Debugging Aids using Temporal Visualization

Supervisor: Bilal Alsallakh, Silvia Miksch

Peter Bodesinsky,Alexander Gruber,Dorna Nassseri

Vienna University of Technology
Institute of Software Technology & Interactive Systems (ISIS)

CVAST-TR May 2011

Authors: **Peter Bodesinsky,Alexander Gruber,Dorna Nassseri**

peter.bodesinsky@gmx.at,alex_gruber@gmx.net,dorna_ns@yahoo.com
http://www.cvast.tuwien.ac.at/

Contact: **Vienna University of Technology**
Institute of Software Technology & Interactive Systems (ISIS)

Floragaße 7/702
A-1040 Vienna
Austria, Europe

Web Address: http://www.cvast.tuwien.ac.at/

Abstract

In this report, we present a novel visualization-based method for debugging software programs and analysing their runtime behavior. One method aims at enabling developers to get an overview of variable stories by visualizing the values which these variables have taken over the course of the program. This helps in spotting errorous or unexpected variable assignments. The other method enables visualization of large arrays as a series or bar charts, which helps in gaining insight about the values in this array, such as value distribution, minimum values or maximum values. We implemented the methods as a plugin for the Eclipse Java IDE.

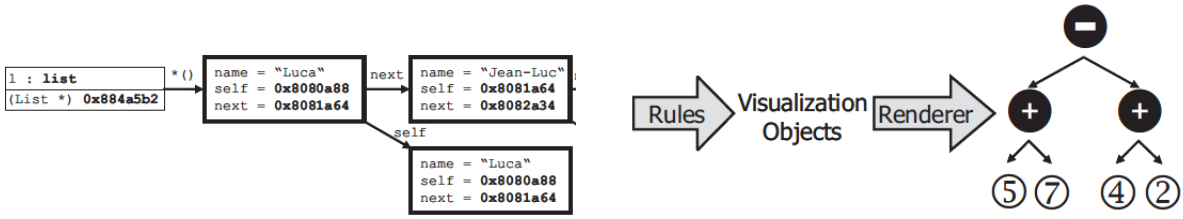
Chapter 1

Introduction

The goal of this project is to ease the debugging process with the help of a visual debugging tool implemented as Eclipse plugin, which is able to visualize variable histories during runtime. It is able to fetch read and write accesses to variables and to display the sequence of access/modifications events inside a view. In the course of our work, we examined different visualization techniques and tried to find the most useful approaches.

Our visual debugging tool is closely related to the general field of software visualization, where visualizations are generated from software or code. In a broader view software visualization can be defined as the "visualization of artifacts related to software and its development process", including not only code, but also other artifacts like documentation or bug reports [4]. Software visualization aims to help understanding, debugging and analyzing software. Different aspects of software can be visualized: The static structure can be taken into account, which means static aspects like data structures or relations between software modules, e.g. UML Class Diagrams. Software evolution is another aspect, i.e. code changes in course of development process can be visualized. The dynamic structure, which means software behavior, can also be considered and in this case the generation of visualizations requires run-time information. Visual debugging is an application area of dynamic software visualization. Visual debugging can be separated into two subcategories. It is possible to show either the program/memory states or to show program code (highlighting) and test results. Our tool visualizes memory states and belongs to the first category. Previous approaches in this area are mainly graph-based, for example interactive visual unfolding, which enables the user to inspect the current memory data structures by clicking nested boxes (Fig. 1.1a). The whole memory structure can be unfolded using traversal-based visualization, where memory structure visualization is generated according to rules (Fig. 1.1b). Further methods are memory graphs and reference patterns [4].

Many development environments, like Eclipse, offer traditional debugging aids, like breakpoints. They provide a simple way to stop program execution when certain conditions are met and can be very useful during debugging. But it can become tedious to check a high amount of variable changes, e.g. in loops with a large number of runs, and to keep track of the previous states. We offer a new method for visualizing these variable histories, and enable the developer to observe variable state history without interrupting the program. The debugging process should by simplified by providing a visual overview of discrete variable changes during runtime. Debugging programs with concurrent code is another application area for our approach. Our tool is able to emphasize the temporal relationships between read/write events of one or more variables and to visually ease the understanding of time-oriented code behavior. Furthermore we are dealing with the visualization of large arrays, because the inspection of large arrays can



(a) nested boxes

(b) render rules

Figure 1.1: examples for memory state based visual debugging techniques [4]

be tedious when using the textual listing of standard IDE debugging tools.

The report is structured in several sections. Section 2 (related work) gives an overview of state of the art tools related to our project. Section 3 and 4 (Visualizing Variable Histories, Visualizing Arrays) describe the main concepts and visualization techniques. Section 5 (Technical Details) provides deeper insight into technical aspects and section 6 (Evaluation) describes the evaluation process, followed by section 7 (Conclusion).

Chapter 2

Related Work

Several developments have been made in visual debugging that have some commonalities with our project.

Matlab has a feature called linked plots (since Release 2008a), and allows to connect a plot to a workspace variable [5]. The plot is automatically updated according to changes of the source data and can be used for debugging together with breakpoints (Fig. 2.1). Although this is a useful approach for static data (e.g. arrays), it does not support time-oriented plotting of variable histories and the debugging process is interrupted on breakpoint hit.

Debug Visualization Plugin for Eclipse visualizes memory states as directed graph [1]. Java objects are displayed as nodes and relations between them as edges. It allows choosing a variable in the Eclipse variables view and adding it to the graph using the context menu. This tool might be well suited to get an overview of memory structures and for program understanding, but maybe not the best choice for most general debugging purposes.

Another Eclipse-based debugging tool is JIVE, which also relies on graphs for visual debugging. It uses runtime information to generate dynamic object and sequence diagrams (Fig. 2.2). Object diagrams reflect the current execution state of a program with links between objects representing method calls. Objects and activated methods within them are represented as nodes. The sequence of method calls is visualized by sequence diagrams. It is possible to

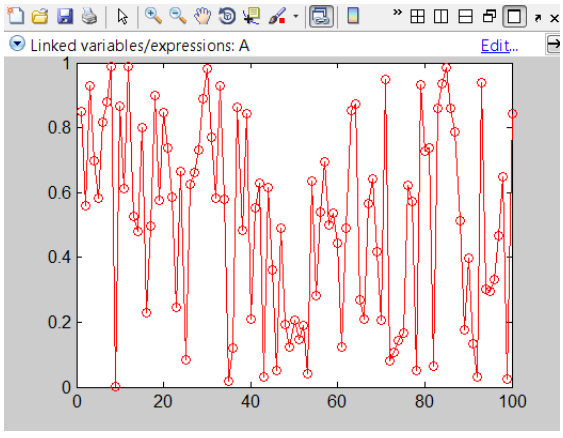
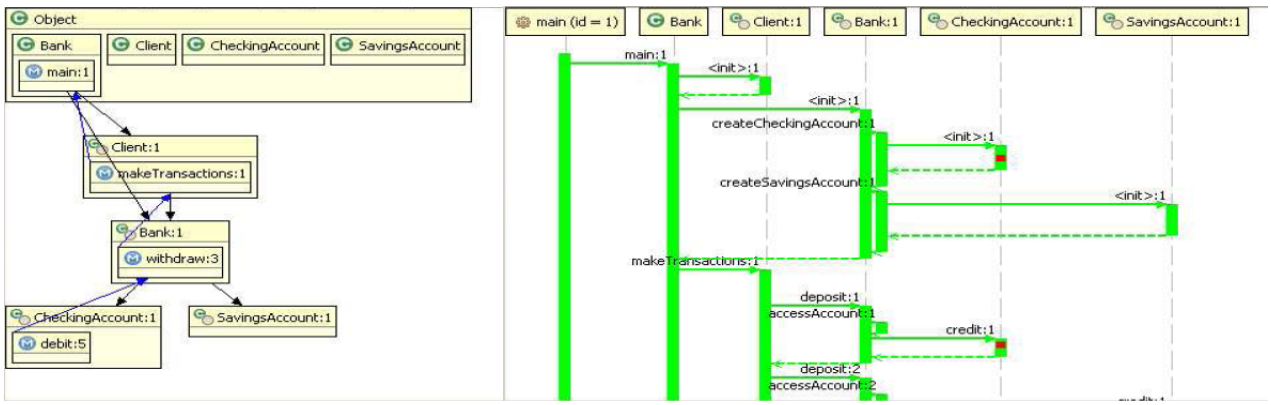


Figure 2.1: linked plot in Matlab showing a workspace variable [5]



(a) object diagram (b) sequence diagram

Figure 2.2: diagram types in Jive [3]

step forward and backward or to select an execution state by clicking on the corresponding point. A query can be executed on the program state history and the results are highlighted in the sequence diagram [3]. JIVE is a useful tool, especially for educational purposes, but for extensive projects the diagram might grow to a large extend and complexity.

Chapter 3

Visualizing Variable Histories

3.1 Avoiding Switching/Halting

The Eclipse IDE offers many debugging aids to step through a program and to locate errors. The most popular and the most widely used debugging aids are different kinds of breakpoints like line breakpoints, method breakpoints, exception breakpoints, watch points and so on. In practice this often means that software engineers have to step through source code line by line or jump from breakpoint to breakpoint, searching for any irregularities in their program. Sometimes this can be quite burdensome or even impossible, especially when programs are time critical or when they run in connection with different other threads, because the original behavior of the program cannot be reconstructed with debugging. It may also become likely that software engineers lose the overview over the source code of the program the more complex and the more bigger it becomes. These debugging problems constitute a gap which is filled by our plugin. Our plugin enables users to watch read or write accesses on selected variables without interrupting the observed program. This ensures that the program runs without interruption and makes it still possible for the user to track the functionality of the monitored program in detail and detect errors. This is especially helpful in conditions where hundreds or thousands of accesses occur on many different variables or objects. Through the visual presentation of the accesses which happen on surveilled variables, it also becomes easier for the user to keep an overview over variable state changes. To realize un-interrupted debugging with our plugin, the user has to mark the definition of a variable with a watch point and select it to be tracked from the plugin. By doing this, the underlying values of the variable are captured via a watchpoint listener offered by JDI (Java Debug Interface). The watchpoint listener records every read and write access on the variable and records it in a data structure ("WatchPointValue") which saves either the current or the new value of the variable depending whether the variable was read- or write accessed. Furthermore it records the timestamp when the modification happened and the name of the accessing thread as well as the name of the method which performed the access. Also the source code line, where the modification was executed is stored. Through saving the timestamps, the name of the accessing method/thread and the code location, the user can easily find out which value was assigned to the prospected variable at which time. He/she can also track at which point in the source code the modification happened and by which thread or method. This helps him/her to understand the actual code behavior of his program. While the observed program runs, variables are usually written, or read several times. In order to store all accesses on the surveilled variables, the recorded watch point values are gathered in lists which are assigned to each watched variable. These variable histories are then visualized in the user interface of the plugin.

3.2 Visualizing Value Histories

The view for visualizing variable histories consists of two main components. The first area on the left side of the view is a treeviewer component which lists all variables which were selected for observation by the user. The other one is the drawing area where the histories are dynamically drawn. The charts which are drawn are step charts which illustrate the changes on surveilled variables discretely. They consist of points which are connected by lines. This connection sets the variable accesses in a chronological order. These step charts are updated with every new access which is recorded for a variable. This means for the user, that he/she can either run the program without interruption and watch the value evolution on his/her observed variables, or he/she can still step through the program in debug mode, while every change is still logged and visualized. Figure 3.1 illustrates the layout of the plugin view. Our plugin differentiates between two kinds of variables. The first are the numeric or primitive variables. They contain values which are decimal or binary numbers and can therefore be ranked by their value. The other type are categorical or complex variables like Strings or Objects. In their case it is not possible to put the values of categorical variables into a vertical order. Thats why we simply visualize them in a straight line where each point on the line marks an access on the drawn variable.

3.2.1 Visualizing Primitive Datatypes

Numerical variables are often used to count something or to calculate huge amounts of data. Thereby they are likely to be accessed very often in a short time. If they are visualized in our plugin this results in a huge amount of drawn points which often form a recognizable patter (eg. calculated functions, or counter variables). By visualizing numeric variables, the user can examine the graphical pattern of the variable histories and find any irregularities in much less time than he would have needed with debugging through the source code line by line.

Figure 3.1: visualization of a numeric type variable in several instances

3.2.2 Visualizing Complex Datatypes

Unlike primitive data types, complex type variables can obtain null values. If this happens on a visualized variable in our plugin, the access where the variable is set to null and all read accesses which return a null value are generally marked in grey. So the user can see at one glance where and when the variable took null values. This is especially useful when he tracks

NullPointerExceptions. If the user examines numeric variables he can easily see which value they have because they are ordered vertically. In case of categorical variables it may become much more difficult to find a specific variable state beacuse they are arranged in a straight line. To solve that problem our plugin has an integrated instant text search. It allows the user to search for certain categorical values which may be obtained in the value histories. If a variable (it does not matter which, because the full text search looks through all values of all drawn value histories) ever contained a value which the user searches for, then it is marked in red. This can be seen for example in Figure 3.2.

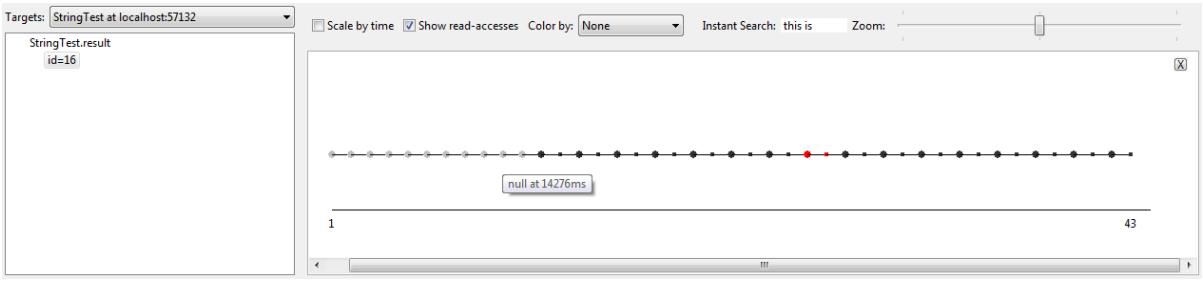


Figure 3.2: visualization of a categoric type variable

3.2.3 Horizontal Scaling and Zooming

The user can check if he/she wants to scale the abscissa in state changes or in a time dependent manner. He/She can do this by checking or unchecking the "Scale by time" checkbox in the upper part of the plugin user interface. If he/she leaves it unchecked, the variable accesses are still drawn in their chronological order, but the horizontal distances are constant and do not represent the time which has passed between each access. If he/she checks the checkbox, the variables accesses are drawn in dependency to the time which has elapsed since the initiation of the tested program. This enables the user to examine the program performance and he/she can also compare between variables and find out whether the chronology of accesses on them is correct. If the user visualizes variable histories scaled by time, there may be graphical clusters of accesses where lots of accesses happen in a short period of time. To look at these clusters in more detail, the plugin includes a zoom function to increase the zoom level or to reduce it. This helps to look at variable state changes in detail on the one hand, and to keep an overview over the variable history on the other hand.

3.2.4 Visualizing variable behavior in several instances

The plugin is capable of recording variable histories in several JVM instances. This means that the user can run the program in a first instance where he observes his variables of interest. Then he can start another instance of the same program, watching the same variables. He can then select both instances in our plugin (by using the combo box on the left upper corner) and visualize both courses of the selected variables. This makes it easy to watch variable behavior under different circumstances for example by feeding the tested program with different input data. He can also decide whether he wants to see only write accesses or whether he also wants to see read accesses. This is done by checking the "Show-read access" checkbox in the upper area of the interface. All read accesses are shown as small squares, while the write accesses are illustrated as circles.

3.2.5 Inspecting Multiple Accesses

An important feature of our plugin is that the user can see graphically which method or which thread has written or read into or from a variable. This is realized by assigning different colors to each thread and to each method which is run in the tested source code. When opening prospected variable, the access is visualized in the colour of the method in the step chart. Referred to Figure 3.1 there is an integer variable visualized which is examined in two different instances. The user can select via the combo box at the upper border of the plugin interface whether he wants to see which thread accessed a variable or which method used it. If he/she selects the method variant he/she can also choose the stack level (by changing the slider next to the combo box) of the accessing method. So he/she can not only see the method in which the variable is modified but also the method names which are engaged in accessing the method to alter the observed variable. This is a useful feature to detect for example phantom problems, where a variable is mutually changed by different threads or methods and may cause errors in the further performance of the program.

3.2.6 Inspecting exact variable states and code locations

Another important feature is to inspect the exact state which the variable took on at a certain timestamp. If the user moves with the mouse over a painted read or write access, the current value of that access is shown together with the timestamp. Furthermore if he double clicks the point, the user can hop directly to the location in the tested program where the access happened. So, he can retrace what happened when, where and to which variable.

Chapter 4

Visualizing Arrays

Every array has a data type or underlying data structure which is assigned during the definition of the array. It consists of at least one or several field elements which are either "normal" variables or data structures who may contain further sub attributes. Searching these field elements in Eclipse during debugging is an awkward task. It is tedious to check every item in the array, especially when the field elements of an array contain many sub attributes, or when the array has a large size. In Eclipse, the contents of an array are displayed in textual lists which can be expanded and explored value by value. To keep an overview of all values or of the array structure will become more difficult the more elements the array contains. This is the point where our plugin takes effect. Unlike the visualization of variable state changes over time, our plugin visualizes the contents of an array at a certain timestamp (snapshot) and gives an overview about the structure. It further analyzes the array contents and creates bar charts which visualize the distribution of the values stored in the array. This is not only possible with simple elements but also with the sub attributes of data structures that are saved as elements into the array. By that way, the values of certain sub attributes can be directly compared with each other.

The view of the array user interface is similar to the view of the value histories. It also contains two main components, where the left one is a treeviewer component and the right one is a drawing area. The treeviewer component enlists the type of elements (not the elements itself!) from the arrays which were selected for observation. If the elements have sub attributes, their names are also listed and in case one of them is selected, the corresponding field elements are visualized in the drawing area. They can either be shown as value-series, similar to the visualization of single variables or they can be drawn as bar chart. In which manner they shall be drawn can be decided by the user through selecting the appropriate option button in the upper area of the plugin view.

The plugin contains a zoom function for the array view too. This enables the user to zoom into a point of interest or to zoom out to gain more overview of the displayed element values. Each attribute or each series of elements is painted in the view as diagram, which can be dynamically added or removed from the view. The array view shows the exact value of an element together with its index in the array, as soon as the user moves over the point with the mouse.

4.1 Series

If the user has decided to show the selected array elements as value series, they are visualized depending on the underlying variable type. If the type is a numeric type, the element values are drawn as line diagram. The points of the element states are connected with each other and

ordered vertically according to their value. But in this case that does not mean that they are horizontally in a chronological order. They are drawn according to the increasing index of the array. The links between the points are only painted to increase the readability of the chart. On the one hand it is easier for the user to determine unusual variations between the variables, and on the other hand he/she is able to estimate an average value as well as it is easier for him/her to locate minimum or maximum values. In case the underlying variable type is a categorical type, the element values are visualized as scatter plot, because the elements cannot be ordered vertically according to their values. There is also no need to connect them with lines because they have no direct relation to each other. They are also drawn in a horizontal line so there are no statistical characteristics detectable. To simplify the location of specific states which the user is interested in, the array view also contains an instant text search. If a point in the view area is found which matches the search string, it is highlighted in red. That makes it easier for the user to find certain values fast, especially when he is interested in the visualization of huge sized arrays. Figure 4.1 shows an illustration of a value series visualized array, which consists of "student" data structures. These data structure contains three attributes name, age and academic year. In the image the names (categorical variable) and the academic years (numeric variable) of each student are displayed. This poses an easy way for the user to inspect all values or sub values of an array quickly, without clicking through lists of data structures, and it is still possible for him to keep an overview over all appearing values.

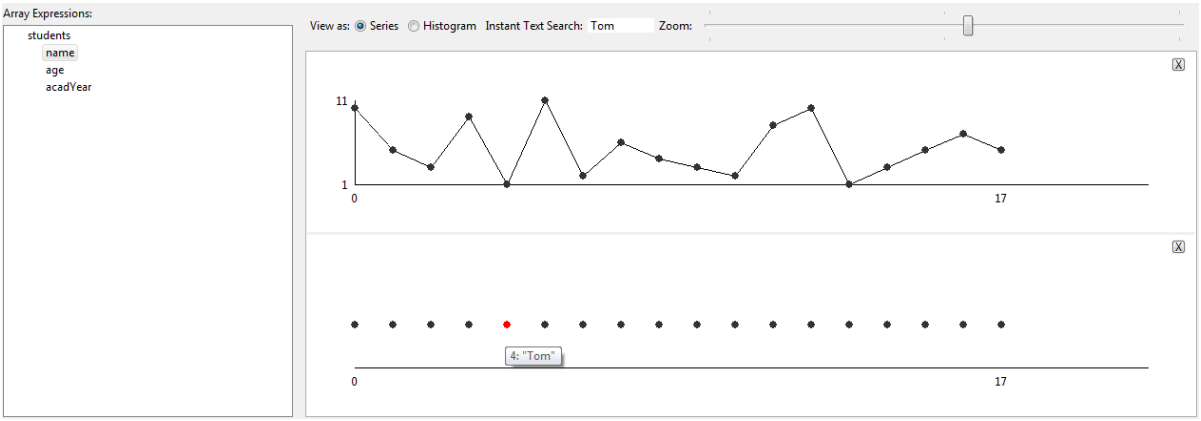


Figure 4.1: visualization of element sub attributes in an array

4.2 Bar Charts

In case the user decides to view a selected series of array elements in form of a bar chart, a graphical bar is assigned to each different value that occurs in the array. Depending on the number of value appearances, the bar increases it's height relatively to the other bars. This enables the user to set the frequency of each value in the array in relation to each other. He/She can determine immediately which value appears most in the array and how often other values appear in relation to the most frequent value. The number of appearances is noted on the ordinate axis while the different values are displayed on the abscissa. Figure 4.2 shows an example of an integer array which has a size of one thousand elements. All elements and their frequency are displayed as bars, which enables the user to see immediately which values appear in his/her large array, how often they appear and which one appears most. So the bar chart is a useful facility to gain a quick overview over the array occupancy.

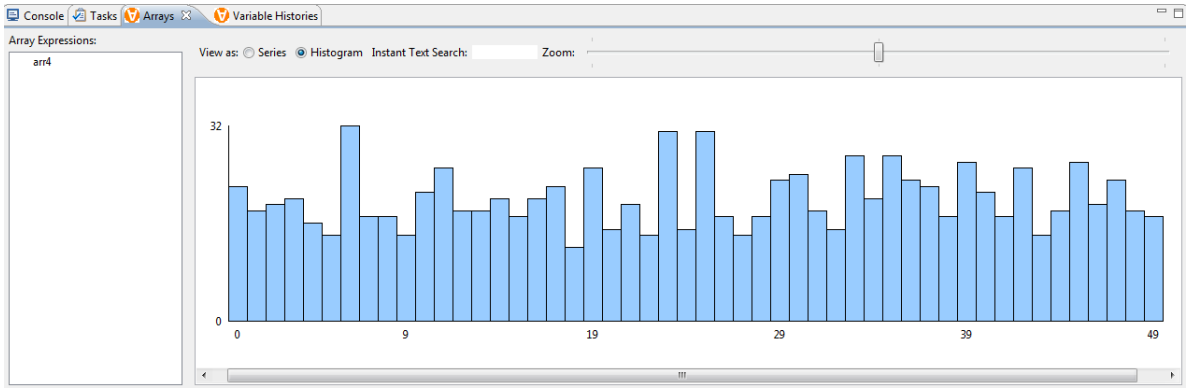


Figure 4.2: visualization of an integer array as bar chart

Chapter 5

Technical Details

Our visual debugging tool is implemented as Eclipse plugin and makes use of Eclipse RCP (Rich Client Platform) architecture. Eclipse can be regarded as plugin framework, and is based on a Platform Runtime at its core, which loads and executes plugins. Each Eclipse application and even the basic Eclipse SDK itself are based on this plugin model. A plugin can use extension points of other plugins or provide own extension points for other plugins. The default Eclipse Platform (Eclipse SDK) is composed of subsystems, each of the subsystems is implemented with the help of one or more plugins. The major subsystems are the Platform Runtime, Resource Management, Workbench UI (defines extensions points to add GUI entries and provides GUI toolkits), Team Support, Help System, Debug Support and JDT (Java-specific part) [2].

Several extension points are used by our project, extension points of the subsystem/package `org.eclipse.ui` are used for UI integration into Eclipse (menu-commands, Eclipse-Views). A java specific JDT extension point named `org.eclipse.jdt.debug.breakpointListeners` is used to get the required data from the debugging process.

The software architecture of our project is based on the MVC pattern, and therefore separated into Model, View and Controller classes. General classes for plugin lifecycle and watchpoints installation events are defined in the package visdebugger. Classes describing the user interface (View) are located in the package `visdebugger.view`. The package `Visdebugger.control` contains the controller classes responsible for drawing and reacting to user input. The `Visdebugger.model` classes define the data model and are responsible for model event notifications, i.e. notification of listeners in case of watchpoint accesses. Package `visdebugger.eclipseuiactions` contains special classes for Eclipse UI interaction. See Fig.5.1 for an overview of the collaboration between the main system components.

5.1 Data

5.1.1 History

In order to draw the state history for a variable, data has to be fetched from the debugging process. Eclipse supports multiple types of breakpoints [6], among them the best known type is a line breakpoint, which is hit when a particular line is executed. Other types include exception, class load and method breakpoints. To fetch the data for the views, a special type of breakpoint called watchpoint is needed, which is declared on a field (variable) and reacts/breaks execution on variable access or modification events. The extension point `org.eclipse.jdt.debug.breakpointListeners` is used to gather information about

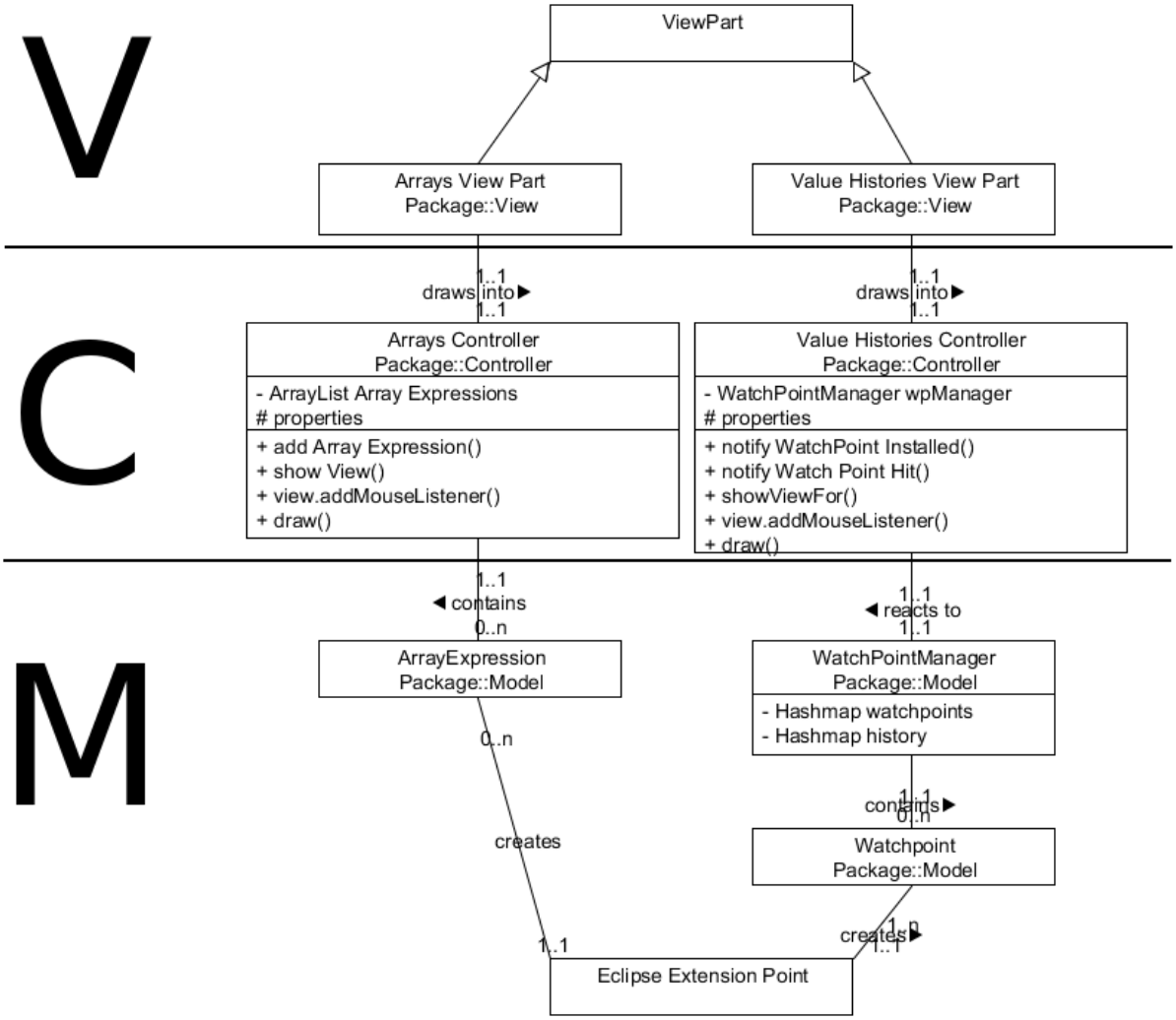


Figure 5.1: class diagram illustrating the dependencies between the main system components

breakpoint installation and removal events. If a new watchpoint is declared, an event listener is installed and access/modification events are managed by an instance of WatchPointsManager, which also holds and gathers information for all variable histories.

5.1.2 Array

The data for the visualization of static arrays is gathered with the help of selection events (`org.eclipse.ui.popupMenus` extension point). If the user clicks the corresponding menu entry in the Eclipse variable view or the code editor view, the selection is analyzed and the value and name of the array is extracted. After this a new ArrayExpression is created, which holds all necessary information for the view.

5.2 Views

The views are integrated in the eclipse UI using the `org.eclipse.ui.views` extension point. JFaces and SWT are used to build the GUI functionality and we decided for the SWT

drawing API in the class `org.eclipse.swt.graphics` to perform the drawing of the diagrams. `ValueHistoryViewPart` and `ArrayViewPart` define the basic GUI elements and layout of our Eclipse views, while the logic for interaction is defined in the controller classes (`ArraysMainController`, `ValueHistoriesMainController`). The diagrams themself are code objects (`ArrayValueView`,`ValueHistoryView`) integrated in the parent Eclipse view and are managed inside `ArrayViewsManager` and `HistoryViewManager`, which are responsible for creating the suitable diagrams and refreshing them. `ValueHistoriesMainController` holds an instance of `WatchpointManager`, gets notified on access/modification events and uses `HistoryViewManager` to refresh the diagrams. Diagram drawing logic and viewing transformations are implemented in the corresponding controller classes, common functionality is implemented in abstract classes (`AbstractArrayController`, `AbstractValueHistoryController`), while special functionality is defined in classes like `NumericArrayController` for integer types histories or `BarChartArrayController` for integer arrays.

Chapter 6

Evaluation

We decided to discuss and analyze usability issues inside the project team. Due to short iteration cycles and joint review, we were able to improve our visualization techniques in the course of the project. Extensive user studies will be performed in the future and may also give further insight into possible application scenarios of the tool.

When analyzing large arrays scalability is another important issue, our tool is able to handle and visualize arrays with thousands of values in general. The value series view can grow large for huge arrays, which might make it necessary for the user to set the zoom factor to a low value in order to maintain an overview of the array. In this case the values series can collapse to clusters, i.e. it might result in multiple data points occupying one pixel. Although it is not possible to see the exact relations between adjacent array values anymore, it is still feasible to get a quick impression of the value distribution, without losing the complete information of the value index locations. Additionally the minimum and maximum value can still be identified (Fig.6.1). The histogram view scales well with huge arrays, and is suitable to get a quick overview of value distributions, especially in cases where visual identification of value index location is not so important.

Even for large arrays the performance is still sufficient for interactive visualization. Only if the array size exceeds an upper size limit, performance can drop in a noticeable way. But this happens in case of arrays containing tens of thousands of values, which are typically analyzed using external software tools.

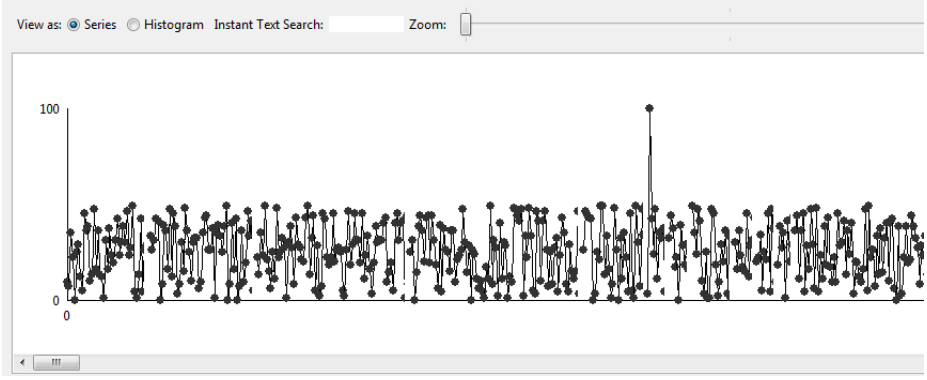


Figure 6.1: although the array values are building clusters, minimum and maximum can still be clearly identified

Chapter 7

Conclusion

We have presented a tool for visual debugging, designed to provide developers with a compact and easy to use GUI integrated into Eclipse IDE. Our tool presents a new approach to visual debugging, by giving a quick visual overview of variable read and write accesses. It enables to observe variable state changes without interruption, while temporal relations are preserved. Especially for time critical concurrent code and a huge amount of variable accesses this can be very helpful. Multiple chart views with a common time origin support debugging of complex programs and allow seeing correlations between different variables. Coloring the points based on different semantics like method names or thread names is another feature, which helps debugging concurrent code segments. In order to differentiate between read and write accesses, the read accesses are drawn as squares and the write accesses as circles. By the means of these features some patterns and error types can be identified visually, without the effort needed in traditional debugging tools. Visualization techniques include step charts for numeric types, which show the values in vertical relation to each other, and line charts for non-numeric types.

Visualization of large arrays in a separated Eclipse view makes debugging more comfortable compared with the standard IDE tools, which only allow inspecting the array values using a textual listing. Our tool is able to visualize array values as series, either as connected points for numerical array data, or without vertical scaling in case of categorical data. Bar charts on the other hand can give a quick overview of content distribution of large arrays by counting value appearances. Furthermore our tool supports zooming and panning and instant text search in both views.

Ongoing evaluation and improvement are planned for the future, because development is not already finished. Feature extension and code cleanup/refactoring will be performed in coming software development cycles. A next step would be setting up an eclipse update site to make distribution and installation as easy as possible. Global availability of the source code is another goal, because it gives developers the opportunity to add functionality and own contributions to the project. Additionally extensive user studies are planned to get usability feedback and receive input for possible improvements.

Bibliography

[1] Debug visualization plugin. http://code.google.com/p/debugvisualisation/. version 04.2011.

[2] *Eclipse Platform Plug-in Developer Guide v3.1.*

[3] Jeffrey K. Czyz and Bharat Jayaraman. Declarative and visual debugging in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM.

[4] Stephan Diehl. *Software Visualization.* Springer, 2007.

[5] matlab documentation. Making graphs responsive with data linking. http://www.mathworks.com/help/techdoc/data_analysis/brh7_h0-1.html. version 04.2011.

[6] Prakash G. R. Types of breakpoints in eclipse. http://www.eclipse-tips.com/tips/29-types-of-breakpoints-in-eclipse. version 04.2011.