SomeTitle

Tørresen, Håvard

Supervisor:
Trætteberg, Hallvard

September 30, 2014

Abstract

Background:

Results:

Conclusion:

Acknowledgements

Contents

1 Introduction **2**

2 Task Description **3**

3 Prestudy **4**
- 3.1 Methods . . . 4
- 3.2 Existing Tools . . . 4

4 Enhancing JIVE **6**
- 4.1 Features . . . 6
  - 4.1.1 In Development . . . 11
- 4.2 Shortcomings of JIVE . . . 12
- 4.3 Suggestions for improvement . . . 12
- 4.4 Implemented changes . . . 17

5 Conclusion **18**

Glossary **19**

Bibliography **19**

List of Figures

1 The JIVE perspective in Eclipse . . . 6
2 A contour diagram generated by JIVE . . . 7
3 A sequence diagram generated by JIVE, while running an instance of MMI-Exercise 1 . . . 8
4 The sequence diagram right click menu . . . 9
5 Normal and collapsed section of a sequence diagram . . . 10
6 The JIVE search panel . . . 11
7 Crude comparison of suggested changes to the contour diagram . 14
8 A section of a larger sequence diagram, showing method calls crossing several unrelated lifelines . . . 15
9 An example of how Figure 8 might look in the isolated view . . . 16

List of Listings

1 Introduction

New students may find programming in general, and object oriented programming in particular, difficult. The understanding of computers, and how they work, is not given a lot of focus in the school system, and only a few students have actually tried learning programming before coming to the university. With little or no previous knowledge, it can be hard to understand the concepts, and to get a mental model of what is going on when writing a program. Especially code not written by themselves, for example exercise frameworks, and code generated by various tools, can be hard to get a good understanding of. Traditional debuggers are not necessarily helping when detecting a runtime error, and often significant amounts of time is spent searching for the cause of a bug, instead of actually fixing it [Ko and Myers, 2006]. Tools that present the state of a program in a simple, visual way may help to understand how a program works, and how its components interact with each other.

During the second year of the computer science, and the informatics studies at NTNU, there is an increased focus on projects and more complex software. Among the mandatory courses of these studies, we find the course Man-machine-interaction (MMI). This course handles topics related to creating graphical interfaces, and specifically how to implement a Model-View-Controller-architecture in the Java Swing framework. The learning goals for the MMI-course are as follows:
Introduce the student to concepts, methods and techniques for designing man-machine-interfaces, knowledge and skills in object oriented construction of graphical window-based interfaces.

Will such tools be useful in helping students to reach the learning goals of MMI and other beginner-courses? Are existing tools good enough? If not, can something be modified to better fit the purpose?

2 Task Description

Examine whether tools like JIVE can be used to aid students with their understanding of programming, and software structures. Attempt to identify changes that may be necessary e.g. default configuration, handling of visual models, performance. Which changes are actually possible to do? How to do them?

3 Prestudy

3.1 Methods

There are several ways a debugger can aid the programmer beyond just showing the current state of a program at a breakpoint chosen before running. For a fresh programmer, either in general, or at a certain project, the most useful method is probably to generate diagrams that visualize the current state, and the path of execution. I.e. some form of object-, and/or sequence–diagram. Such diagrams can make it easier to get an overview of a programs current state, see the contents of objects and how they relate to each other, and to understand how the various components work together.

In order to generate the diagrams, the tools can analyze both the source-code, and an execution trace, depending on the type of diagrams to generate. For a general overview, a class diagram, showing how object types are composed, can be generated by simply analyzing the source code. But in order to get a more specific view, for example of how the different components interact, an analysis, either in real-time or from an execution trace, of a running program is needed. Such an analysis can be used to generate object diagrams that show how objects interact and what values they have at a given time. One can also get a sequence diagram, showing what methods were called, and in what order.

Execution traces can also be used to enable backwards stepping of program execution. Stepping back in time allows the user to not only see the failure state of a program, but to go back and see what caused the problem, instead of adding a new breakpoint and running the program again. Each reverse step can be fairly cheap, but it may still be impractical to make large jumps in the execution history.

One can avoid the potential disadvantages of manual backstepping by using queries instead. Queries enable the user to asks the debugger about the current and earlier states of execution in a simple way. The debugger then does the work of finding what was asked for, instead of the user manually searching through the program states.

3.2 Existing Tools

There currently exists several tools that provide one or two of the methods mentioned above. GDB offers a tracing environment, but due to its command-line interface it is not necessarily easy to use on its own. The Trace Viewer Plugin [Kranzlmüller and Klausecker, 2009] for g-eclipse, uses a trace to generate visualizations of the program execution, and thus makes it easier to understand, but is designed for massive parallelism, and may not be very useful for understanding smaller programs. Whyline [Ko and Myers, 2009], and the Trace-Oriented Debugger [Pothier et al., 2007] also utilize execution traces, but use them to enable querying, instead of providing visualizations. Additionally, Whyline exists only as a separate application, and does not integrate into any IDE.

JAVAVIS [Oechsle and Schmitt, 2002] provides visualizations in the form of UML-diagrams, but does not provide any debugging features. Code Canvas uses an interesting way of visualizing an entire project, everything from source-code to design documents and diagrams are layered onto a large canvas, allowing easy navigation between various elements, but is restricted to Microsoft Visual Studio. Jinsight [Pauw and Vlissides, 1998] is a powerful tool built by IBM, supporting both tracing and visualization. However, it is restricted to z/OS and linux on system Z, preventing most people from using it.

JIVE seems to be the only tool that utilizes all three methods, as well as being freely available as a plugin for eclipse, making it easy to install and use. During program executiion, Jive generates a contour diagram [Jayaraman and Baltus, 1996], and a sequence diagram. Combined with an execution trace, it allows the user to jump back and forth in the execution, and have the diagrams updated accordingly. Querying is supported through the JQL, and is accessed trough a simple graphical interface with templates for the most common queries, as well as a text-field allowing the user to write any kind of query.

Due to all the extra work being done when using jive to debug a program, the performance is not always acceptable. For small non-interactive programs, the added waiting time may not be a problem, but larger programs may suffer from a significantly longer execution time, and even simple interactive programs can use up to a second to respond to input on a fairly powerful computer.

4 Enhancing JIVE

In this section I will first explain the various features of JIVE, before identifying any shortcomings, and finally suggest improvements.

4.1 Features

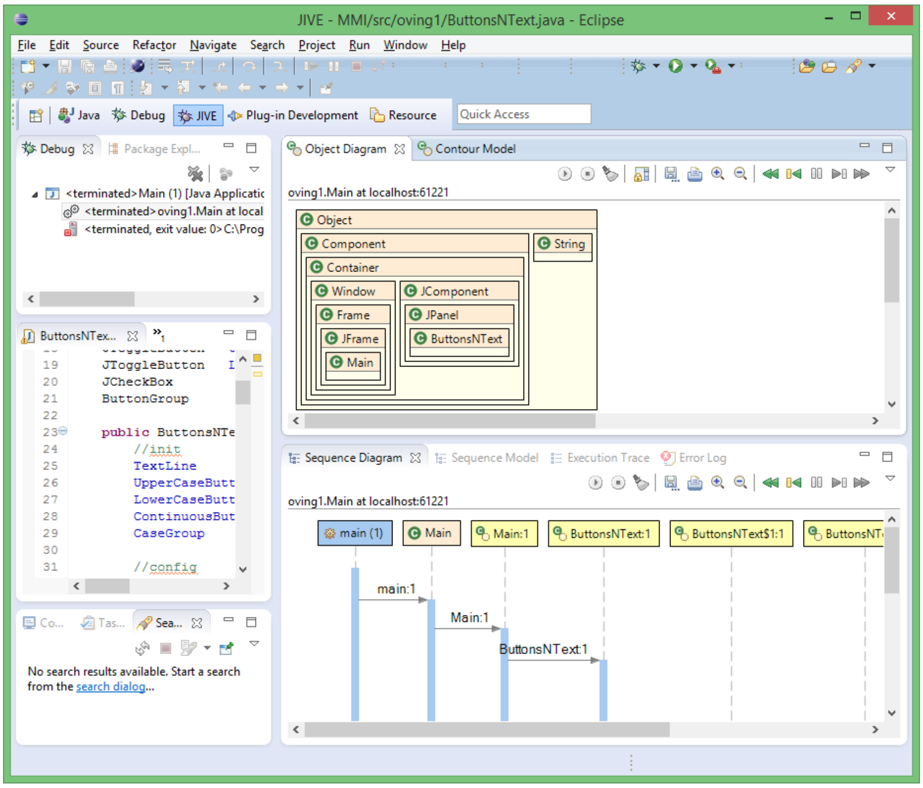


Figure 1: The JIVE perspective in Eclipse

As mentioned in the prestudy, JIVE installs as a plugin in the eclipse IDE, adding another perspective in the environment shown in Figure 1. The default views shown in this perspective are the "object diagram" and "sequence diagram" views, making the diagrams generated during debugging JIVEs two most apparent features. The diagrams are updated according to the current state of the debugged program, so that the backtracking functionality allows you to see the entire execution graphically step-by-step. The other views provided by JIVE are as follows: "contour model", "sequence model", "execution trace" and "search".

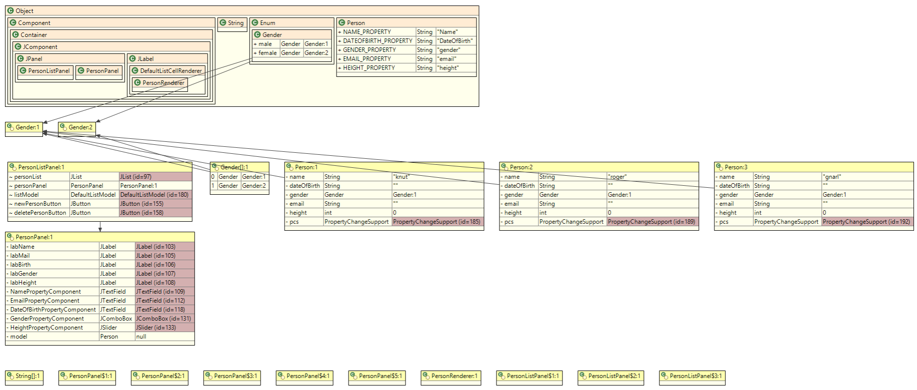


Figure 2: A contour diagram generated by JIVE

The object diagram-view shows the current state of the program by using a contour-diagram. Contour-diagrams, as shown in **??**, are based on an old technique to give semantics to Algol-like languages. The basis has been extended to support modern concepts, such as object-oriented programming. Objects are represented by a box, or contour. Within the contour, the objects variables are shown, with name, type and value. The contour also uses arrows to point at other contours that are related, e.g. an other object representing the value of a variable, or an enumerator. Inheritance is shown by putting the contour of an object within the contour of the extended object. Object instances are kept separate from the contours representing inheritance, but will have relational links when necessary. Method calls are also represented in the diagram, in their own contours, linked to the calling object. JIVE offers to hide some of the information, such as inheritance or the composition of objects, in order to make the diagram smaller, and easier to read. Something that can be especially useful when working with larger programs, with many objects and relations. Visibility is aided by the use of colors to highlight specific elements of the diagram, such as variables bound to objects, and method-calls.

Figure 3: A sequence diagram generated by JIVE, while running an instance of MMI-Exercise 1

The sequence diagrams, shown in Figure 3, are fairly standard, with threads and objects represented by boxes on a row at the top, each with a vertical life-line. The actual sequence is shown with a thicker lifeline, with arrows representing method-calls. In order to differentiate the threads where the execution is happening, the sequences are colored with the same color as the thread- box the sequence originates from, regardless of witch objects and methods are involved. In Figure 3, one can clearly see the two colors representing the main- and the AWT-event-thread, and how elements in the diagram are colored by their parent thread.

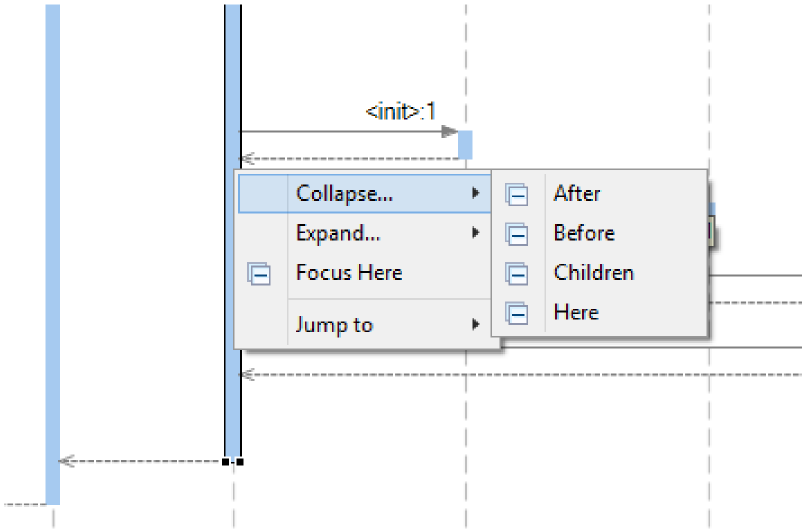
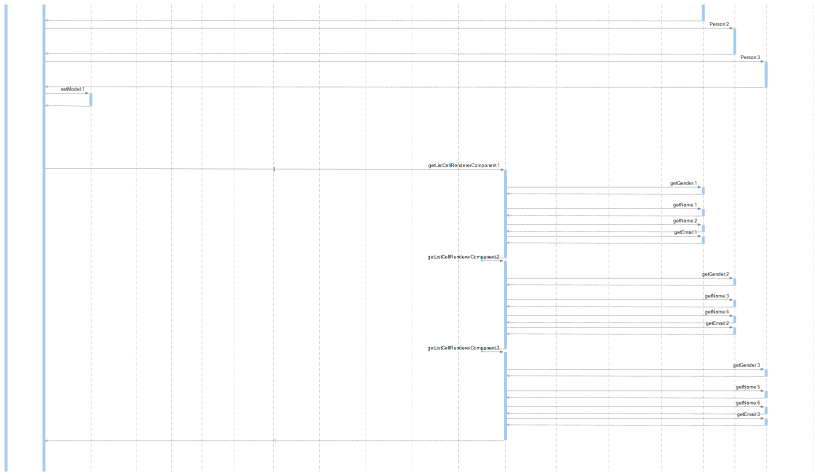
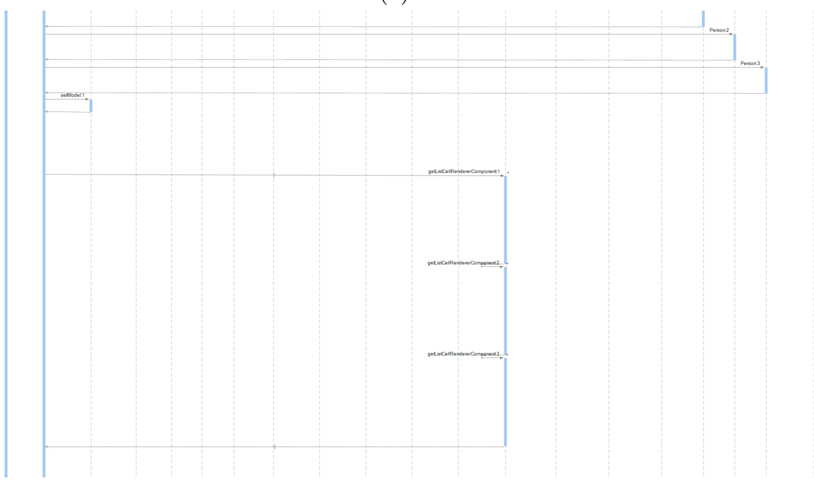


Figure 4: The sequence diagram right click menu

Right-clicking on a lifeline offers the ability to collapse method-calls originating from that lifeline in order to hide unnecessary information, shown in Figure 4. The collapse-menu offers four options when used on the lifeline of an object: after, before, children and here. Before and after collapses lifelines that occurred before or after the selected event, at the same depth in the sequence-tree. Children collapses all events that are children of the selected event, while here collapses the selected event.Right clicking on an object instance at the top of the diagram also offers to collapse that objects entire lifeline, the result of which can be seen in Figure 5. The same options are of course available for expanding collapsed elements. Right-clicking also offers to set the execution-state via the jump-option, updating the contour-diagram and -model to show that state.



(a) Normal



(b) Collapsed

Figure 5: Normal and collapsed section of a sequence diagram

Both of the diagrams can be saved as a high-resolution image at any time, so that one can look at diagrams from earlier runs, instead of being forced to view them through JIVE. This also helps to visualize any changes made to a program, and to see what the effects are on the program flow. They also share the ability to zoom in and out, further helping with the handling of larger diagrams.

Closely related to the diagrams, is the ability to quickly jump backwards and forwards between execution states. This is enabled by the trace-log, containing an entry for every single event that occurs during execution of a program. Each event is assigned an identifying number, in ascending order, and information about thread, type, caller, target, and the location of the source-code is stored. The log is used as the basis for the models that make up the diagrams, and can be saved as both XML- and CSV-files for later use. As mentioned in the Prestudy, logging every event has a significant impact on run-time performance, limiting the size of the programs that can be used in a meaningful way. On the other hand, it allows the almost instant jumping between recorded states, as opposed to techniques that save a snapshot at predefined intervals, requiring the program to be run from the snapshot-state to the desired one, even if it is just a single step backwards.

The model-views each display an alternate view of their respective diagrams.They show the data-model representing the diagrams in a hierarchical structure, much like the organization of files and folders. Right-clicking on an event in the sequence model allows you to set the execution state to that event, and have the diagrams updated accordingly. Clicking on elements in the contour-model allows you to inspect the values of objects and their variables.

Finally, the trace-log enables the use of queries to search for specific events in the execution. The queries are presented through a new tab in the eclipse search-window as seen in Figure 6, easily accessible through the search-view, and comes with several pre-defined templates to simplify searching. For example, searching for when a certain variable gets a certain value, only requires the user to specify the variable-name, its parent, and the value.

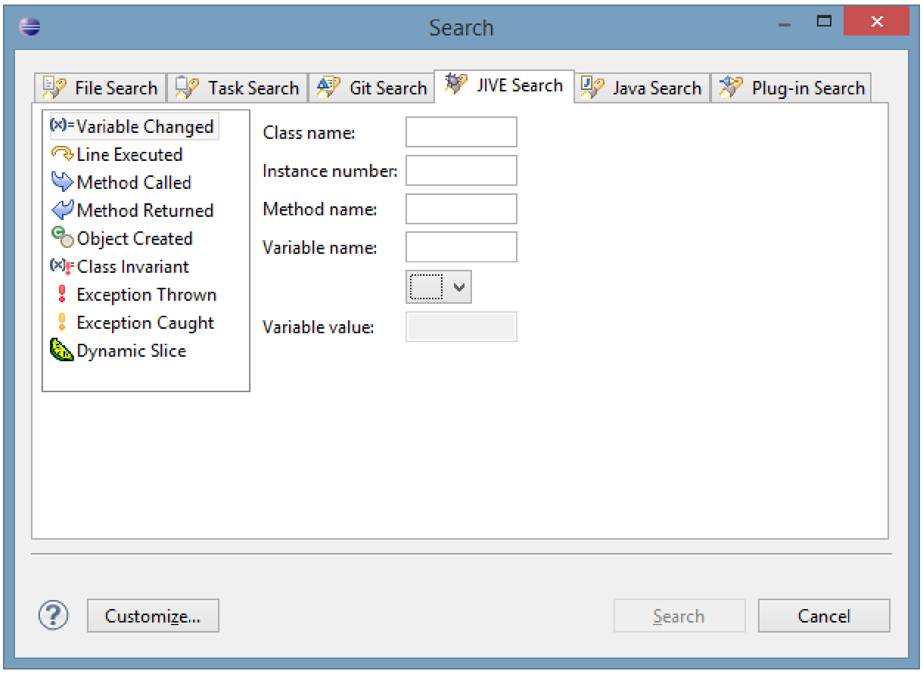


Figure 6: The JIVE search panel

4.1.1 In Development

Regex folding and state-diagrams go here.

4.2 Shortcomings of JIVE

Even though JIVE offers several useful features, there is still room for improvements, both for general use, and more tailored towards development of graphical interfaces, as is the focus of the MMI-course.

The diagrams are possibly the most useful feature of JIVE, showing a lot of information in an understandable way, and offering some useful interaction. But there is still room for improvement: Inner types are displayed with their automatically generated, and fairly anonymous, name unless given a proper name in code. By default, most of the classes defined by the JRE itself are ignored, omitting potentially important information in GUI-applications. For the sequence-diagrams, it is naturally not important no show the internal behavior of every object, but they are also hidden in the contour-diagram. Related to this, is the lack of visual connections between standard-objects, and for example instances of listeners added to them. More related to the target group for this project, is a lack of differentiation between the different object-types in typical graphical architectures. Especially MVC, whitch is a major focus of the MMI-course, has certain distinct types that are more important than others.

The sequence-diagrams can quickly become huge and hard to navigate, and the zooming function has very few levels outward, as well as leaving all text to small to read when zooming out. Additionally there does not seem to be any way to vertically compact the sequence-diagrams, making them unnecessarily long in cases where calls to ignored or hidden methods are being made. For instance when horizontally collapsing large parts of the diagram, leaving the parent lifeline at the same length as before, as can be seen in Figure 5b. Some of the papers on JIVE mention regex-folding,that could be used to substitute a series of events with a single new event, but it is nowhere to be found in the latest version of the plugin.

Another potential for improvement is the process of stepping through recorded states, which currently requires manual interaction for each step, making complete replays of an execution infeasible due to the massive number of events recorded even for simple programs.

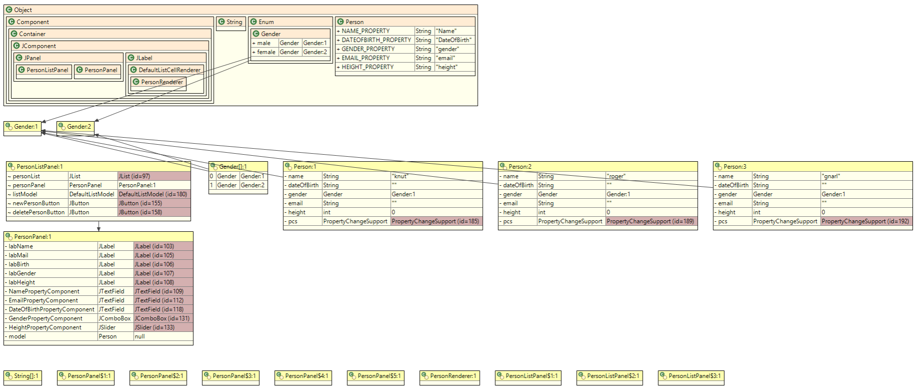
The search-view, while useful, is very strict in terms of what it looks for. For example, searching for the creation of an object requires its full name, including packages, so searching for creations of JButton-instances would require a search for javax.swing.JButton.

4.3 Suggestions for improvement

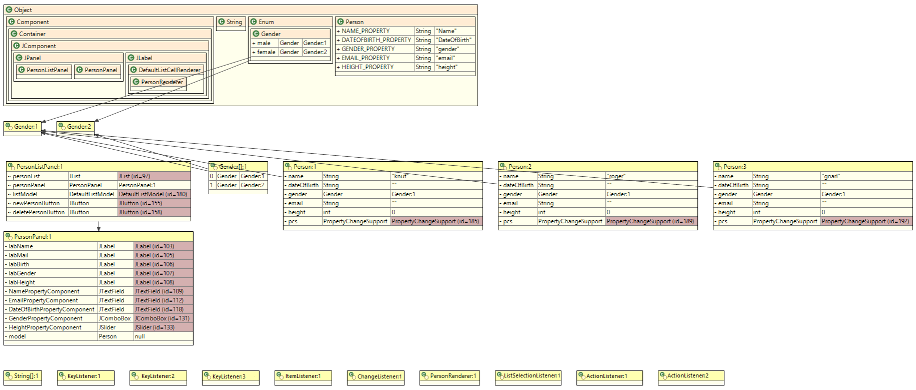
Based on the mentioned shortcomings, the following improvements are suggested.

The ability to detect an inner type with a generic name, and display its parent type in the diagrams instead of its own. This will make, among others, listener-heavy programs much more understandable as one can see what kind of listener each object is, instead of guessing, based on when it is invoked in the sequence-diagram. Further helping the same situation, is to visually link listeners to the objects they are added to, in the contour-diagram, making it clearly visible which object is being listened to by the different listeners. A crude visualization of these two changes, compared to the original, can be seen in Figure 7.

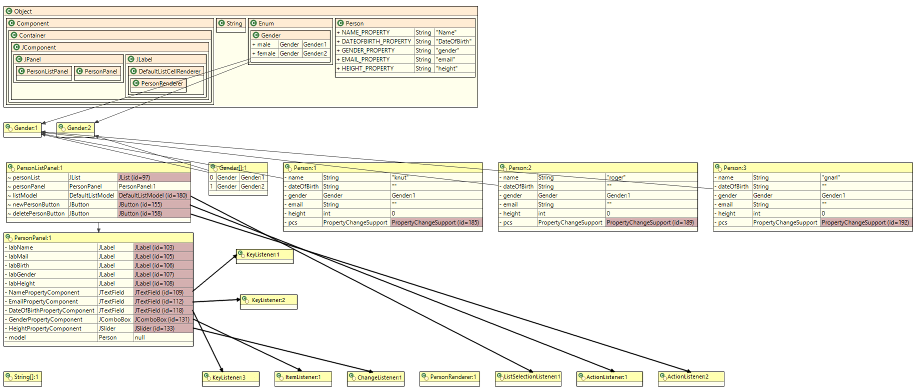
Finding ways to highlight the different parts of e.g. an MVC-architecture, making it clearly visible which objects make out the models, views and controllers, may further help the understanding of a program. But such highlighting must be balanced in order to not create a visual chaos of different colors. Adding symbols instead of colors might be a better solution, both maintaining the current color-scheme, and adding new information. The coloring, highlighting and naming of inner types should apply to both of the diagrams.



(a) Original



(b) Changed naming of inner types



(c) Listeners connected to the object listened to

Figure 7: Crude comparison of suggested changes to the contour diagram

Exploring changes to the default exclusion-filter, in order to provide more useful information out of the box, is also an option. This will provide a better experience for certain scenarios, but may be useless in other cases by providing too much unnecessary information. A way to easily switch filters by defining presets might be preferable. The filter might also be extended to support both exclusion and inclusion, allowing a more fine-grained selection of interesting classes. As an example, one might be interested in ignoring the entire javax-package, but still allowing javax.swing in order to see more GUI-related objects. It is also important to not allow too many packages through the filter, as the performance can suffer immensely from logging too many events.

Allowing more ways to interact with the diagrams, e.g. hiding elements or compressing sequences, should improve usability for larger programs and longer runs. The way the sequence diagrams currently work, object instances are added as they appear, resulting in later method calls to draw lines crossing large parts of the diagram, as shown in Figure 8.

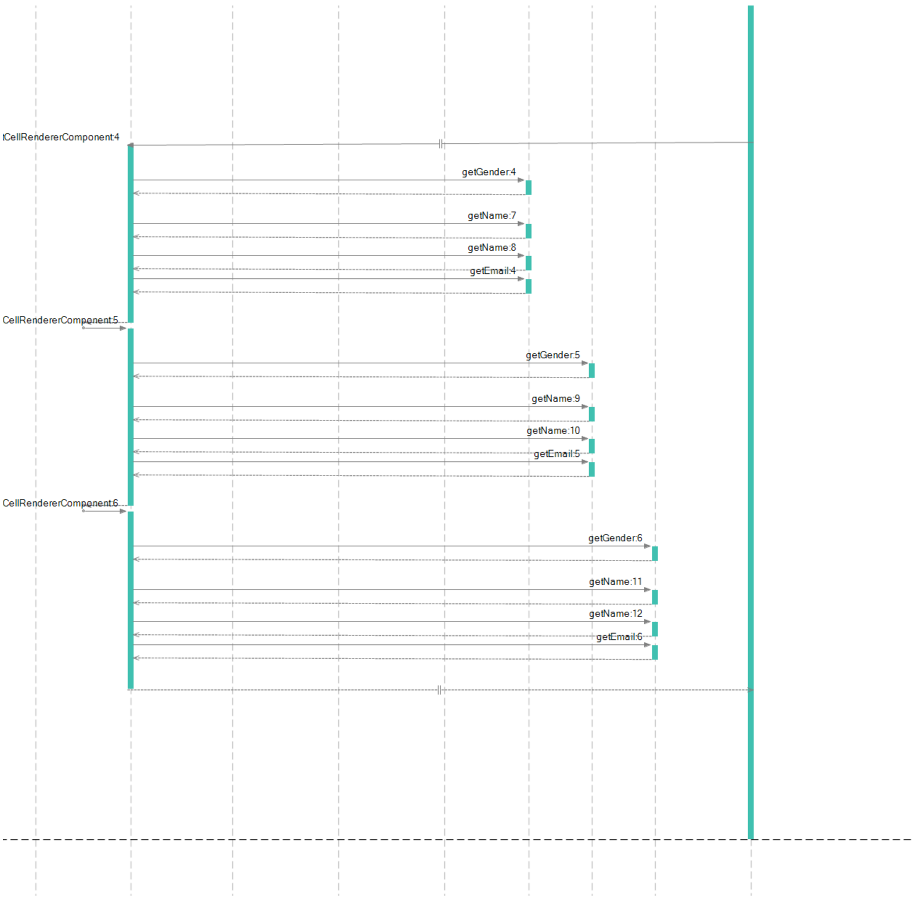


Figure 8: A section of a larger sequence diagram, showing method calls crossing several unrelated lifelines

In the case of exploring events triggered by a listener, the ability to isolate the involved objects, and reorder the diagram could be very useful, providing a clean view of a smaller series of events.Using the same figure as an example, the rightmost, and longest, bar would be moved to the far left, the three second-longest would be in the middle, while the shortest ones would be to the left. In total, only the lifelines of those five objects would be visible, all the ones that are being crossed in Figure 8 are unnecessary, and would be hidden as shown in Figure 9. This function should be accessed by right clicking on the topmost event that is desired, and selecting a "view isolated" option. Whether the isolated view should appear in a new eclipse view, or simply modify the existing view of the sequence diagram, is something that should be determined by user-testing.

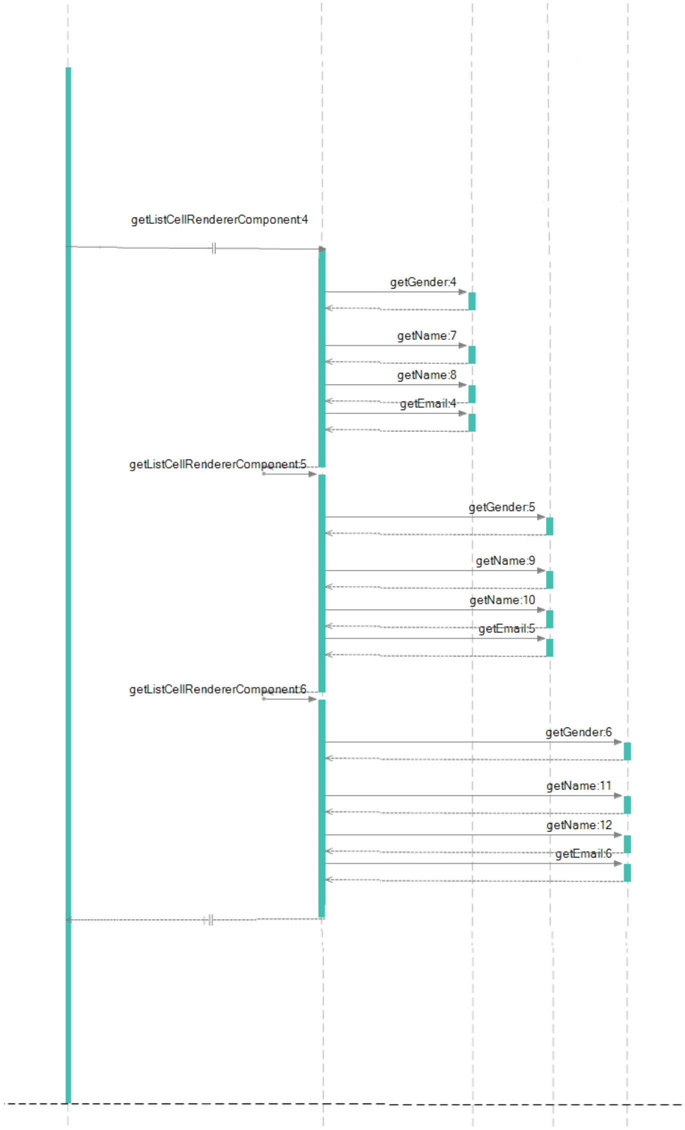


Figure 9: An example of how Figure 8 might look in the isolated view

Stepping through the recorded states is, as mentioned above, cumbersome. While there are quick and easy ways to jump straight to any interesting state, it may also be interesting to view a playback of a part of, or the entire execution. This would allow an easier way of observing changes happening in a program. The playback would automatically step through the recorded states at a pace that the user should be able to adjust, updating the diagrams for each step. Playback should be possible to initialize from any selected event that has been recorded, as starting from the beginning would often be a waste of time.

Searching can be improved by relaxing the requirements for for search-terms. The requirement of full class names in searches caused some confusion, and made me wonder if the feature was working at all. Relaxing this requirement to allowing partial class names, or substrings in general, would be an improvement to the usability of the tool.

4.4 Implemented changes

The first change to be implemented was the identification and presentation of instantiated interfaces, which are now displayed with an appropriate icon, as well as being labeled with the interface it implements, instead of the class name that is assigned by default. This function was also expanded to identify instances of abstract classes, and the lambda-expressions that were introduced with Java 8.

The filtering function was expanded with the ability to specify packages that are not to be excluded from the execution model. Adding a package to be included is as simple as adding a '+' in front of the package-name when adding it to the filter. Unfortunately, a package may hold several classes that the user has no interest in seeing, but that are used by the few classes that the user is interested in. This can result in very poor performance, and cluttered diagrams, but can be handled by adding the unwanted classes to the filter for exclusion.

The isolated view of the diagram does what was proposed in Figure 9,

5 Conclusion

Glossary

Breakpoint A source code marker telling the debugger to halt program execution at a certain point. 4

Code Canvas A visualization-tool for Microsoft visual studio, showing code, diagrams and documents on a large layered ccanvas. 5

Contour diagram An enhanced object diagram, showing objects, their variables and their relations to other objects [Jayaraman and Baltus, 1996, Streib and Soma, 2010]. 5

Execution trace A log of all changes to the state of a program throughout its execution. 4

GDB GNU debugger. A multiplatform, multilanguage CLI-debugger with tracing. http://www.sourceware.org/gdb/. 4

IDE Integrated Development Environment. A software application that provides facilities for software development such as source code editor, compiler etc. 4

JAVAVIS Tool that generates UML diagrams from running java applications [Oechsle and Schmitt, 2002]. 5

Jinsight An advanced debugger made by IBM, supports visualization, and powerful analysis [Pauw and Vlissides, 1998]. 5

JIVE An advanced debugging tool supporting visualisation, backward stepping, and querying. http://www.cse.buffalo.edu/jive/. 5

JQL Jive Query Language, used to formuate queries within the jive debugging environment. 5

Trace Viewer Plugin Tool to visualize and analyze communication of parallel message passing programs [Kranzlmüller and Klausecker, 2009]. 4

Trace-Oriented Debugger Trace-Oriented Debugger. A debugging tool that executes queries on program traces [Pothier et al., 2007]. 4

Whyline A query-based debugger that provides an easy way to find out why things are as they are [Ko and Myers, 2009]. 4

References

[Jayaraman and Baltus, 1996] Jayaraman, B. and Baltus, C. (1996). Visualizing program execution. *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 30–37.

[Ko and Myers, 2006] Ko, A. and Myers, B. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, ...*, 32(12):971–987.

[Ko and Myers, 2009] Ko, A. J. and Myers, B. a. (2009). Finding causes of program output with the Java Whyline. *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, page 1569.

[Kranzlmüller and Klausecker, 2009] Kranzlmüller, D. and Klausecker, C. (2009). Scalable Parallel Debugging with.

[Oechsle and Schmitt, 2002] Oechsle, R. and Schmitt, T. (2002). Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). *Software Visualization*, pages 176–190.

[Pauw and Vlissides, 1998] Pauw, W. D. and Vlissides, J. (1998). Visualizing object-oriented programs with jinsight. *Object-Oriented Technology: ECOOP'98 ...*, pages 541–542.

[Pothier et al., 2007] Pothier, G., Tanter, E., and Piquer, J. (2007). Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10):535.

[Streib and Soma, 2010] Streib, J. and Soma, T. (2010). Using contour diagrams and jive to illustrate object-oriented semantics in the java programming language. *Proceedings of the 41st ACM technical symposium ...*, pages 510–514.