Contents

List of Figures

Chapter 1

Introduction

Upon starting their studies in computer science, new students may find the concepts of programming in general, and object oriented programming in particular, to be difficult to grasp. 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 helpful 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 students 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 Human-machine-interaction (HMI). This course handles topics related to creating programs with a *Graphical User Interface* (GUI), and specifically how to implement a *Model View Controller* (MVC) architecture using Swing, a part of the framework surrounding the Java programming language, providing components that are frequently used in graphical interfaces. The learning goals for the HMI-course are as follows:

> Introduction to important concepts, methods and techniques related to human-computer interaction and design of user interfaces. Knowledge and practical experience with implemetation of user interfaces in object-oriented frameworks.

Through the course, students are introduced both to the theories of good user interface design, and how such interfaces can be implemented in Java. Java, and the *Integrated Development Environment* (IDE) Eclipse, make a common framework

for many of the programming courses at NTNU, making them the main focus when looking for tools to support the students.

1.1 Research Questions

The goal of this report is to look into the available tools that can aid developers, focusing on the potential use in the HMI-course. This implies a focus on how to better understand the interactions within programs implementing the MVC pattern, as well as the behavior of graphical interfaces. Looking into the tools, it will identify the different types, as well as their strengths and weaknesses, using this as a basis for comparison. After comparing, and determining which tool is likely to be of most use, the report will look at both the potential for improvements, and how it may fit into the teaching process, including the effect of any improvements made. To be more specific, the following research questions have been formulated:

Research Question 1 *What is the current state of the various visualization tools that are available?*

Research Question 2 *Could any of these be integrated into the current teaching environment at NTNU, consisting of Java and Eclipse?*

Research Question 3 *Is there room for improvement in how these tools are used, and the ease of using them?*

Research Question 4 *Would the use of such tools and any improvements actually be useful for the students?*

Chapter 2

Prestudy

This section will serve to get an overview of the current situation of debugging methods, visualization techniques, and the various tools that are available.

2.1 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.

Object diagrams are similar to class diagrams, in that they both show the objects of a program, and their connections between each other. Where they differ, is that while the class diagram shows the connection between classes, and what the classes are composed of, the object diagram shows the state of a program at a certain point in time. As a program is executed, its state changes, and the exact combination of objects and the connections between them will change accordingly, and the object diagram reflects this. Some programs will return to a certain state after performing a task, while others do not have this 'stable state'. The transition between stable and working states can be illustrated with a state diagram, which typically abstracts the states to 'idle' and various 'tasks', like 'processing input'. A sequence diagram shows the order in which the program is executed, showing components invoking methods on each other.

Depending on the desired kind of diagram, different techniques are used for the generation. Class diagrams can be made by analyzing the source code of a program, while object diagrams require information that can only be acquired by analyzing a

running program, and logging what happens. Such a log, or *execution trace*, can also be used to create a sequence diagram, as those also need information that can not necessarily be acquired by analyzing code. State diagrams are usually created manually by a system architect, as automated identification of states is not a trivial task.

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. There are different ways to store the information describing each step, providing various trade-offs between access-time and memory usage. The straightforward way would be to store each state independently, sacrificing memory for a near-constant time to load the details of a step. The load time will naturally be affected by the amount of data that must be analyzed, but there is no need to work through any the steps in between. If the data is stored as a differential of the previous state, the opposite situation is created. The amount of memory required is significantly reduced, and stepping to the immediate neighbors is very fast, but jumping between any two steps would require analyzing every step in between the two. A hybrid approach is also possible, relying on a differential model, but also introducing checkpoints every n steps, where all information is stored. This will provide a balance between the two previous methods, and could be adjusted to fit the characteristics of the system the tool is running on.

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.

A major downside of execution traces, is the performance penalty from doing extra work for every step in the execution. The amount of work will vary, depending on implementation details and of course what is done with the log. If all the tracing-process does is write to a log-file, which will be used later, one can achieve significantly smaller performance impact compared to a system that does real-time analysis of the data.

2.2 Existing Tools

There currently exists several tools that provide one or more of the methods mentioned above. The following list is not extensive, but includes some of the most interesting tools, considering the focus of this report. Some of the tools listed below does not support Java or eclipse, but provide interesting features that are worth mentioning, despite their incompatibilities with the desired teaching environment.

GNU debugger (GDB) offers a tracing environment, and support for many languages, but not Java. Due to its *Command Line Interface* (CLI), it is not necessarily easy to use on its own, and so, there are several front-end platforms that provides a graphical environment around GDB, including Eclipse.

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, and the languages it supports.

The *Trace Viewer Plugin* [Kranzlmüller and Klausecker, 2009] for g-eclipse – a now discontinued version of eclipse, geared for development of grid-computing software – uses a trace to generate visualizations of the program execution, and thus makes it easier to understand, but is designed for massive parallelism, requires a special version of eclipse, 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.

The *Debug Visualization Plugin* for eclipse provides an alternative to the variable view provided in eclipses debug-perspective, providing a graph that represents the variables of a program. The user still needs to use regular techniques they would use normally in order to pause the program-execution, and be able to actually view the state of the variables. This was previously tested in the course TDT4100 - Object oriented programming, with the conclusion that the diagram provided quickly became to large, including the contents of objects that were not important.

JAVAVIS [Oechsle and Schmitt, 2002] provides visualizations in the form of UML-diagrams, but does not provide any debugging features.

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 is a tool that utilizes execution traces to provide diagrams while running a program. Developed at the university of Buffalo, it is installed a an Eclipse plugin, and provides several new views to display the information it provides. In addition to providing diagrams, the trace log is also used to enable backstepping, which is coupled with the diagrams to always show the selected execution state.

Of the mentioned tools, the Trace Viewer Plugin, Trace-Oriented Debugger, Debug Visualization Plugin, and JIVE are all available as eclipse-plugins, and are

thus fairly simple to integrate into the existing teaching process. The features they provide, on the other hand, vary. The Trace Viewer Plugin is as mentioned, designed for massive parallelism, and does require a specific version of eclipse, and because of this, it is not really suited for further study.The Trace-Oriented Debugger provides a debugging environment supported by trace logs, as implied by the name. It presents its information mostly in a textual way, and does not provide any visual diagrams of the program structure, or execution order. The Debug Visualization Plugin expands on the debugging functionality of Eclipse by providing a visual view of variables, and is designed to be used alongside the rest of the debugging environment provided by eclipse.

JIVE seems to be the only tool that utilizes all three methods mentioned in section 2.1, as well as being freely available as a plugin for eclipse, making it easy to install and use. During program execution, 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 with pre-defined search-templates added to the built-in search window in Eclipse.

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 are likely to 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.

Chapter 3

Methodology

While exploring the field in the previous chapter, the first research question was answered in the process. The next chapters will be used to answer the remaining questions using the following techniques.

Having looked into the various tools available, and compared their feature sets to determine which one is most likely to fit with the current teaching environment, question two is close to an answer as well. By performing a closer study of the features of JIVE, its compatibility with the teaching environment can be answered to a better degree. Such a study is also an essential part in finding whether there is any room for improvement, and answering the third question. In addition to discovering potential improvements through the use and exploration of JIVE, determining whether the changes necessary to implement them are feasible to accomplish within the given time-frame is important to avoid half-implemented changes.

After thoroughly exploring JIVE, and identifying potential improvements, it is time to select what to improve, and implement the necessary changes. The available time-frame limits how many improvements it is possible to implement, as some time will most certainly be lost in understanding the source code of JIVE. Some time is also required at the end to answer the final research question. This is best answered by performing a user evaluation with students in the target group. This group consists of those described in the introduction, students in their second year of the computer science and informatics study-programs.

The types of evaluations available are limited to av few. Reaching a wide audience, the use of questionnaires can provide a large amount of data to support the research, but the nature what is being evaluated makes this an infeasible method. In order to give proper answers, the participants require a hands-on experience with the tool, and that would require them to acquire and install both the original version, and the modified version. This is a rather large obstacle that would deter most potential candidates from participating, and defeats the point of a questionnaire.

The alternative is an in-person evaluation with a smaller group of students, where the participants get access to a pre-configured system and enough time to make an informed opinion of what they are presented with. The availability of volunteers is not likely to be great in either case, but a properly executed in-person evaluation is likely to get more information out of the participants, by asking questions and discussing potential issues with them.

Chapter 4

An exploration of JIVE

This Chapter will provide details on the various features of JIVE, and how they are used. This will be followed by an identification of various weaknesses, if any, as well as suggestions on how those areas may be improved.

4.1 Features

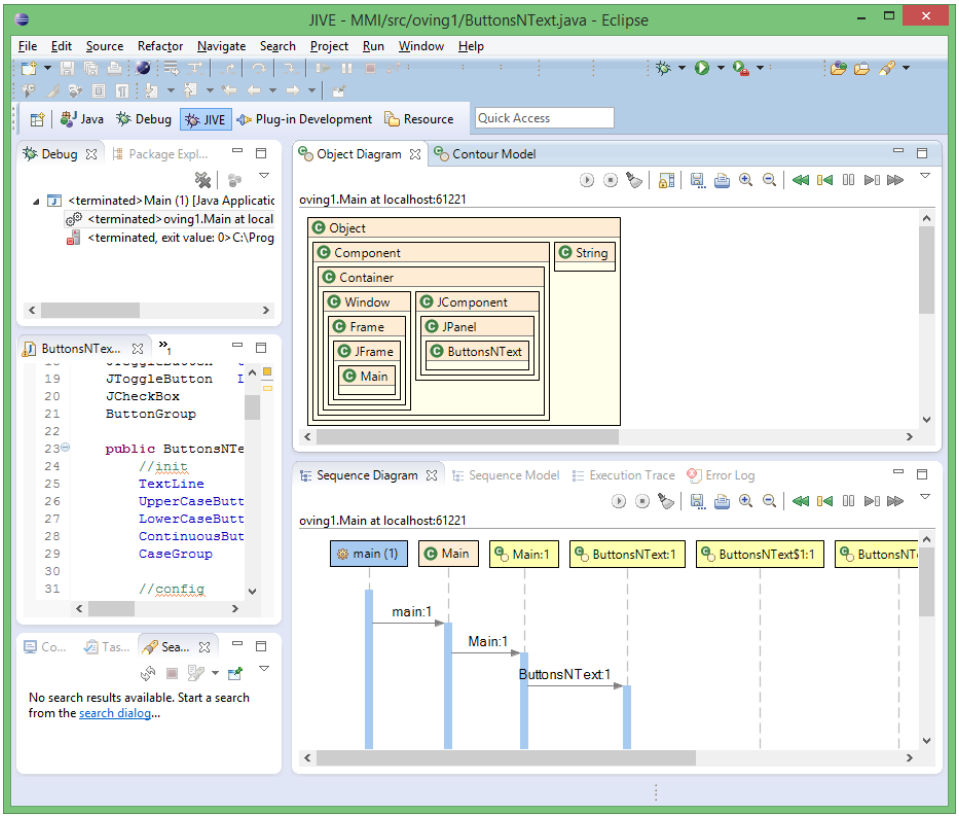


Figure 4.1: The JIVE perspective in Eclipse

As mentioned in the prestudy, JIVE installs as a plugin in the eclipse IDE, adding another perspective in the environment shown in Figure 4.1. 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".

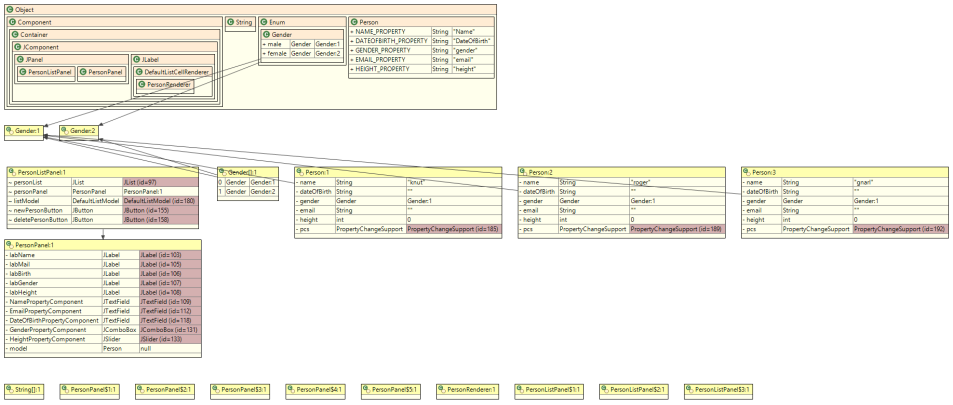


Figure 4.2: A contour diagram generated by JIVE

The object diagram-view shows the current state of the program by using a contour-diagram. Contour-diagrams, as shown in Figure 4.2, 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, and can be compared to object diagrams, in terms of the information that is shown. 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.

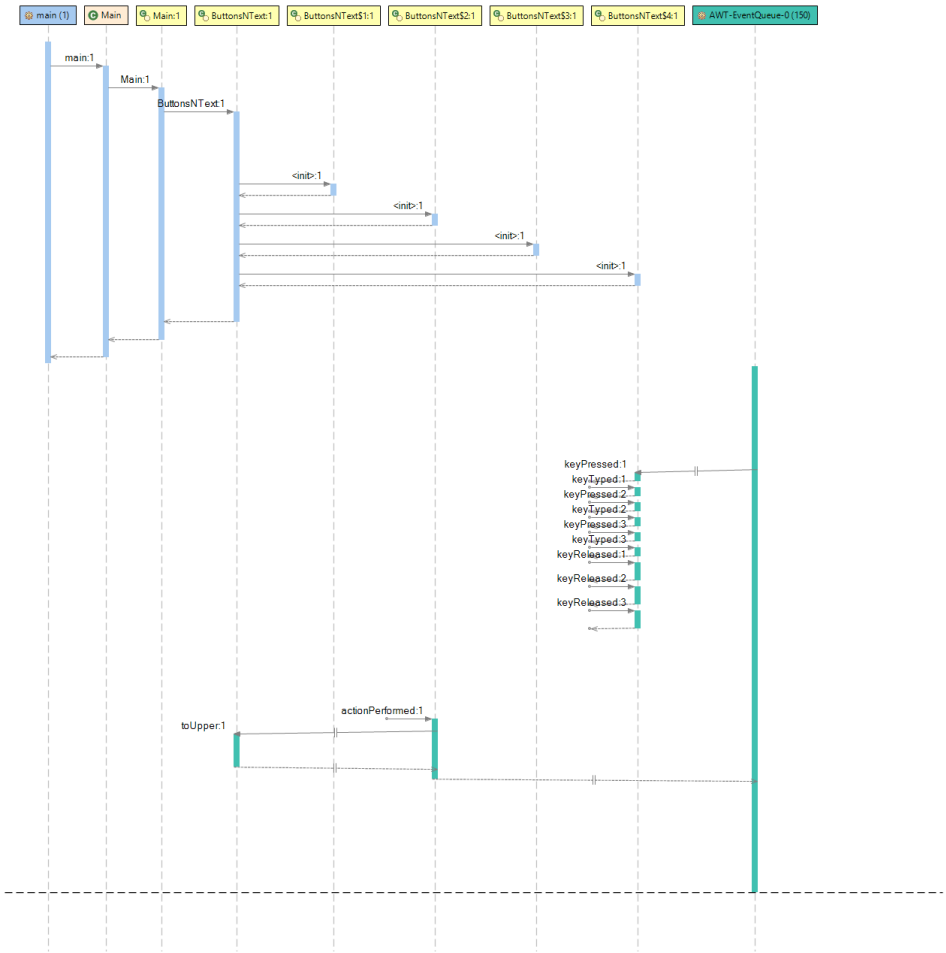


Figure 4.3: A sequence diagram generated by JIVE, while running an instance of HMI-Exercise 1

The sequence diagrams, shown in Figure 4.3, are fairly standard, with threads and object instances represented by boxes in a row at the top, each with a vertical line coming down. The actual process is shown with a thicker lifeline overlaying the vertical line of the object that is currently active, with arrows between lifelines 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 which objects and methods are involved. In Figure 4.3, 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.

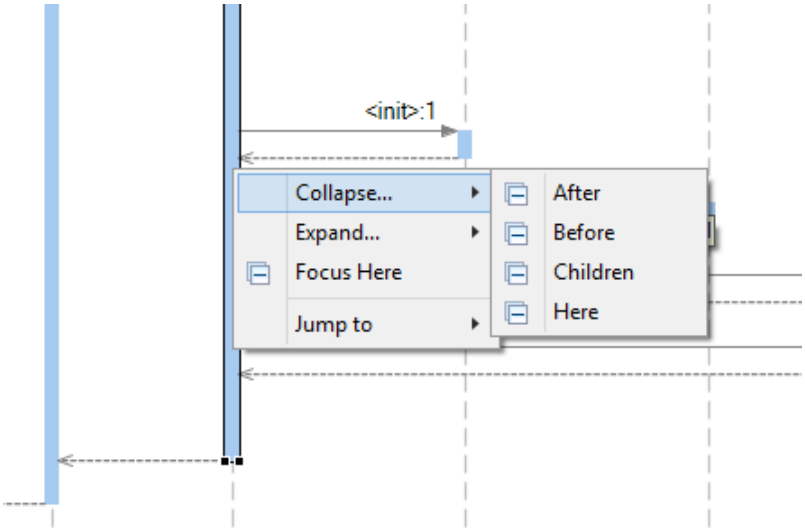
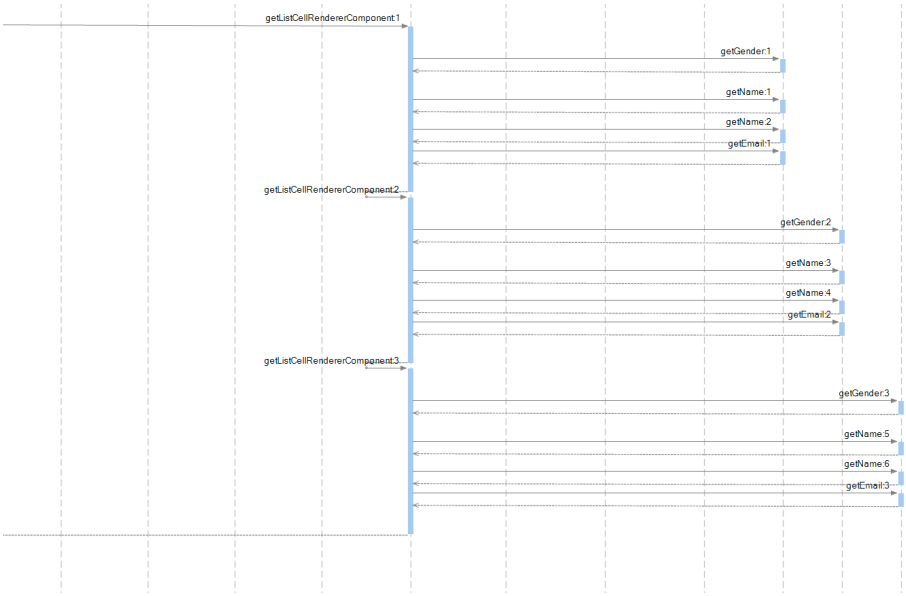


Figure 4.4: The sequence diagram right click menu

Right-clicking on a lifeline provides the ability to collapse method-calls originating from that lifeline in order to hide unnecessary information, shown in Figure 4.4. The collapse-menu provides 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 provides to collapse that objects entire lifeline, the result of which can be seen in Figure 4.5. While Figure 4.5 shows the result of collapsing the lifeline of an object, a similar result can be achieved by selecting 'collapse children' at the parent lifeline, or by selecting 'collapse here' on each of the three method-calls shown. The same options are available for expanding collapsed elements. Right-clicking also provides to set the execution-state via the jump-option, updating the contour-diagram and -model to show that state.



(a) Normal

(b) Collapsed

Figure 4.5: 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. The diagrams 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, which contains an entry for every single event that occurs during the execution of a program, excluding those that are removed by the exclusion filter. 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 and complexity of the programs that can be used with JIVE in a meaningful way. On the other hand, it allows quick 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.

In order to improve both performance and readability, the mentioned exclusion filter checks the origin of each event, and determines whether or not the event is to be logged. By default, the filter excludes the entire Java API, and by doing so, it focuses on the classes that are made by the user.This filter can be adjusted to better suit the program being analyzed, by adding or removing entries in order to see more or less details of the execution.

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 4.6, easily accessible through the search-view, and comes with several pre-defined templates to simplify searching. For example, searching for when a variable gets a certain value, only requires the user to specify the variable-name, the object it is contained withing, and the value.

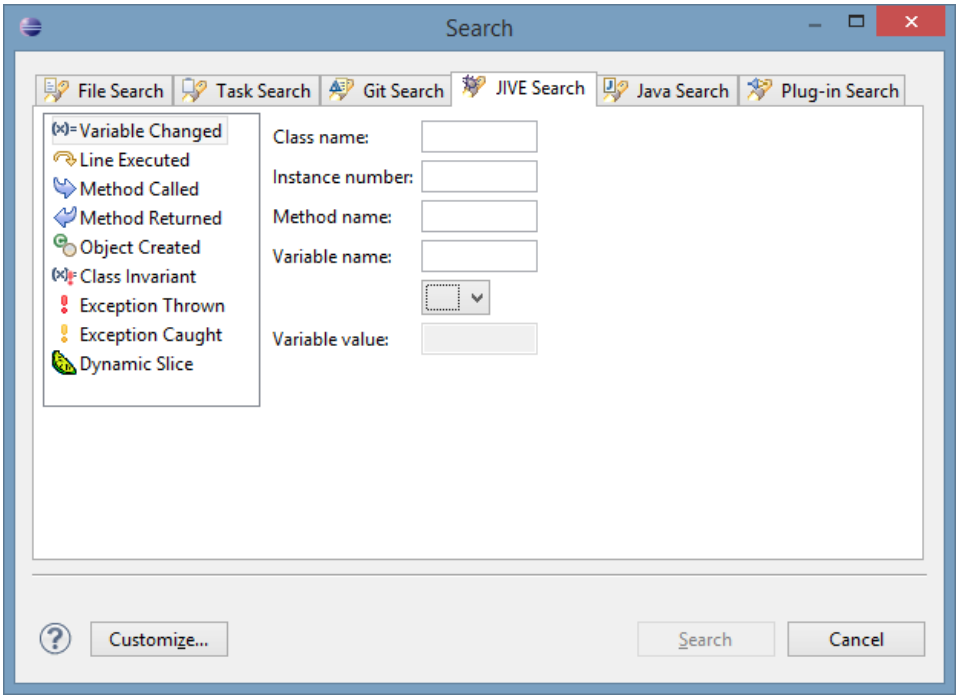


Figure 4.6: The JIVE search panel

4.2 Suggestions for improvement

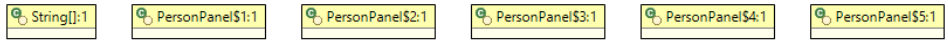
While exploring the features of JIVE, a few issues regarding use and behavior were encountered. None of these issues can be considered to be major, but they still expose a potential for improvement. This section will point out some of the issues, and provide suggestions on what can be done to improve them.

4.2.1 Visual changes to the diagrams

In both diagrams, object instances are labeled with the name of the object type, and an instance-number, separated with a colon, e.g. 'PersonPanel:1'. While this provides the expected information, it is affected by the way Java handles the naming of anonymous inner types – classes that are defined within another class, and not given a name by the developer. Java automatically names these classes by appending a dollar sign and a identifying number to the name of the class they are defined within, e.g. 'PersonPanel$1'. GUI-programs are written with several such classes to handle user input, each implementing a listener-interface of some

kind, and they show up in the diagrams with their generated names, that do not reflect the purpose of the class. The ability to detect an inner type with a generic name, and display the interface it implements, instead of is actual name should help with this issue. Among others, listener-heavy programs should become more understandable, as the kind of listeners in use will be clearly visible, removing the need to guess based on when they are invoked in the sequence-diagram. Figure 4.7 illustrates this as the change from part a to part b.

To further help users identify listeners, they could be visually linked to the objects they are listening to in the contour-diagram, making it clearly visible which object is being listened to by the different listeners, as illustrated in Figure 4.8 illustrates this change, along with the relabeled objects. While the effect is possible to achieve by applying an appropriate filter, this filter must let a large amount of objects through, causing the diagrams to become cluttered and challenging to navigate. On the other hand, the effect of the listener being triggered is clearly visible in the sequence diagram, and it is possible that this is easy enough to connect with a users interactions with a program, so that such a link will not be necessary.



(a) Original, cropped view of Figure 4.2

(b) Changed naming of anonymous inner types

Figure 4.7: comparison of the suggested label-changes to the diagrams

A last visual change would be to attempt further identification of the components that make up an MVC-architecture, and highlight them. Being a change focusing solely on a specific architecture, it has to be possible to deactivate when not needed. The major challenge with this feature is to find a way to identify the different components. While listeners can usually be identified by their name, of by their implementation of certain interfaces, this is not necessarily the case for models and controllers. A possible solution would be to require developers to follow a certain naming scheme, but this would still have a risk of false positives – leading to incorrect diagrams, and adoption of such a scheme with the sole purpose of using a special tool, is not something to expect. The highlighting could be done by adding colored frames to the identified objects, or by adding a icon indicating the role of a given object. Not as big of a challenge, this does pose the problem of not making the diagrams too distracting and hard to read due to the added elements.

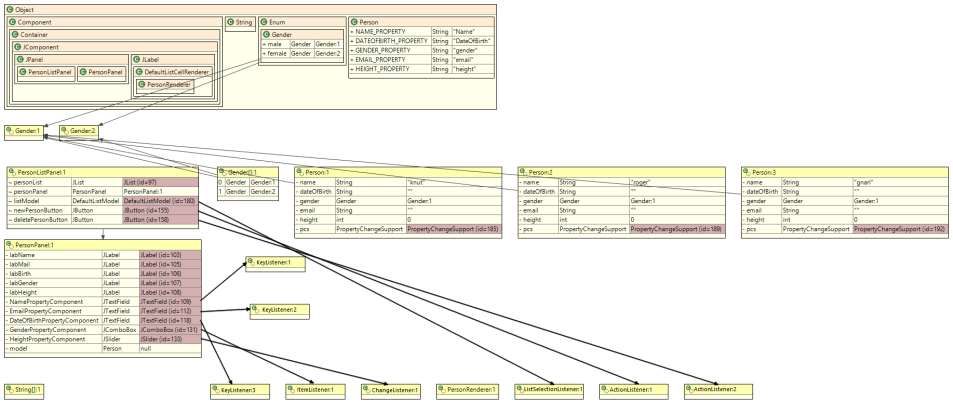


Figure 4.8: Listeners are relabeled, and connected to the objects they listen to. The illustration is based on Figure 4.2

4.2.2 Functional changes

Both diagrams can easily grow to sizes that make them hard to navigate and to get an overview of. While the contour diagram offers different presentation modes, by for instance only showing objects, and not their data-fields, the sequence diagram offers fewer options. Collapsing parts of the diagrams do help in reducing the amount of information shown, and thus ease the interpretation of the diagram, but are still some issues that are a result of how the diagram is drawn. Firstly, there is currently no method of vertically compressing the sequence diagram, resulting in long sequences with no visible interaction if that sequence has already been collapsed, or if there are a lot of filtered events occurring within the lifeline of an object. There is an ongoing development to solve this, by substituting an event-pattern with a single new event, and thus reducing the vertical size of the diagram. If this technique is applied when using the existing horizontal collapse, the diagram should shrink in both directions.

The other issue is a consequence of having the objects at the top of the sequence diagram added in the order in which they are used. This is generally a good way to order the objects, especially when considering the start-up sequence of a program, which is then displayed in a well organized way. Where this approach becomes problematic, is when the newer objects start interacting with the older ones, causing the lines that indicate method calls to cross over parts of the diagram, and often going back and forth. This problem is illustrated in Figure 4.9, where a listener triggers activity in an older part of the program.

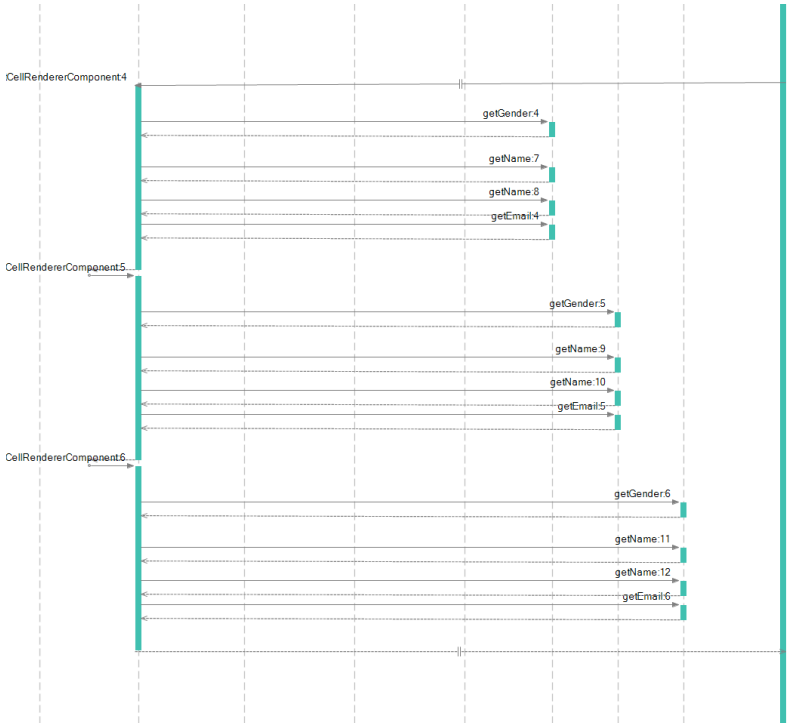


Figure 4.9: A section of a larger sequence diagram, showing method calls crossing several unrelated lifelines

In some cases, the amount of unrelated lifelines that are crossed becomes so large, that fitting the entire sequence within the view of a single screen becomes impossible without zooming out, and thus rendering the text unreadable. Scrolling sideways to follow consecutive events is not ideal, and can potentially be disorienting. This could be solved by allowing users to view such a sub-sequence in a separate, isolated environment, with the objects lined up in the order in which they are used, as illustrated in Figure 4.10.

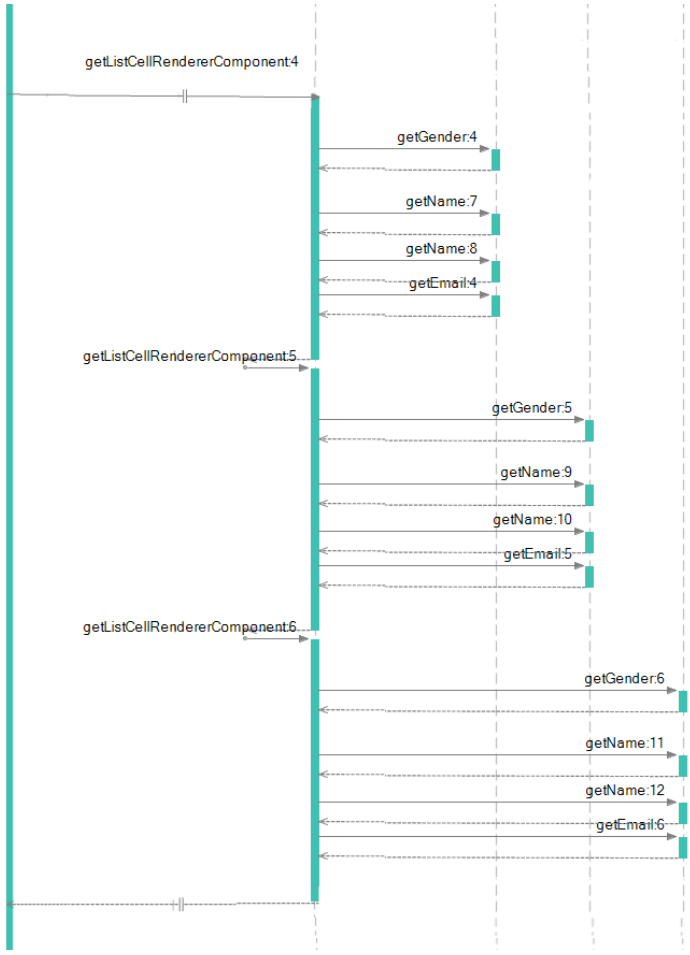


Figure 4.10: An example of how Figure 4.9 might look in the isolated view

The filter operates by excluding unwanted elements from the trace-log, and only takes a list of elements to exclude as input. While this simplifies the internal operation, and makes it easy for users to understand, it also makes it hard to build a more fine grained filter, where , for instance, a package containing several sub-packages is excluded, but one of those sub-packages is not. The current way to achieve this is to individually list each of the sub-packages that are excluded, while omitting the one that should be included. This results in a potentially large list of packages, and will take an unnecessary amount of work to add to the filter. By instead allowing the users to add the one package to be included, and using both a list of inclusions and a list of exclusions to build the filter, the amount of work for the user is significantly reduced, while maintaining readability.

The ability to search through all the events is a potentially useful feature, depending on the scenario in which JIVE is used. When merely getting an overview of a program, it is likely not needed at all, but for debugging purposes, it has a clear use as a way of finding specific events. It is unfortunately very strict in its interpretation of the search terms, being both case-sensitive, and requiring full class names, including the package the class is contained within. It is assumed that this behavior was chosen in order to give an accurate, and limited set of results, instead of risking a large amount of false positives. On the other hand, this behavior goes against the expectations set by most search providers in other areas, and can lead to confused users, that may, at worst, consider the search to be non-functional. By relaxing the search criteria to not be case sensitive, and to allow partial matches, the usage of this feature would be less confusing, and still allow the users to narrow their results by being more specific with their search-terms.

Chapter 5

Implemented changes

The first change to be implemented was the identification and presentation of instantiated interfaces, which are now displayed in the diagrams with an appropriate icon, as well as being labeled with the interface it implements, instead of the generic class name that it is assigned by default. An example of this is shown in Figure 5.1 This function was also expanded to identify and label instances of abstract classes, and the lambda-expressions that were introduced with Java 8. In order to still allow the actual class name to be visible, the tool-tip shown when hovering the mouse pointer over an object remains unmodified, showing the actual class name and icon.

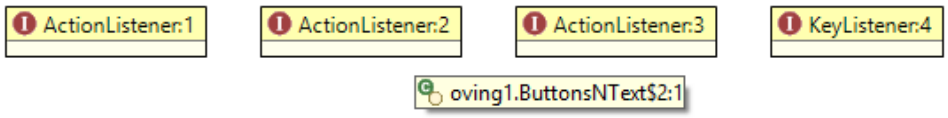


Figure 5.1: A view of how instantiated interfaces are shown after the modification. Also shown, is the tool-tip text for the second instance from the left. Originally, they would be labeled as 'ButtonsNText$1:1', 'ButtonsNText$2:1' and so on, with the green class-icon.

The filtering function was expanded with the ability to specify packages that are not to be excluded from the execution model, as suggested in section 4.2. Adding a package to be included is as simple as adding a '+' in front of the package-name when adding it to the filter. As an example, the default filter excludes the javax package, and any subpackages that are not specifically listed with a '+' in front. When adding the line '+javax.swing.*' in order to let swing components through the filter, one will let every subpackage of 'javax.swing' though the filter as well, which

is the intended design. Unfortunately, there are a lot of classes in both swing, and its subpackages that are not used directly when writing swing-programs, and a user will have no interest in seeing these in the diagrams. This can result in very poor performance, as unnecessary events are logged, and cluttered diagrams, but can be handled by adding the unwanted classes to the filter for exclusion. Depending on the package in question, the amount of extra items added to the filter can become quite large, and identifying all unwanted classes and subpackages may take hours at worst. One weakness in the filter, related to the problem of unwanted packages, is that due to its design, allowing the contents of a package, will also allow any classes that are contained directly within the parent package, as this also has to be removed from the internal list of exclusions that make up the filter. This is not a big problem in the case of 'javax.swing', as the 'javax' package does not contain any classes, but there are definitely other packages that will show this behavior.

With an appropriate filter, the connection between listeners and the objects they listen to should become visible, via the existing object-containment-links, given that the listener-field is actually visible. An unfortunate behavior of JIVE is that only the fields defined within a class are visible in the figures representing instances of that class. Any inherited fields are not shown, and this is unfortunately the case for many UI-components in the swing-framework. The quick solution to this problem, simply showing all inherited fields, would result in a large amount of useless information, cluttering the diagram, and hiding the useful information. In lack of a good solution, and with a desire to not make large fundamental changes, preferring minor adjustments that use the existing framework in better ways. Assuming that the listener-field is visible, this would in most cases be a list of listeners, and so the filter would also need to allow that type of list through. As there are a large amount of lists in use behind the scenes in a graphical program, this would be another source of unnecessary clutter. The filter itself is likely to become very large due to the extra exclusions needed to hide unwanted classes, as a consequence of letting something through.

There is currently no easy way to quickly swap the entire filter for another one, but the filters are stored in eclipses launch configuration files. These configurations are stored as XML-files, and can be easily modified in any text editor. By managing these files directly, a user can quickly swap the entire filter, copy it to another launch-configuration file, or share it with others. After modifying these files, Eclipse requires a restart to detect the changes, adding to the time and effort required to make any changes this way. This is of course not a good way of handling the issue, and ways of improving this should be explored in the future.

The isolated view was implemented as a separate view-tab, and does what was proposed in Figure 4.10: it displays the events caused by the selected event, and hides everything else. Figure 5.3 shows what Figure 4.10 looked like after the modification. Figure 5.4 shows a similar, but slightly more complex situation in the isolated view. By right clicking on an event in the regular sequence diagram,

and selecting the ‘Isolated view’ option, shown in Figure 5.2, the isolated view is triggered. It is also possible to further focus on a part of the diagram from within the isolated view, by right clicking on the desired event, and selecting the ‘Isolated view’ option again. All of the functionality from the regular sequence diagram has been retained in the isolated view, so that the only difference is what parts of the execution are visible.

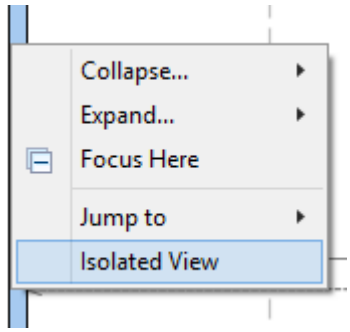


Figure 5.2: The new ’Isolated view’ option.

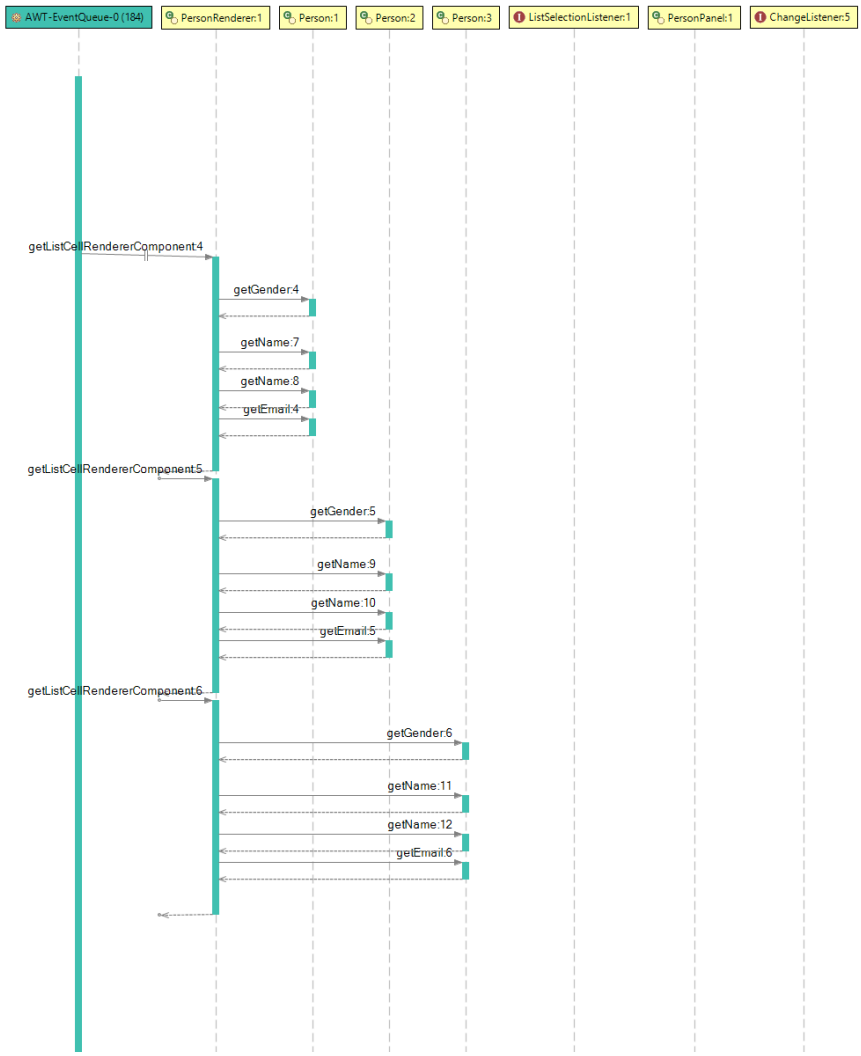


Figure 5.3: How Figure 4.9 looks in the isolated view. The three last elements shown on the top are referred to in parts of the diagram not shown. This view was achieved by selecting the 'AWT-EventQueue' lifeline for the isolated view.

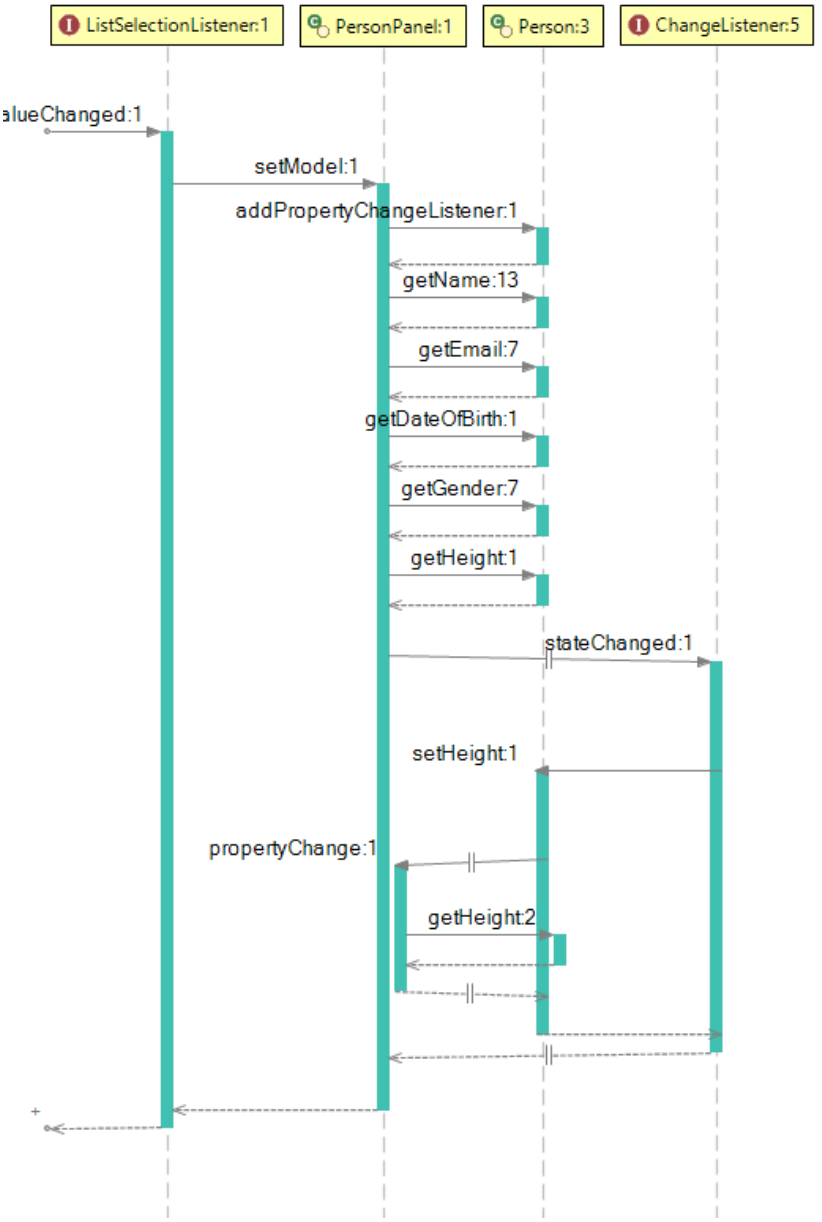


Figure 5.4: Another example of the isolated view, showing a significantly reorganized diagram.

The searching functionality was relaxed by allowing partial matches, and dis-

abling case sensitivity. While relaxing the search may cause false positives, a user should be able to identify such cases quickly, or at least identify the results they were looking for. The fact that partial matches and case insensitivity is the standard behavior in most search-engines, sets an expectation for the behavior of the search within JIVE. Unfortunately, due to the existing implementation consisting of separate searching- and matching-methods for each search type, not all searches were updated to this relaxed state.

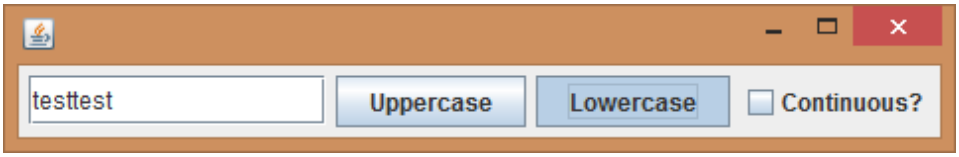
Chapter 6

Evaluating usefulness

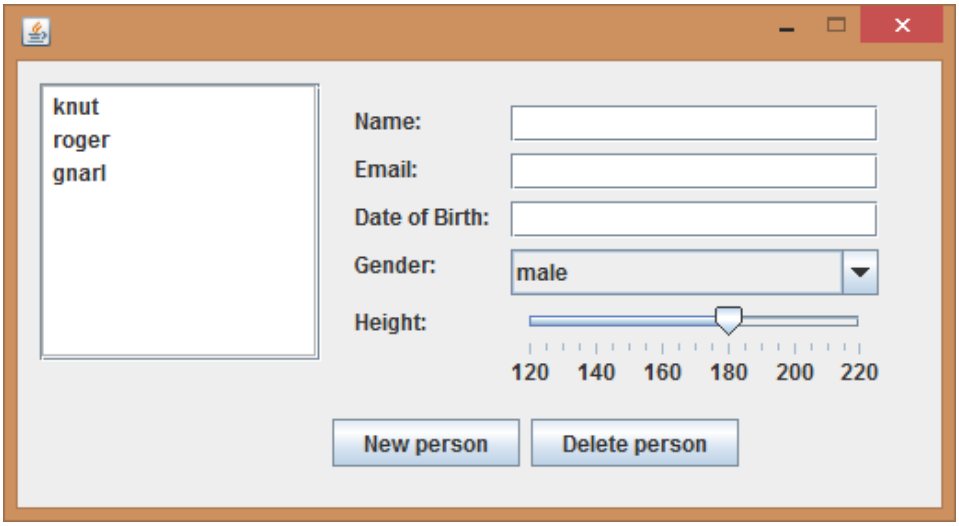
6.1 The test

In order to evaluate both the usefulness of JIVE, and the changes that were implemented, an evaluation was performed with a small group of students, one by one. During this test, the students were given a demonstration of the main features of JIVE, using implementations of the first, and the last exercise of the HMI-course as examples. After getting familiar with JIVE, they were shown the modified version, focusing on the new and modified features that were implemented. They were then asked questions about how easily they understood the diagrams, and how useful they found the changes to be. As well as whether the performance trade-off was acceptable for occasional use, if they could see themselves using JIVE, and if so, in what situations.

The Example programs both consisted of a simple user interface in swing, implementing the MVC pattern. Exercise 1 consists of a text field and two buttons to transform the text entered in that field to either uppercase or lowercase, and with an option to continuallytransform as new text is written. Exercise 4 is an application that manages a simple list of persons. The interface it presents is split between a list of persons, and a set of text fields to enter information about the selected person, as well as buttons to create new, and delete existing persons. Both of these programs, shown in Figure 6.1, were relatively simple, with few events following directly from a user interaction, and as such, should have a limited performance reduction when used with JIVE. More complex programs are likely to suffer a greater performance impact from JIVE, but are also out of the scope for the HMI-course.



(a) HMI-exercise 1



(b) HMI-exercise 4

Figure 6.1: The two example programs used in the evaluation

6.2 Results

The students that participated in the evaluation were mostly in their third year, and had already finished the HMI-course the previous semester. This meant that they were familiar with the example programs, and had some knowledge of the MVC-pattern, and how it behaves in Java-swing. While this excluded the experiences of someone completely unfamiliar to these concepts, it did give the insight of relatively fresh experiences, as the participants remembered what they were finding hard to understand back then. There was also one participant who was in his second year, and as such, would take the HMI-course during the next semester.

The evaluation resulted in mostly positive feedback, with every participant seeing some kind of use for a tool like JIVE. Of those in their third year, only one participant did not consider the use of diagrams to understand code to be useful. In his experience, he learned just as well from reading the code directly, and building his own mental model of the program structure, but as he preferred simple text editors to IDEs, he could be considered to be outside of the target demographic. On the other hand, he did recognize the usefulness of JIVE as a tool to create

documentation, and for presenting his own work to other participants in a project. The rest of the participants shared the view that JIVE, or a similar tool, could have been very useful when trying to understand the connection between GUI elements, listeners, and the actions that are triggered by them. A few specifically mentioned that they found that connection to be difficult to understand. One of them also had experiences as a student assistant, and recognized JIVE as a tool that could give students a needed insight into what is going on in their programs.

The diagrams themselves were found to be both sufficiently detailed, and easy to understand, showing the available information in a simple and structured manner. The tendency for the diagrams to grow large and unwieldy did become a problem, especially when running with the less restrictive filter, showing many of the swing-internals. When hiding everything but user-created classes, the diagrams remained at an acceptable size, although the sequence diagram got quite long after a while.

The implemented changes were generally well-received, with the ability to view a subsection of the sequence diagram in the isolated view being the most preferred change. The changes to the filtering mechanism, while useful, were found to still be somewhat hard to use, due to the lack of an easy way of importing an existing filter, and the amount of work necessary to adapt a filter to the program being examined. The modified labeling of object instances provided a small, but useful addition to the visible information.

Regarding performance, it has already been shown in section 2.1 that the tracing behavior of JIVE must affect the analyzed program negatively. The specific penalty varies with the complexity of the program, and the level of detail that is being logged. The performance of each example program, with both versions of JIVE, and with two different levels of detail in the filter for the modified version, for a total of six measurements, were tested separately from the user evaluation, but on the same computer. Each result was measured five times, with the averages reported in the table below:

Version of JIVE	Program	5-run average
Original JIVE	Exercise 1	5,0s
Modified JIVE, standard filter	Exercise 1	16,5s
Modified JIVE, swing-focused filter	Exercise 1	18,3s
Original JIVE	Exercise 4	8,2s
Modified JIVE, standard filter	Exercise 4	25,0s
Modified JIVE, swing-focused filter	Exercise 4	54,4s

As these results show, there is a significant degradation in performance, going from the original version if JIVE to the modified one. The impact of letting a larger amount of events through the filter adds another solid penalty. The reason for the latter has been explained in section 4.1, and is simply a reflection of the added work

of processing more events as they are let through the filter. The impact observed from simply switching to the modified version, can be explained by the way the filter mechanism was altered. By first constructing a list of every package found in the workspace of the chosen program, and then use this list to add and remove excluded packages from the filter, the filter remains an exclusion filter, and its size reflects the size of the package list. When JIVE then checks to see if an event matches an entry in the filter, it has to look through a large amount of entries, and that clearly affects its performance.

As expected, the participants all agreed that the performance of JIVE was too poor for continuous use, but they still found it acceptable for the occasional use that getting familiar with an example, or documenting a project would require. While a larger project could take several minutes to reach the initial stable state for a user to interact, one would only be interested in including the entire project when generating documentation for delivery. By using the filter to ignore well known parts of the project, one could more easily show the interesting or complex parts, while saving time. In either case, the performance was not found to be a reason to never use JIVE, but it would clearly depend on the complexity of the program, as well as what the filter would be letting through.

While the feedback from the participants was generally positive, they still found some areas that they thought could be further improved with new or modified functionality. The mentioned challenge with importing an existing filter, was among the suggested changes, as well as adding a suggestion-functionality, similar to the autocomplete feature found in many search providers, including parts of eclipse, when manually entering new filter entries. Another issue was the risk of diagrams becoming so large that they would be nearly impossible to navigate. Especially when running larger programs, or when letting more objects through the filter, the size would become a problem. To help with this, there was a desire to have the line of objects at the top of the sequence diagram visible at all times, regardless of how far down in the diagram one is currently looking, which would make the larger diagrams easier to navigate. The ability to compress the height of the sequence diagram, was also suggested. Another suggestion was to make it possible to trigger the isolated view from the mentioned line of objects at the top of the sequence diagram, instead of having to find both the desired objects, and then scroll down to find the first event it is involved in. Finally, it was suggested that the placement of objects in the contour diagram could be reworked to give a more compact diagram. For instance, preventing a single row of objects without connections to others from growing wider than a certain threshold, making a new line instead.

Chapter 7

Conclusion

This section will summarize the previous sections in light of the research questions proposed in the introduction.

7.1 Research Questions

Research Question 1 *What is the current state of the various visualization tools that are available?*

There are currently several different tools that can aid a developer in his understanding of programming, ranging from the regular debuggers found in most IDEs, to code analysis and execution tracers. Most of these focus on one or two features, instead of attempting to combine them all. Regarding NTNUs chosen teaching environment, the amount of available tools is naturally reduced, but there is still a large portion of tools available, thanks to the plug-in support of Eclipse. Among these, JIVE was found to be of most interest, due to its combination of features, including tracing, diagram generation, and state-jumping.

Research Question 2 *Could any of these be integrated into the current teaching environment at NTNU, consisting of Java and Eclipse?*

Both JIVE and other tools are available as Eclipse-plugins, and as such, they can easily be integrated into the courses that use Eclipse. Unless a very platform-specific mandatory framework is used, students are of course free to choose a different IDE while following a course, and by doing so, they will also opt out of the use of JIVE or any other tools. But these students are not likely to be the ones that would benefit the most from such tools in the first place, as a decision to not use recommended tools usually indicates that they know what they are doing. As mentioned in [Gestwicki and Jayaraman, 2005, p. 99], JIVE has already been successfully used to explain design patterns to students in a graduate-level seminar, indicating that it is suitable as an educational aid.

Research Question 3 *Is there room for improvement in how these tools are used, and the ease of using them?*

As detailed in chapter 4, there are definitely a potential for improvements in how JIVE works, and is used. While the existing features are useful as they are, they were found to be lacking in usability. The diagrams were found to lacking in their description of certain classes, making identification difficult. Larger programs were also found to quickly grow the sequence diagram to large sizes, making it harder to get an overview.

Some of these points were the focus of the improvements that were implemented in chapter 5, and their results are summarized when answering the last research question.

Research Question 4 *Would the use of such tools and any improvements actually be useful for the students?*

As the results from the user evaluation show, the use of JIVE, or a similar tool, can certainly be of use for students that are having trouble understanding the interactions in the programs they are creating.

7.2 Future work

STIKKORD:
Test in actual teaching, lectures student assistants, etc. If gui is a desired use case: Focus on performance, minimize unnecessary work/avoid work as soon as possible. JFX is essentially not usable with JIVE in current state: too many things going on behind the scene

in general:
further improvements on usability.
compression of diagram height
looser search terms
improving filter, tree-view of classes/packages found in workspace with checkboxes for inclusion/exclusion?

Glossary

Breakpoint A source code marker telling the debugger to halt program execution at a certain point. 5

Code Canvas A visualization-tool for Microsoft visual studio, showing code, diagrams and documents on a large layered canvas.
http://research.microsoft.com/en-us/projects/codecanvas/ 6

Command Line Interface A text-based interface for interacting with programs via a terminal. 6, 34

Contour diagram An enhanced object diagram, showing objects, their variables and their relations to other objects.
[Jayaraman and Baltus, 1996, Streib and Soma, 2010] 7

Debug Visualization Plugin An Eclipse plugin that provides a graphical view of variables during debugging.
https://code.google.com/a/eclipselabs.org/p/debugvisualisation/ 6, 7

Execution trace A log of all changes to the state of a program throughout its execution. 5

GNU debugger A multiplatform, multilanguage CLI-debugger with tracing.
http://www.sourceware.org/gdb/ 6, 34

Graphical User Interface A user interface composed of graphical elements, as opposed to a CLI. 2, 34

Integrated Development Environment A software application that provides facilities for software development such as source code editor, compiler etc. 2, 34

JAVAVIS Tool that generates UML diagrams from running java applications.
[Oechsle and Schmitt, 2002] 7

Jinsight An advanced debugger made by IBM, supports visualization, and powerful analysis.
[Pauw and Vlissides, 1998] 7

JIVE An advanced debugging tool supporting visualisation, backward stepping, and querying.
http://www.cse.buffalo.edu/jive/ 7

Model View Controller A design pattern defining an architecture where data is kept separate from the component that displays it to a user. 2, 34

Trace Viewer Plugin Tool to visualize and analyze communication of parallel message passing programs.
[Kranzlmüller and Klausecker, 2009] 6, 7

Trace-Oriented Debugger Trace-Oriented Debugger. A debugging tool that executes queries on program traces.
[Pothier et al., 2007] 6, 7

Whyline A query-based debugger that provides an easy way to find out why things are as they are.
[Ko and Myers, 2009] 6

Acronyms

CLI Command Line Interface 6, 32

GDB GNU debugger 6

GUI Graphical User Interface 2, 16, 17, 20, 29

IDE Integrated Development Environment 2, 4, 6, 29

MVC Model View Controller 2, 3

Bibliography

[Gestwicki and Jayaraman, 2005] Gestwicki, P. and Jayaraman, B. (2005). Methodology and architecture of JIVE. *Proceedings of the 2005 ACM symposium on ...*, 1(212):95–104.

[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.