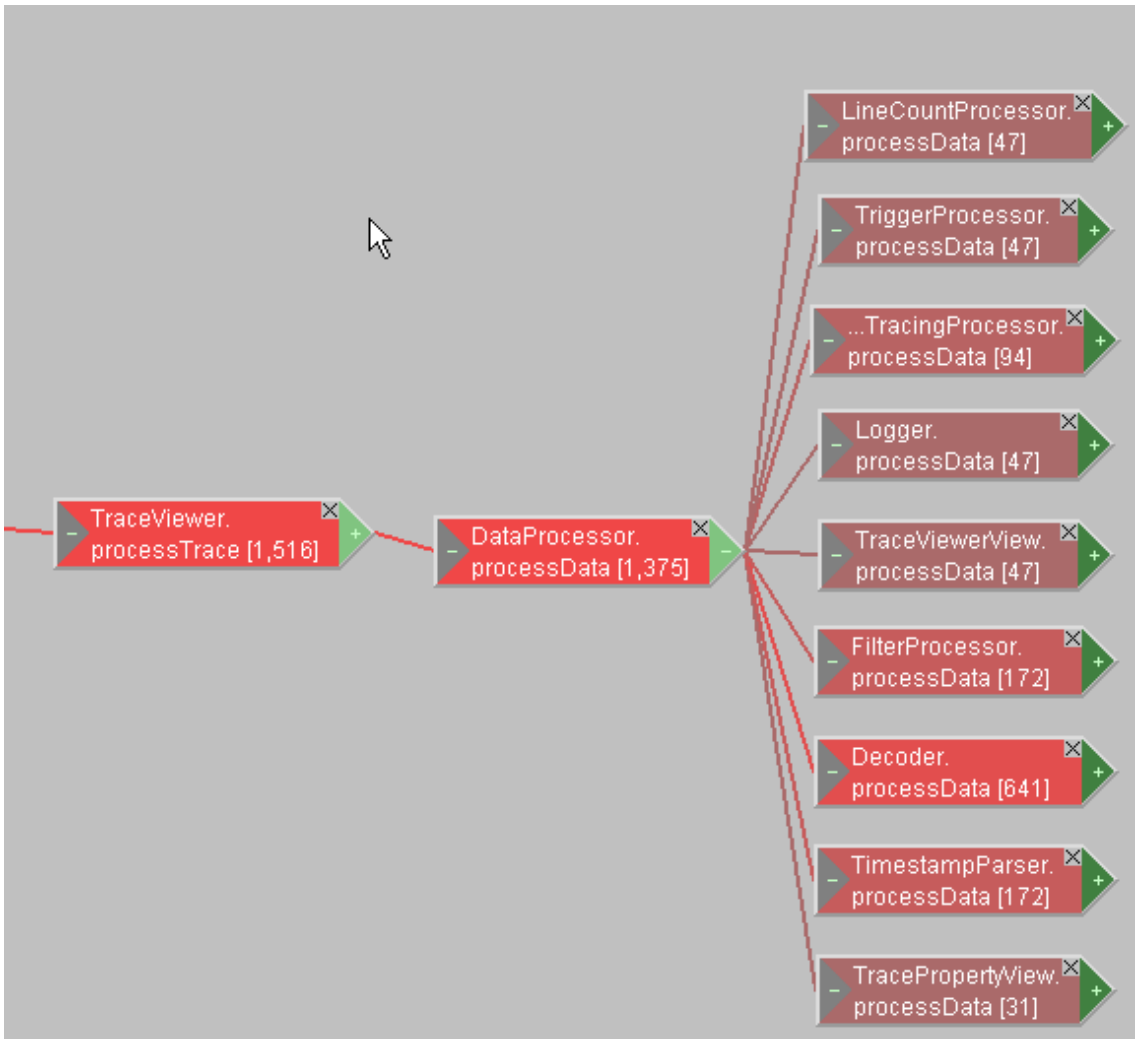

Figure 16. CPU times taken by different DataProcessors

Memory consumption was also analysed using JProbe Suite by opening 5 million traces. In Figure 17 we see the overall memory consumption of Eclipse IDE containing TraceViewer plug-In. The blue color in the figure signifies memory in use and yellow the amount of allocated memory from the system. The green line represents the start of the use case and the red line the end. The red dots in the bottom of Figure 17 depict CPU usage.

When the use case starts, Eclipse is loaded and approximately 12 megabytes of memory is allocated and about 8 megabytes are in use. About in the time of 01:00 in Figure 17 there is about a one megabyte drop in the memory usage graph which is caused by the Java garbage collection. About 10 seconds later, the reception of traces is started. Immediately, approximately 3.5 megabytes of memory are allocated and the memory usage peaked. Peaks in the figure depict the traces being attached to the TextViewer widget and the drops are the cuts from it before updating the view to the user. About in 3 million traces, some

more memory is allocated. At 5 million traces, a total of 16 megabytes of memory is allocated. The trace flow stopped but memory usage does not drop. This is caused simply by the fact that Java did not launch a garbage collection at that time. From the figure we will see that the memory consumption will not get very high even if there are a lot of traces. The overall consumption rose very little and that can be explained by the fact that TraceViewer keeps record of the indices in the file where every trace block can be found.

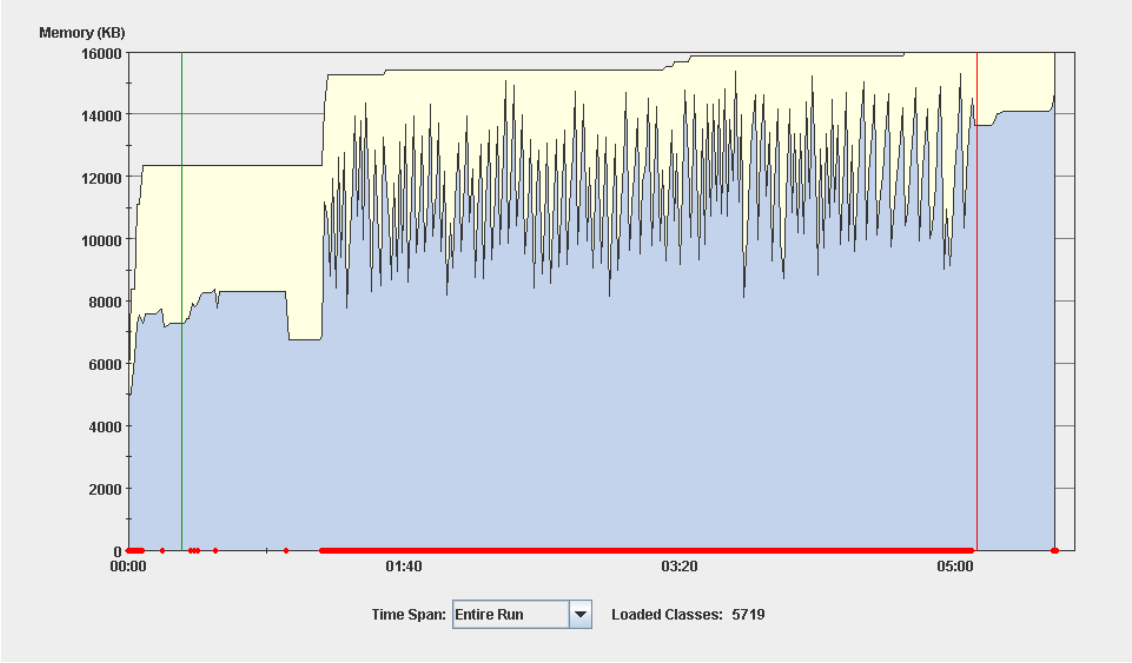

Figure 17. Memory consumption while opening 5 million traces

6.2. Overall evaluation

With a powerful computer, TraceViewer is able to receive and process 25000 traces per second. That is much more than any human can process. TraceViewer can be used to easily find the interesting information among the traces and use other means to follow what the application is doing. TraceViewer does what it should do and does not consume too much memory. If the developer uses Carbide.c++ as an editor for his/her application, using Trace Builder to instrument and TraceViewer to display traces are worth a try. The user being able to jump from a trace to a code line defining is a timesaving feature and helps the user to easily follow the program execution without the slowness of ordinary debug methods.

6.3. Improvement propositions

TraceViewer was implemented using the original requirements. During the design and implementation phases, ideas for improvements and new features were introduced. Here is a list containing a few of them:

1. Variable tracing history – Now the TracePropertyView only shows the last value of the desired variable. History about the value changes is needed. The history contains a trace number and a timestamp where the change happened and, of course, the value. The user can jump to the trace lines by clicking the lines in the history view and to the source code lines also if wanted.
2. External filter application – The user can define his own script or application to be used as an ASCII filter for manipulating traces. TraceViewer will forward traces to a standard input (stdin) of a specified application. The application can then do whatever it wants to the traces and print the results back to a standard output (stdout) where TraceViewer will read them and continue processing.
3. Logical operations to filtering – In the requirements there was only inclusive and exclusive filtering. Logical operations should be added to make more advanced filtering rules. For example, a rule saying "Hide traces containing rule1 or rule2 but not rule3" would then be possible.
4. Multiple file support – TraceViewer uses one file to store the traces received from the trace source. In Windows FAT32 file system, the maximum file size can only be 4 gigabytes [Microsoft, 2007] which will cause TraceViewer to stop receiving traces when the file exceeds that size. Data could, for example, be divided into multiple 100 megabyte files.
5. Plug-in architecture – To not make TraceViewer too big and hard to maintain, a plug-in architecture for it should be implemented. There is already a list of DataProcessors so it would be easy to provide an API for other plug-ins to use to add their own DataProcessor to the processing list.
6. Cooperation between Trace Builder and TraceViewer – TraceViewer and Trace Builder should work together more. For example, when the user adds more traces to the source code and compiles the software, Trace Builder should inform TraceViewer to re-load the XML decode file because it has changed. Also, traces in TraceViewer could be accessed from Trace Builder a similar way that the source

code line can now be accessed from traces in TraceViewer. The user clicks a trace in source code and the first occurrence of that trace is shown in TraceViewer.

7. Summary and conclusions

The reader should now have a general understanding of the design of TraceViewer and how it can be used to receive and manipulate traces as a part of the Open System Trace concept. This chapter shortly summarizes the key points covered in the thesis.

Tracing is defined as monitoring application execution in run-time. Tracing differs from debugging in the way that tracing will not stop the application execution and therefore is more real-time than debugging. Anyway, traces are normally used as a debugging tool to follow the application run path. Tracing has a performance impact on the device which might prevent the use of traces in time critical applications.

Open System Trace (OST) is a tracing concept involving several components used to apply a fully working tracing environment. OST consist of Trace Builder which is used to instrument traces into the source code of the software, OST Encoder / Decoder which is used from Trace Builder and TraceViewer to encode trace decode information on to the XML file and then generate the decode model from the XML file and decode binary traces into human readable traces.

TraceViewer is a plug-in for Carbide.c++ framework which is based on the popular open source development environment called Eclipse. Eclipse is originally developed by IBM in 2001 and is now managed by Eclipse Foundation. It is mainly known as a Java development environment and it contains excellent tools for Java development. Carbide.c++ is built on top of Eclipse and is a development tool meant for Symbian C++ development. There are four different versions of Carbide.c++ ranging from the Express version meant for beginners all the way to the OEM version meant practically for the device creation users.

TraceViewer can be used to receive traces from a smartphone. Traces can be filtered, colored, count, searched from, written to a log file and so on. TraceViewer uses Standard Widget Toolset (SWT) in its user interface components. SWT gives a native look and feel for the components in different platforms and operating systems.

The biggest challenge in the design of TraceViewer was that there can be millions of traces. The view which displays traces only contains a couple of hundred traces at a time and the scrolling of traces is done from the file generated from the trace source. This way the memory will not be a problem. Without trace manipulations, TraceViewer is able to receive and process 25000 traces per second with a powerful computer.

TraceViewer is bound to the OST Decoder which decodes binary traces received from the trace source. Because the OST Encoder saved metadata about the trace locations in source code files to the decode files it generates, the user can jump from trace to a corresponding source code line where the trace is defined. This is really useful and timesaving when the developer wants to follow program execution.

References

[CNET NEWS, 2006] Eclipse marks five years of expansion, http://news.com.com/2061-10795_3-6132997.html (2.25 miljoonaa)

[Eclipse SDK, 2007] Eclipse SDK: Viewers. Available as

[Eclipsepedia, 2007] Available as, http://wiki.eclipse.org/index.php/RCP_FAQ#What_is_the_Eclipse_Rich_Client_Platform.3F

[Forum Nokia, 2007] Forum Nokia – Carbide Development Tools for Symbian OS C++, available as http://www.forum.nokia.com/main/resources/tools_and_sdks/carbide_cpp/

[JDT, 2007] Eclipse Java Development Tools (JDT) Subproject, available as http://www.eclipse.org/jdt/

[JProbe Suite, 2007] Java Profiler for J2EE and Java Performance Monitoring with JProbe. Available as http://www.quest.com/jprobe/

[Microsoft, 2007] NTFS Preinstallation and Windows XP. Available as http://www.microsoft.com/whdc/system/winpreinst/ntfs-preinstall.mspx

[MSDN, 2007] Introduction to Instrumentation and Tracing. Available as http://msdn2.microsoft.com/en-us/library/aa983649(VS.71).aspx

[Royce, 1970] Winston W. Royce, Managing the development of large software systems. In: Proceedings of the 9th International Conference on Software Engineering, (1987), 328-338

[Symbian, 2006] Symbian Development Library, UIDs in Symbian OS Tools And Utilities, available as http://www.symbian.com/developer/techlib/v9.1docs/doc_source/n10356/uids/howtoobtainuids.html

[UsabilityNet, 2006] UsabilityNet: Rapid prototyping methods. Available as http://www.usabilitynet.org/tools/rapid.htm

[VEP, 2006] Visual Editor Project. Available as http://www.eclipse.org/vep/
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/jface_viewers.htm