Contents

List of Figures

Chapter 1

Introduction

Upon starting their studies in computer science, new students may find the general concepts of programming, 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 attention in the school system – which is more concerned with the use of computers as tools, not how these tools are made – and only a few students have actually tried learning programming before beginning at 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 automatically generated code, can be hard to get a good understanding of. Traditional debuggers are not necessarily helpful when detecting a runtime error, as they tend to only point at where the error occurred, along with a classification. Depending on the language and tools being used, this error may be specific enough to, for example, indicate an indexing error, or so vague that it only says 'something went wrong here'. The result is often that a significant amount of time is spent searching for the cause of a bug, instead of actually fixing it [Ko et al., 2006].

The use of *Integrated Development Environment*s (IDEs) – tools that integrate source code editors with facilities for compiling and running programs, and intercepting errors that occur during run-time – can help with the challenge of finding the cause of an error, and might even suggest a way to fix it. But getting an overview of an entire program, and how its components interact with each other is not necessarily easier. Introducing 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 TDT4180 – *Human–Computer Interaction* (HCI). This course handles topics related to creating programs with a *Graphical User Interface* (GUI), and specifically how to implement a *Model View Controller* (MVC) architectural pattern 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 HCI-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 implementation 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 Eclipse IDE, make a common framework for many of the programming courses at NTNU, which makes compatibility with them an important criteria when looking for tools to support the students.

The MVC pattern describes a way of dividing graphical applications into a trinity of models, views and controllers. The models store data, and only accepts modifications from the controller. The controller acts as a mediator between views and models, ensuring a safe and consistent modification of the model, and informing views of any changes. The views are simply presenters of the data contained in the model. While the view can request data directly from the model, all modifications are relayed through the controller. This pattern provides several benefits, such as separation of responsibility, modularity and robustness, but it can also be complicated to implement properly.

1.1 Research questions

The goal of this report is to look into the available tools that can aid developers, concentrating on the potential use in the HCI-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 improvement, 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, so that the information they provide can be refined or presented in a more understandable way?*

Research Question 4 *Would the use of such tools and any improvements actually be useful for the students, and help them understand the internal structure, and order of execution, in a program?*

Chapter 2

Prestudy

This chapter 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

Debuggers are tools that are designed to aid a developer in the process of finding errors, or bugs, in a program. Depending on the language and platform, this ranges from gathering information when a program crashes, to inserting *breakpoints* – points in the code where the debugged program will be suspended, allowing the contents of its memory to be inspected, and potentially modified. While having obvious uses, these techniques requires a certain level of knowledge of both the program in question, and programming in general, to be really useful. The information provided by these debuggers is mainly textual, and that can itself be a limiting factor in the understanding of the programs [Larkin and Simon, 1987].

Larkin and Simon has shown that the use of diagrams can enable an individual to absorb the presented information faster than if it was purely textual. It is then natural to explore the generation of diagrams to visualize the structure of a program, and its path of execution, as a way of helping developers. The *Object Management Group* (OMG) has, with the *Unified Modeling Language* (UML) specification, established a widely used standard for how a large variety of diagrams describing software design and architecture should be formed. Of these, it is the class-, object- and sequence-diagrams that are of most use when describing the architecture and execution pattern of a program. Such diagrams can make it easier to get an overview of a program's current state, see the contents of objects and how they relate to each other, and to understand how the various components work together.

Class-diagrams show what classes, or objects, a program consists of. The classes are described with both the methods they contain, and connections they have to other classes, indicating relations between them.

Object-diagrams are similar to class-diagrams in that they both show the objects

of a program and their connections between each other. They differ in that while the class-diagram shows the connection between classes and what the classes are composed of, whereas 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 transitions between different 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. The active components are shown at the top, with vertical *lifelines* below them, representing time. The lifelines alternate between thin and bold lines to indicate whether or not they are involved in the execution at a specific point in time. As the components in the program invoke methods on each other, their respective lifelines are connected to each other with arrows.

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 during the planning of a system, as they are used to specify the general behavior of a system.

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 load time. The load time will be affected by the amount of data that must be analyzed, but there is no need to work through any of the steps in between. If the data is stored as a differential to 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. By adjusting the value of n to fit the characteristics of the system that the tool is running on, an appropriate balance between the two previous methods can be found.

The potential disadvantages of manual backstepping can be avoided by using queries instead. Queries enable the user to ask 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, it is possible to achieve a significantly smaller performance impact compared to a system that does real-time analysis of the data.

2.2 Existing tools

There are currently several existing tools that provide one or more of the methods mentioned above. The following list is not exhaustive, but includes some of the most relevant tools, considering the focus of this report. Some of the tools listed below does not support Java or Eclipse, but have interesting features that are worth mentioning, despite their incompatibilities with the desired teaching environment. Download links for the tools are listed in their respective glossary entries for the tools that are available.

***GNU debugger* (GDB)** A part of the GNU project, and maintained by the Free Software Foundation, it offers a tracing environment, and support for many languages, although the support for Java is limited. Due to its *Command Line Interface* (CLI), it is not necessarily easy to use on its own, and as a consequence, there are several front-end platforms that provide a graphical environment around GDB, including Eclipse.

Code Canvas [Deline and Rowan, 2010] Developed by Microsoft, uses an interesting way of visualizing an entire project, everything from source-code to design documents and diagrams are layered onto a large canvas. This allows the user to easily navigate between various elements, but this tool is restricted to Microsoft Visual Studio, and the languages it supports.

Trace Viewer Plugin [Köckerbauer et al., 2010] Developed by the MNM-team at the Ludwig Maximilian University of Munich, Germany. A plugin 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. These visualizations should make the execution easier to understand, but the plugin is designed for tracing the massively parallel programs that are used on high performance computing clusters, and requires a special version of Eclipse. Smaller programs, like the ones students write as a part of the exercises in the HCI-course, are not massively parallel, making the diagrams of them less useful.

Trace-Oriented Debugger [Pothier et al., 2007] Developed at the University of Chile in Santiago. Utilizes execution traces to enable its debugging and querying features. It offers high performance tracing, being able to maintain usable interaction while debugging complex software, but its only visual representation of programs is the 'mural', a graph that shows event density over time.

Whyline [Ko and Myers, 2009] Developed at the Carnegie Mellon University, Pennsylvania, USA. Like the Trace-Oriented Debugger, it makes use of execution traces to enable querying, instead of providing visualizations. It aims to explain why something happened in a program, and does so by looking at the history of the involved components. This tool only exists as a separate application, and does not integrate into any IDE.

Debug Visualization Plugin for Eclipse provides an alternative to the variable view in Eclipse's debug-perspective, producing a graph that represents the variables of a program. The users 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, as this plugin does not do any tracing. This was previously tested in the course TDT4100 – Object oriented programming, with the conclusion that the generated diagram quickly became too large, showing the contents of objects that were not important, and shuffling objects as new ones were added.

JAVAVIS [Oechsle and Schmitt, 2002] Developed at the University of Applied Sciences in Trier, Germany. Creates visualizations in the form of UML sequence- and object-diagrams, but does not include any debugging features.

Jinsight [De 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, and especially students, from using it.

***Java Interactive Visualization Environment* (JIVE)** [Lessa et al., 2010] Developed at the University at Buffalo, New York, USA. A tool that utilizes execution traces to generate diagrams while running a program. It is installed as an Eclipse plugin, and adds several new views to display the information it provides. In addition to generating diagrams, the trace log is also used to enable backstepping, which is coupled with the diagrams to always show the selected execution state.

2.3 Selecting a tool

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. While GDB supports both Java and Eclipse, the Eclipse-integration is mainly to enable the use of Eclipse as an environment for writing programs in C and C++. Since Eclipse already includes a powerful debugger for Java, it is not considered necessary to use another for the same purpose. As the other tools that were mentioned lacks support for Java, Eclipse, or both, they are all deemed unfit for integration with the desired teaching environment. The features provided by the remaining tools varies. Some are overlapping, like the tracing, while others offer completely different functionality, or combine a different set of techniques. As they are all plugins that integrate

into Eclipse, they both allow and encourage usage alongside the existing debugging functionality, instead of attempting to replace it. Users are also free to not use them, if that is what they desire.

The Trace Viewer Plugin is as mentioned, designed for massive parallelism, and requires a specific version of Eclipse, which has been discontinued. As small programs with a graphical interface, such as those that are used in the HCI-exercises, are typically not massively parallel, but instead almost entirely serial in nature, this tool does not fit the scope of the course. Due to this, and the fact that the plugin itself is no longer available, it is not suited for further study.

The Trace-Oriented Debugger provides a debugging environment supported by trace logs. It presents its information mostly in a textual way, and does not generate any visual diagrams of the program structure or execution order. It does offer a 'mural', a graph depicting the frequency of events through time, but this does not describe the program structure in any way. While providing support for developers by letting them look back at earlier stages of their programs, and quickly jump to events of interest, the lack of visual information can be considered a drawback.

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 in Eclipse. It does not perform any tracing, and instead uses the information exposed by Eclipse's debugger when the debugged program reaches a breakpoint to draw its diagram. As mentioned, this plugin has already been tested with another course. This testing concluded that while the information it provides is useful, the elements in the diagram does not stay put, causing confusion as they move around during execution. There were also complaints about too much information being displayed, making it hard to get an overview of the program, and identifying the important components. The latest version of this plugin does support the hiding of elements, as well as letting the user reorganize the diagram, but any such configuration seems to be reset when moving to a new breakpoint.

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 both a *contour-diagram* [Jayaraman and Baltus, 1996] – a diagram similar to object-diagrams – 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.

While the diagrams, like the diagram generated by the Debug Visualization Plugin, may suffer from the inclusion of too many objects, JIVE integrates a filtering mechanism to reduce the amount of objects included in both diagrams.

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 significant increase in 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 remaining chapters will be used to answer the remaining questions following an adaptation of the 'design science' process.

As described in [Vaishnavi and Kuechler, 2004], design Science involves a five-step process consisting of the following phases: Awareness of the problem, suggestion, development, evaluation and conclusion. These phases can be revisited during the process, resulting in an iterative cyclic process. Repeating previous steps can often be necessary as more knowledge of a subject is acquired, and previous assumptions must be reconsidered.

While the report will generally follow this process, some of the steps are suited for adjustments to better fit the task. Each of the steps are summarized below, with an explanation of the intended deviation from the normal process.

3.1 Awareness

Gaining awareness of the problem is necessary to be able to solve it. In this case, the problem will be any issues discovered with JIVE. Through a thorough exploration of JIVE and its features, it is a fair assumption that some of the most apparent issues will be revealed. This exploration is also intended to better understand whether JIVE can be properly integrated with the teaching environment, or if any modifications are required.

While it is likely that several issues will be revealed, it is natural to continue the process of exploration instead of moving on to the next step in the design science process and repeating the entire cycle for each issue that is revealed. This results in a focused exploration-phase, and likely a more efficient use of time.

3.2 Suggestion

The same all-at-once approach will be taken when considering suggestions to solve the discovered issues. During both this and the previous phase, it is necessary to

prioritize the various issues according to the evaluated importance and estimated effort required to solve the issues. This is mainly to provide focus for the next phase, development, where it is assumed that most of the time will be spent. Proper prioritizing will help towards an efficient use of the available time.

3.3 Development

As opposed to the previous two steps, the attention is directed at single issues during the development-phase. In this phase, it is expected that the awareness- and suggestion-phases will be revisited, triggered by the discovery of technical details that change the feasibility of a solution. Partial implementations will be subject to internal evaluations where specific solutions are considered, and the awareness of the problem is updated accordingly.

3.4 Evaluation

While each of the issues that result in an implemented change goes through internal evaluation during development, there is also need for external evaluation. This is another phase where it is suitable to gather all issues, as they are all part of the package that is JIVE.

In order to determine the usefulness of JIVE, and the effect of the implemented changes, the external evaluation will be performed with participants from the targeted user group. As mentioned in the introduction, this group consists of students in their second year of the computer science and informatics programs.

The types of evaluations available are limited, with web-based questionnaires and in-person interviews being most suitable.

Reaching a wide audience, the use of web-based questionnaires can provide a large amount of data to support the research, but the nature of what is being evaluated makes this method unfit. 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. Although they could be tasked with comparing images of existing and new functionality, and given performance numbers to consider, it would not be the same as actually experiencing the differences, and getting to explore them at the users own pace.

Another issue appears in how questions are formulated, and what kind of answers they result in. Requiring participants to quantify their experiences along numbered scales makes it easy to combine the results from all participants. On the other hand, as each question must be limited in scope in order to get a precise answer, the amount of questions needed to get a good overview of each participants experience, can easily be perceived as different ways of asking the same question, resulting in repeated answers.

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. As the availability of volunteers is limited, 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.

A hybrid approach could also be used by performing an in-person evaluation, and requiring participants to fill in a questionnaire afterwards. This would still require one or more pre-configured systems, and some time spent on each participant to give an introduction to the tool they are evaluating. Doing this in an effective manner, requires multiple assistants, and while the meaning of the questions could be further explained upon request, the answers would still be limited in the same manner as with a web-based questionnaire.

3.5 Conclusion

The final phase will summarize the results from the previous steps in light of the research questions, and attempt to reach a conclusion on the research. This is again a collective step, where all issues are considered as a whole. Focus will be on whether the previous steps have been able to provide answers to the research questions, and the success in that regard. Some attention will also be given to consider whether other approaches might have been better suited, and in what direction any future research should look.

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

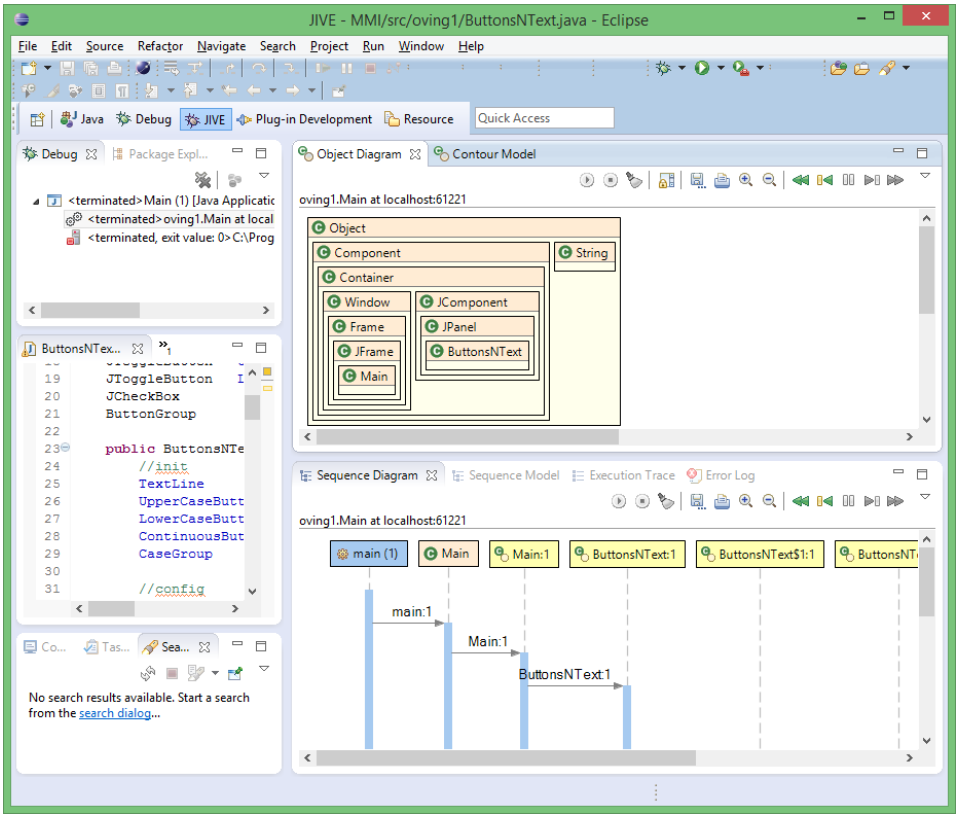


Figure 4.1: The JIVE perspective in Eclipse.

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 that are generated during debugging the two most apparent features of JIVE. 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*.

4.1.1 The object diagram

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 object's variables are shown, with name, type and value. The contour also uses arrows to point at other contours that are related, e.g. another 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. This can be especially useful when working with larger programs that are composed of 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.

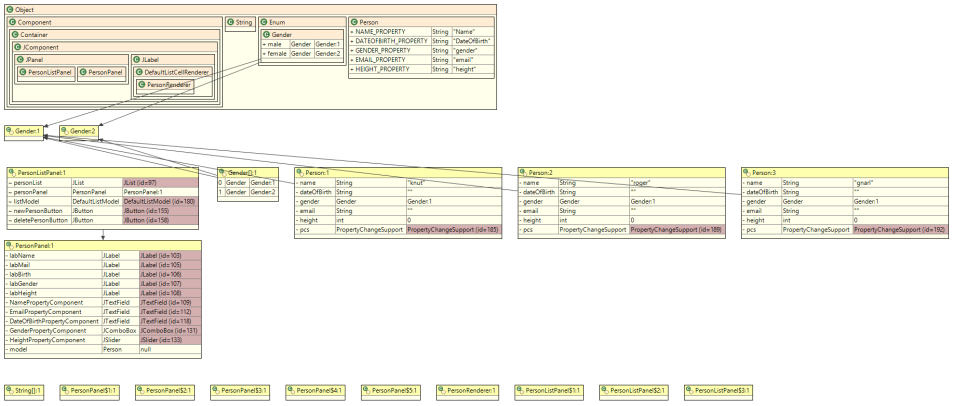


Figure 4.2: A contour diagram generated by JIVE.

4.1.2 The sequence diagram

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, the two colors representing the 'main'- and the 'AWT-event-thread' are clearly visible, and how elements in the diagram are colored by their parent thread.

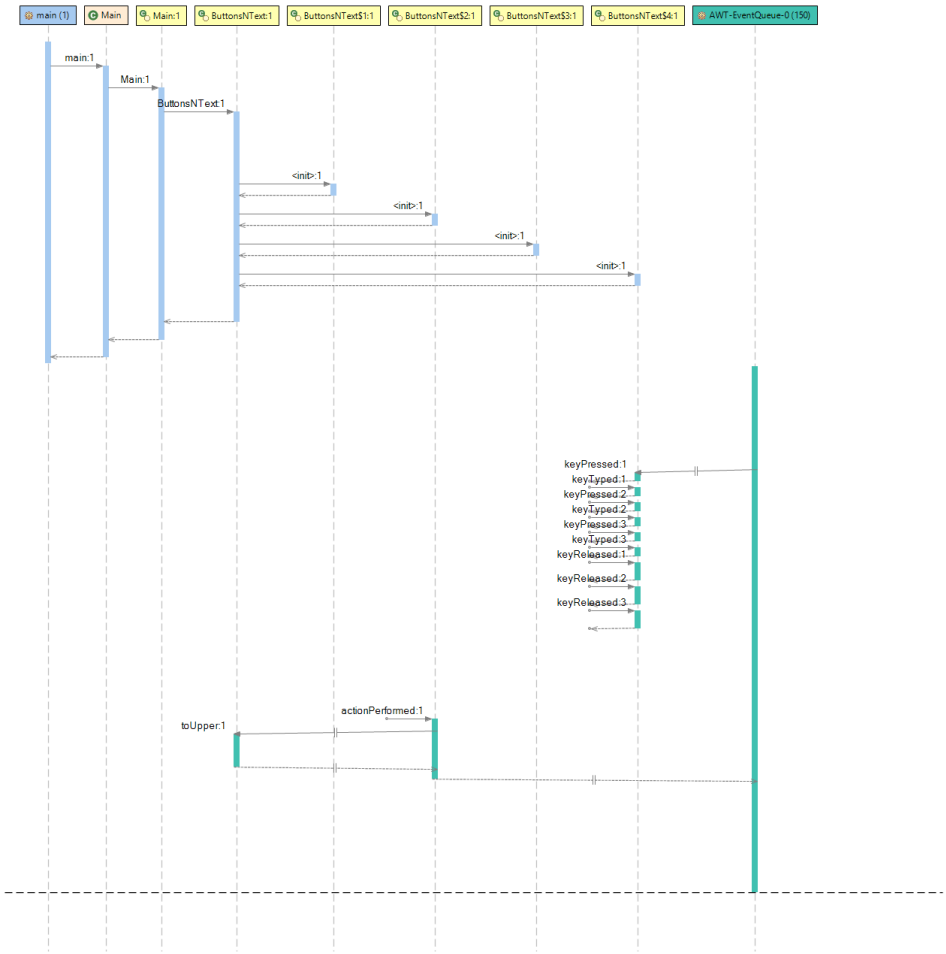


Figure 4.3: A sequence diagram generated by JIVE, while running an instance of HCI-Exercise 1.

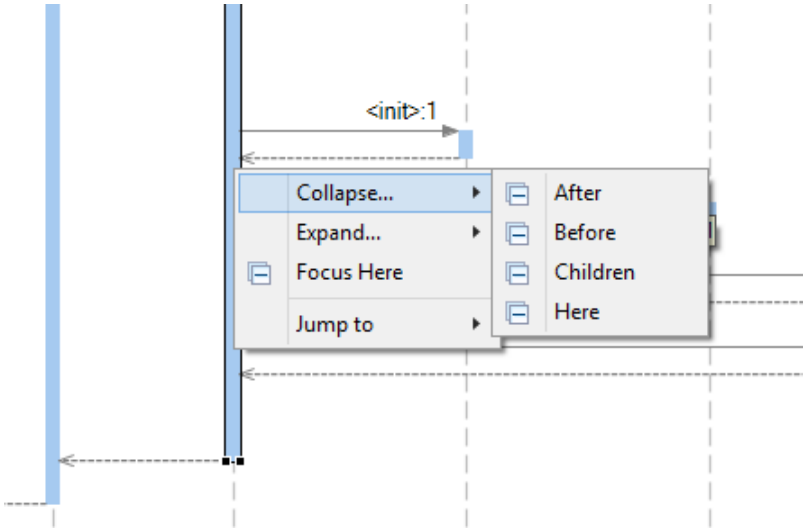
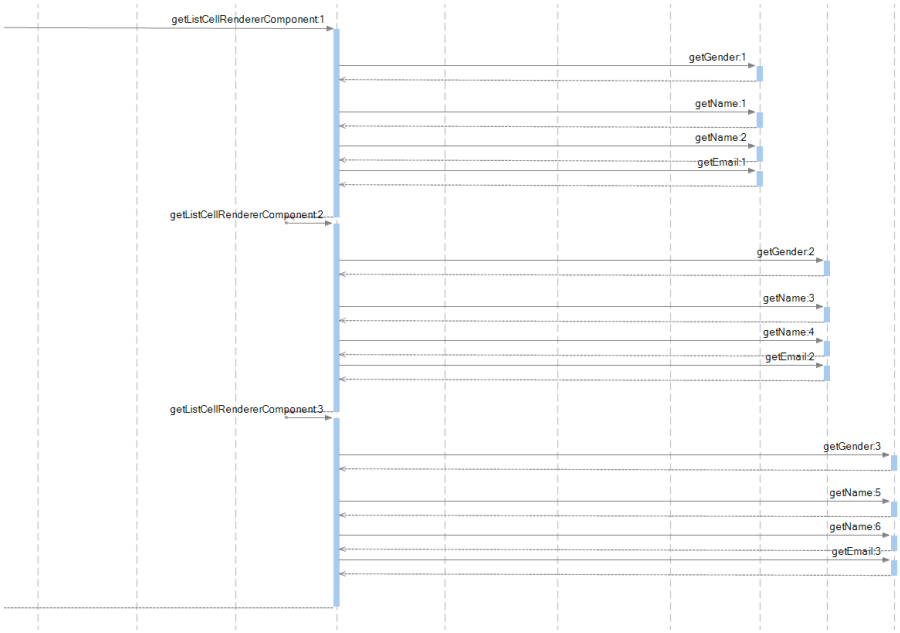


Figure 4.4: The sequence diagram right-click menu.

Right-clicking on a lifeline provides the ability to collapse method-calls originating from that lifeline in order to hide unnecessary information, as shown in figure 4.4. The collapse-menu gives 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 gives the opportunity to collapse that object's entire lifeline, the result of which can be seen in figure 4.5. While this figure 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 also available for expanding collapsed elements. Right-clicking also provides to set the execution-state via the *Jump to*-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. Note the '+' symbols in (b), at the top of each bar, indicating a collapsed element.

Both of the diagrams can be saved as an image at any time, enabling the user to look at diagrams from earlier runs, instead of being forced to view them through JIVE. these images can also be useful as documentation. 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 have the ability to zoom in and out, further helping with the handling of larger diagrams.

4.1.3 Time-stepping

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 *eXtensible Markup Language* (XML)- and *Comma Separated Values* (CSV)-files for later use.

When jumping between execution sates, both of the diagrams are updated accordingly. The contour diagram shows a representation of that state, while a horizontal line is added to the sequence diagram to show where the state is located on the time-line.

4.1.4 The exclusion filter

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 hide or show details of the execution, respectively.

JIVE maintains an individual filter for each *debug launch configuration* in eclipse, all based on a default filter. This makes it possible to have different filters for the same program, with for instance varying degrees of detail, or different areas of focus.

4.1.5 The model views

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.

4.1.6 Searching

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 within, and the value.

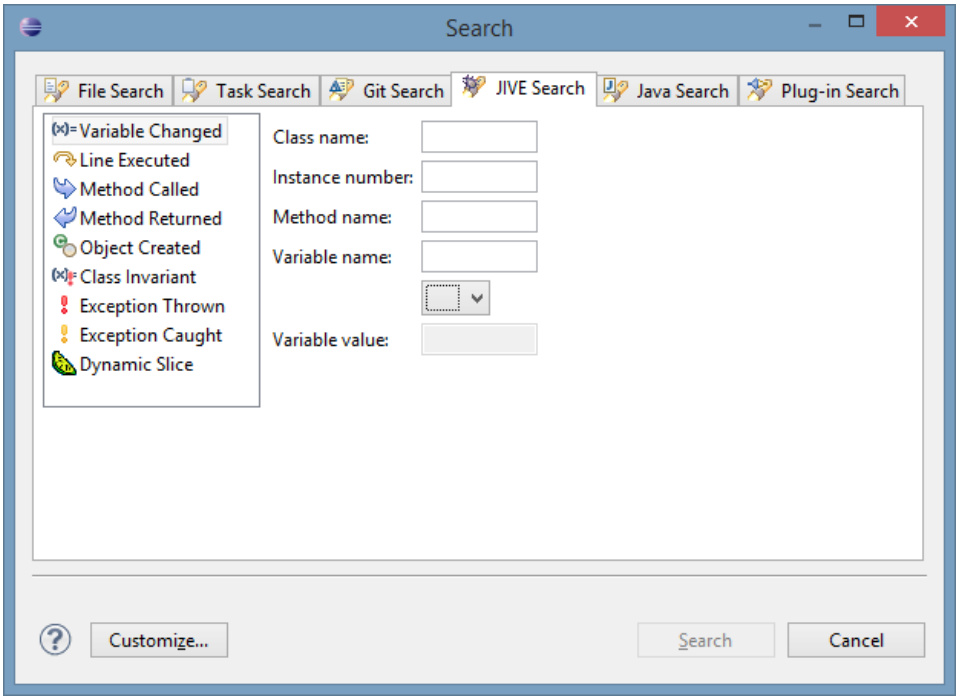


Figure 4.6: The JIVE search panel.

4.2 Suggested changes

While exploring the features of JIVE, a few issues regarding use and behavior were encountered. None of these issues can be considered to be of major importance, they still expose a potential for improvement. This section will point out some of the issues, and suggest specific improvements to address them.

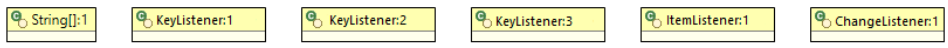
4.2.1 Visual changes to the diagrams

In both the sequence- and the contour-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 an 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 its actual name, should help with this issue. This should make listener-heavy programs 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. 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. This is partially caused by the design of the Swing-framework, as a component does not have direct control of its listeners, but instead delegates the handling of listeners to another object. Additionally, JIVE does not show inherited fields unless the ancestor-object is let through the filter, and as listeners are typically handled by inherited methods, this adds another series of types that must get through the filter.

On the other hand, the effect caused by the listener being triggered is clearly visible in the sequence diagram. It is possible that connecting this effect with a user's interactions with a program is simple enough for the user to not need the addition of a specific link.

(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 must 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, or 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. It can not be expected that users will adopt such a scheme for the sole purpose of using a specific tool, consequently, this solution is not pursued. The highlighting could be done by adding colored frames to the identified objects, or by adding an icon indicating the role of a given object. While highlighting is not as big of a challenge compared to the identification, it does pose the problem of maintaining the readability of the diagrams, and not make them harder to read due to the added elements.

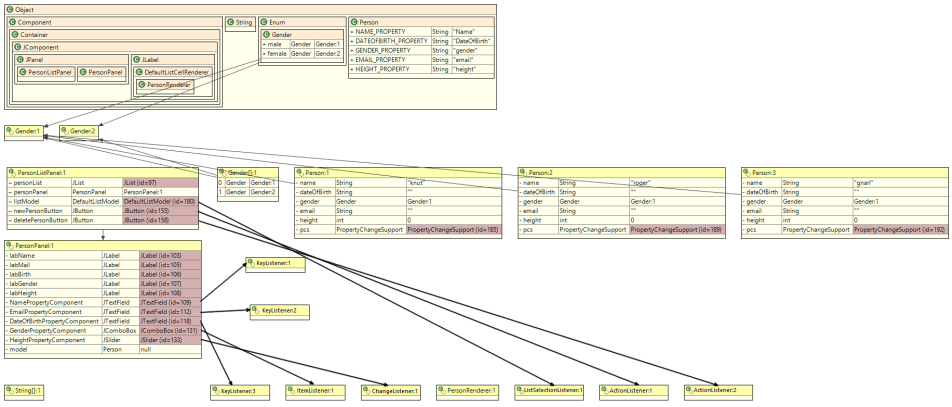


Figure 4.8: Listeners are relabeled, and connected to the objects they listen to. The illustration is based on figure 4.2.

4.2.2 Compacting diagrams

Both the sequence- and the contour-diagrams can easily grow to sizes that make them hard to get an overview of, and to navigate. 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 there are still some issues that result from 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 of ordering 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.

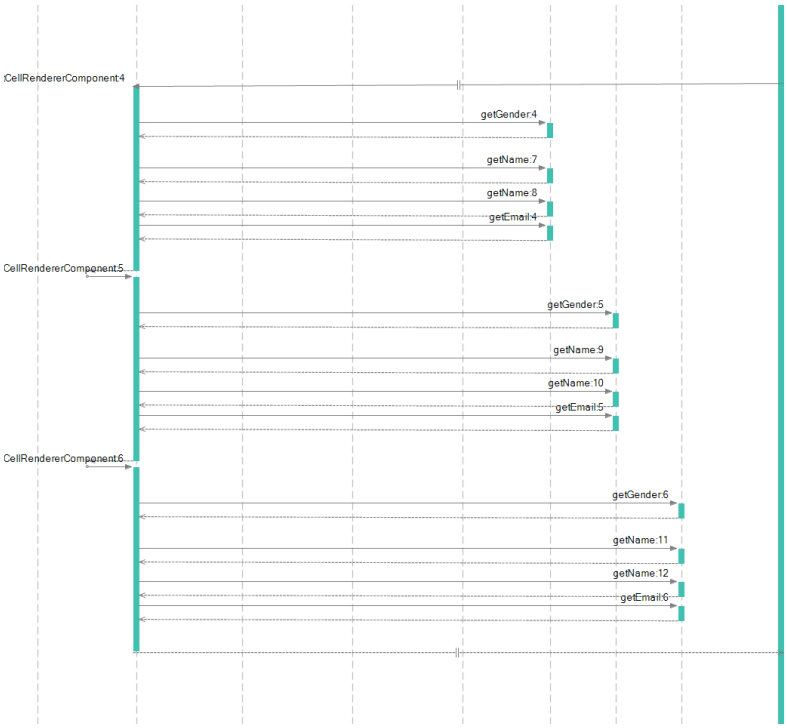


Figure 4.9: A section of a larger sequence diagram, showing method calls crossing several unrelated lifelines.

In some cases, the amount of unrelated lifelines that surround a sequence becomes so large, that fitting the entire sequence within the view of a single screen is 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.

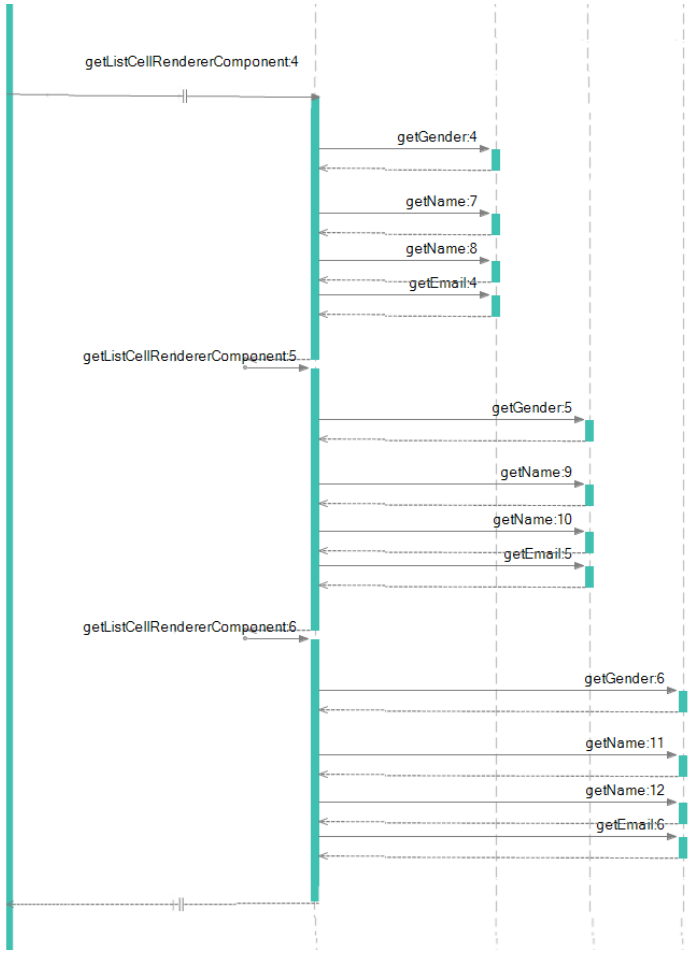


Figure 4.10: An example of how figure 4.9 might look in the isolated view.

4.2.3 Changes to the filter

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. For example, 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.

Changing the filter is currently done by adding or removing one entry at a time, making large changes time consuming and prone to spelling errors. Since the filters are stored in the XML-files Eclipse uses to describe how to launch a program, directly modifying these files is a possible work-around for making large changes, but they require Eclipse to be restarted to have any effect. These files are also hidden within the configuration folders that Eclipse uses, making them hard to find without instructions. Having an easier way of swapping, or making large changes to filters, would likely increase the use of custom filters, and would enable the distribution of pre-made filters, designed for specific purposes. This feature could be extended further, to include multiple default filters that a user could select from when running a program.

4.2.4 Searching

The ability to search through all logged events can be a 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. The search is unfortunately very strict in its interpretation of the search terms, being both case-sensitive, and requiring full class names, including the path to where the class resides. 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 confusion. At worst, users may be led to believe that the search is 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.

4.2.5 Automation

Despite the measures that have been taken to reduce the performance impact of JIVE, it will always be possible to create a scenario where the analyzed program will take a significant amount of time to run. For instance when testing a larger project. For non-interactive programs, this is only a matter of waiting for them to finish. But in order to get the desired results from an interactive program, it becomes necessary to check back regularly in order to provide the correct input. Integrating automation tools that trigger pre-defined inputs when they detect that the program is waiting for interaction, would go a long way towards solving this. A user could first run a session without JIVE, and record the desired interactions. These interactions would then be fed into the automaton tool and played back during the JIVE-run. As this kind of feature is not required for the exercises given as a part of the HCI-course, it is not prioritized above the other changes.

Chapter 5

Implemented changes

This chapter will list the changes that were made to JIVE, and give some information on how they work.

In order to be able to implement any changes, it was first necessary to gain some knowledge about how Eclipse's plugin-framework is used, and how JIVE is structured. The graphical components of JIVE utilize the *Graphical Editing Framework* (GEF), an MVC framework used to create graphical views and editors in Eclipse.

5.1 Labeling of objects

The first change to be implemented was the identification and presentation of instantiated interfaces. These are now displayed in the diagrams with an appropriate icon, as well as being labeled with the interface they implement, instead of the generic class name that they are 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, and displays the actual class name and icon.

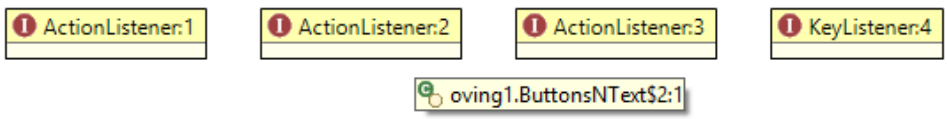


Figure 5.1: A view of how instantiated interfaces are shown after the modification. Also shown is the tool-tip text for the second instance from the left. Originally, they would be labeled as 'ButtonsNText$1:1', 'ButtonsNText$2:1' and so on, with the green class-icon.

5.2 Expanding the filter

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 for inclusion is done by prefixing the package-name with a '+' when adding it to the filter, as shown in figure 5.2. 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, every subpackage of 'javax.swing' is let though the filter as well, which is the intended design.

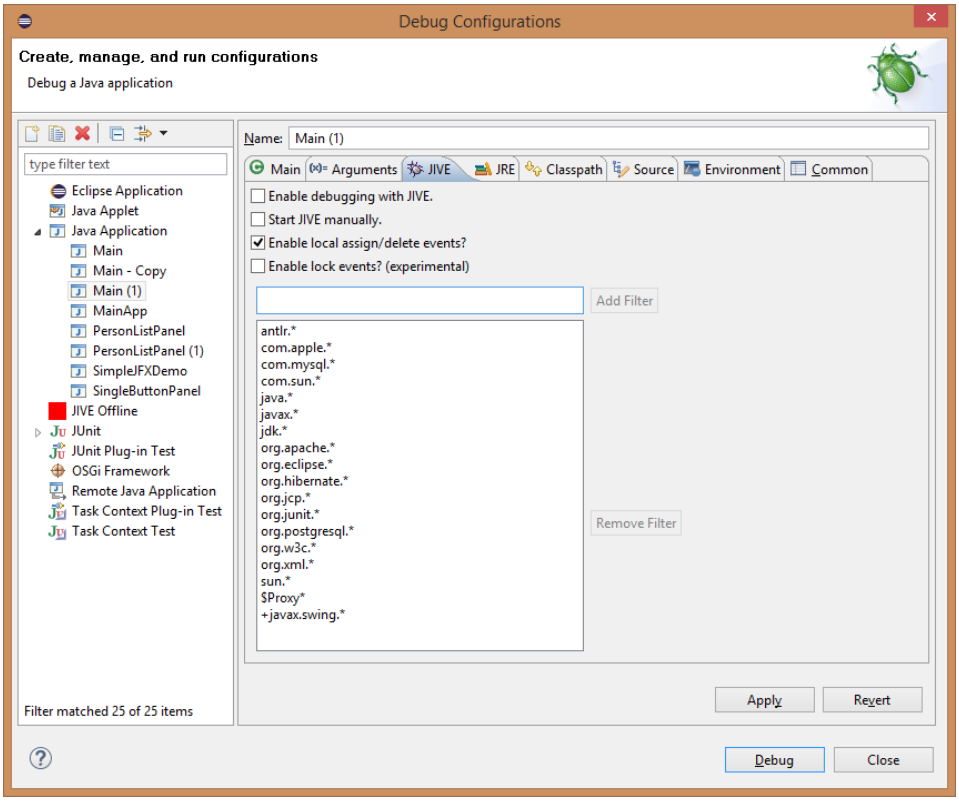


Figure 5.2: The JIVE-tab in the launch configuration menu, showing the default exclusion filter, with the addition of an entry to include 'javax.swing.*'.

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 added to the underlying model – as well as cluttered diagrams. By adding these unwanted classes to the filter for exclusion, the problem is handled. 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. A weakness in the filter is related to the problem of unwanted packages. Due to its design, allowing the contents of a package will also allow any classes that are contained directly within the parent package, as the parent 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 most likely other packages that will show this behavior.

Internally, the filter is still only excluding events. The changes were made to the construction-phase, where every package that is found in the Eclipse workspace is added if it is found in the list that the user has made. This allows the removal of entries that are marked with a '+', but results in a rather large filter that affects the performance negatively, as is shown in the evaluation results in section 6.2.1. During development, the performance was not a major concern, and as such, it was not prioritized. It is possible to reduce the impact of the filtering changes by further altering the construction process to result in a smaller filter. By improving the way an event is checked against the filter, possibly by looking up hashed values instead of comparing strings, further gains should be possible.

5.3 The isolated view

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 5.4 shows what figure 4.10 looked like after the modification. Figure 5.5 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 5.3, 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 which parts of the execution are visible.

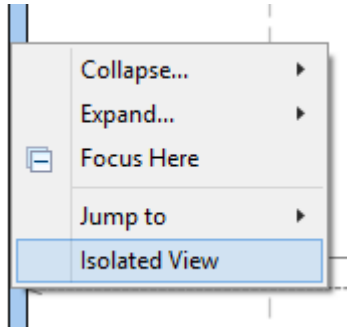


Figure 5.3: The new *Isolated view* option.

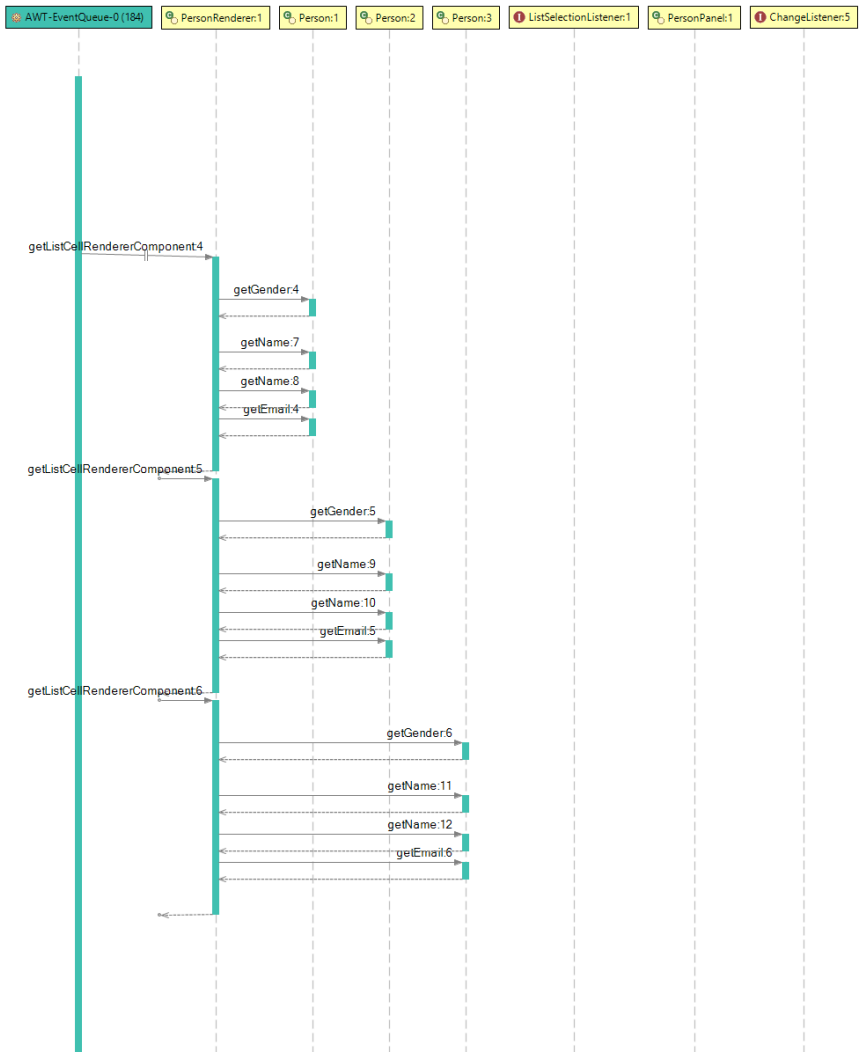


Figure 5.4: How figure 4.9 looks in the isolated view. The three last elements shown on the top are referred to in parts of the diagram not shown. This view was achieved by initiating the isolated view from the 'AWT-EventQueue' lifeline.

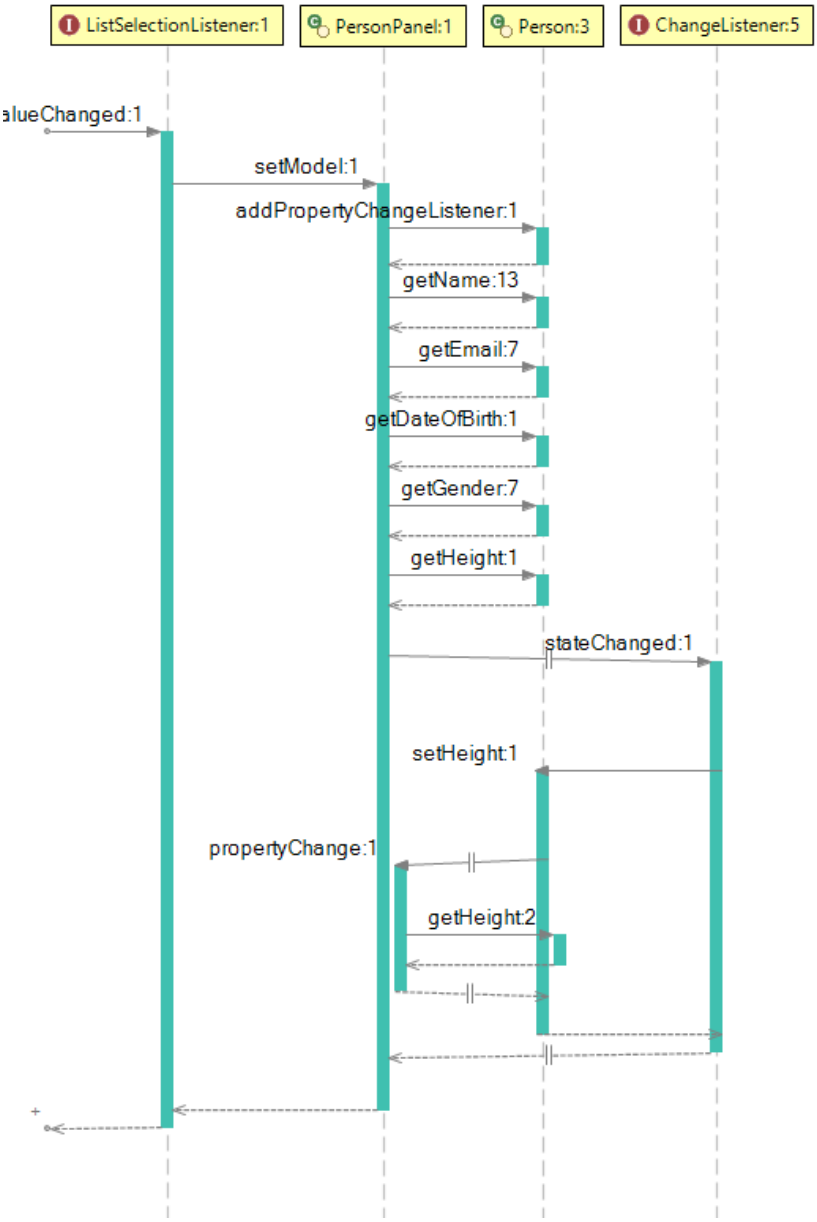


Figure 5.5: Another example of the isolated view, showing a significantly reorganized diagram.

5.4 Relaxing the search

The searching functionality was relaxed by allowing partial matches, and disabling 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 in order to prioritize more important changes. The only changed search was the *Object created*-search. Ideally the searches should all be matched through the same matching method, only differing in the gathering of necessary information.

Chapter 6

Evaluating usefulness

This chapter will look at the external evaluation of JIVE and the implemented changes, providing details on both the execution of the evaluation, and the results that were acquired.

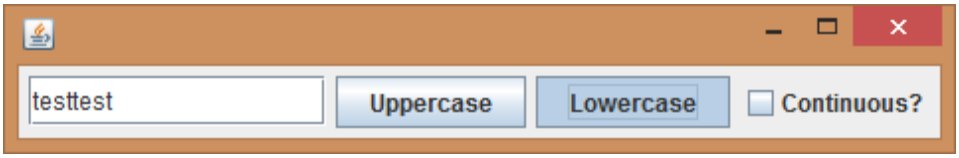
6.1 The test

In order to evaluate both the usefulness of JIVE, and the changes that were implemented, the evaluation was performed individually with a small group of six students. The participants were initially recruited through a request sent to the participants of a second-year course that is mandatory for those who follow the informatics program. This group was selected as they were matching the target group outlined in chapter 1. After only getting a single response, another request was sent to the students association of the computer science program, bringing in the five remaining participants. The latter group were all in their third year, and had already finished the HCI-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 during the course.

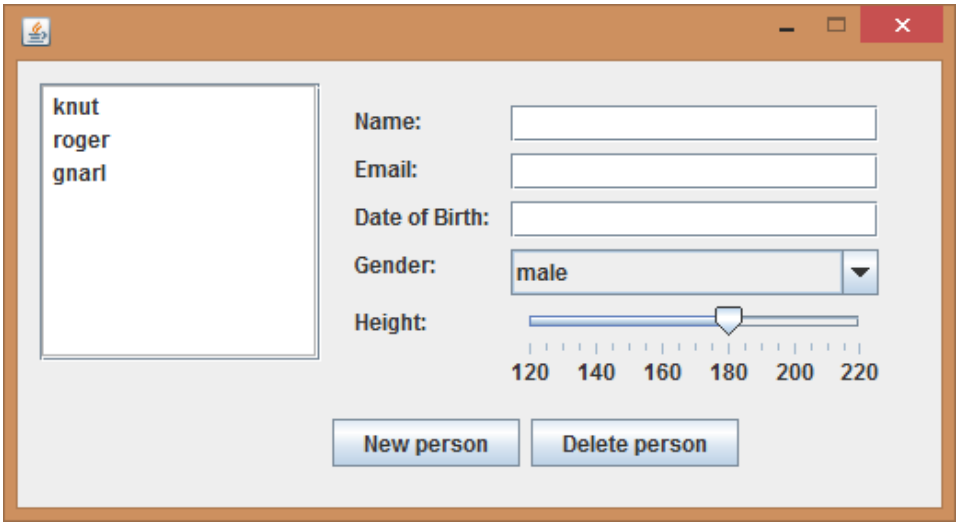
During the test, the students were given a demonstration of the main features of JIVE, using implementations of the first and the last exercise of the HCI-course as examples. After getting familiar with JIVE, they were shown the modified version, focusing on the new and modified features that were implemented. 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 continually transform as new text is written. Exercise 4 is an application that manages a simple list of people. The interface it presents is split between a list of people, and a set of text fields to enter information about the selected person, as well as buttons to create new, and delete existing people. Both of these programs, shown in figure 6.1, were relatively simple, with few events following directly from 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 HCI-course.



(a) HCI-exercise 1



(b) HCI-exercise 4

Figure 6.1: The two example programs used in the evaluation.

6.2 Results

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 that was showing more 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.

6.2.1 Performance

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 was measured as the time between starting the program, and the moment it could be interacted with. Each of the example programs were run 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. They were tested separately from the user evaluation, but on the same computer. Each result was measured five times, with the averages reported in table 6.1.

Table 6.1: Timing results from benchmarking

Version of JIVE	Program	5-run average
Original JIVE	Exercise 1	5.0 s
Modified JIVE, standard filter	Exercise 1	16.5 s
Modified JIVE, swing-focused filter	Exercise 1	18.3 s
Original JIVE	Exercise 4	8.2 s
Modified JIVE, standard filter	Exercise 4	25.0 s
Modified JIVE, swing-focused filter	Exercise 4	54.4 s

As these results show, there is a significant degradation in performance when going from the original version of 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 allowed 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, the user 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, the more interesting or complex parts could more easily be shown 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.

6.2.2 New issues

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. Along with this, there was a desire to add a suggestion-functionality that would behave similarly to the autocomplete feature found in many search providers, including parts of Eclipse. This feature would suggest package-names as the user started typing in the text field to add a new entry.

Another issue was the risk of diagrams becoming so large that they would be nearly impossible to navigate. The size would especially become a problem when running larger programs, or when letting more objects through the filter. 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 the user 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.

Following along the same lines, it was suggested that the placement of objects in the contour diagram could be reworked to give a more compact diagram. Figure 6.2 illustrates the problem, with a lot of unused space, and especially bottom row of objects extending far beyond the rest. If the size of the diagram was restricted, for instance by preventing it from growing wider than the top row, preferring multiple shorter rows of objects instead of a few long ones. This would let users see more

information without scrolling the view of the diagram or zooming out.

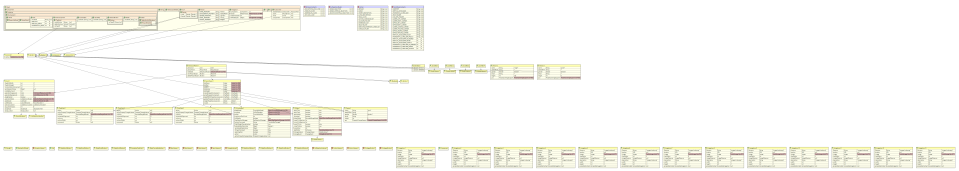


Figure 6.2: Contour diagram that has grown unnecessarily large.

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 the desired objects, and then scroll down to find the first event it is involved in.

Chapter 7

Conclusion

This chapter will summarize the findings of the previous sections in light of the research questions that were put forth 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 tools and execution tracers. This paper has focused on the tools that provide information from programs as they are running, and has found that most of them are based on a limited set of techniques to provide their users with information. The techniques in use are mostly based on trace-logging, with an analysis of the log after the program has finished execution. This analysis is then used to provide a detailed description of the program's behavior, often with interactive navigation of the data. Other techniques involve visualization with graphs and diagrams, either based on trace-logs, or on run-time-data captured when a program is suspended at a breakpoint. Not all of the discovered tools were available for download, and might be considered abandoned, while others are still under active development.

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

Among the tools that were examined, a variety of platforms were supported. Due to the focus of the report, most of them were designed around supporting programs written in Java, and a majority of these were integrated with Eclipse by design. As such, they could be integrated with several courses at NTNU, both through use during lectures, or by making them a part of the recommended tools. This would cover most students participating in a course, except those who actively choose not to use them. The most promising tool, JIVE, chosen for its combination of

tracing and diagrams, has already been successfully used to explain design patterns to students in a graduate-level seminar [Gestwicki and Jayaraman, 2005, p. 99], 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, so that the information they provide can be refined or presented in a more understandable way?*

Section 2.3 concludes that JIVE is the tool that is most suitable for use in the target environment, and, as such, it was selected for further study. As detailed in chapter 4, there is a potential for improvements in how JIVE works, and what kind of usage it encourages. While the existing features are useful as they are, some of them were found to be insufficient in terms of usability. Certain classes were considered to have a confusing naming scheme that made their 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. Other potential improvements were identified during the evaluation, and are detailed in section 7.2.

Research Question 4 *Would the use of such tools and any improvements actually be useful for the students, and help them understand the internal structure, and order of execution, in a program?*

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 within the programs they are creating. While the exact use case varied, all of the students that participated in the evaluation saw some situation where JIVE would have been useful. Some referred to situations they experienced when they were taking the HCI-course, and to experiences as a student assistant, while others imagined the use for generating accurate documentation for projects.

The implemented changes were all well received, and were considered to be an improvement upon the official version of JIVE. The participants also expressed desires for further changes, some of which were already identified, but not implemented, and some that could be considered as new ideas.

7.2 Future work

This report has explored some of the currently available tools that are designed to help developers get an insight into the software they are working on, focusing on usage in connection with education. Through the exploration of JIVE, several potential points of improvement were identified, but not implemented due to prioritizing within a limited time frame. Additional improvements were discovered by the evaluation participants during the evaluation. These modifications are listed below, in no particular order:

- Less restrictive search
- Further improvements to the filtering performance and granularity
- Improving the way the filter is edited
- Compression of sequence diagram height
- Making the top of the sequence diagram visible at all times
- Restricting growth of contour diagram
- Further exploration of indicating indirect connections in the contour diagram

The usage and performance of the filter is an important point to continue working on. As the filter directly affects the experience of JIVE, both by reducing the size of the trace-log, and through its performance impact, this is the key factor in enabling the use of JIVE for larger programs. While the live analysis that JIVE does makes it unlikely that performance will reach the point where continuous use can be considered, a significant reduction in startup- and response-time can be achieved.

With larger programs, the search function becomes more important. As detailed in section 4.2.4, the relaxation of the search terms will make the behavior more similar to the behavior expected of other search providers, causing less confusion.

Apart from the implementation of additional modifications, it would be desirable to start the use of JIVE during lectures, and encourage students to try the tool for themselves. This can be considered a natural next step in gaining more data on its use, in order to properly understand how students can benefit from such a tool, as well as identifying and prioritizing further refinements of JIVE.

Glossary

Breakpoint An instruction that is inserted into the source code, telling the processor to halt program execution at a certain point, and allowing a debugger to inspect the state of that program.

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/`

Comma Separated Values A common plain-text file format for storing values, separated by a comma. Sometimes called character separated values, as the separator is not necessarily a comma.

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

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

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

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

EXtensible Markup Language A markup language defining a set of rules for encoding documents in a format readable for both humans and machines.
`http://www.w3.org/TR/REC-xml/`

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

Graphical Editing Framework A framework for the IDE Eclipse. Used to create graphical views and editors for diagrams.

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

Human–Computer Interaction The study of design and use of interfaces between humans and computers. It is also the name of the NTNU-course TDT –4180 .

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

Java Interactive Visualization Environment An advanced debugging tool supporting visualisation, backward stepping, and querying. As of the writing of this report, the latest version is 1.9.29.v20121216b
`http://www.cse.buffalo.edu/jive/`
[Lessa and Jayaraman, 2010][Czyz and Jayaraman, 2007]

JAVAVIS Tool that generates UML-diagrams from running Java applications. Not to be confused with the computer vision library with the same name.
[Oechsle and Schmitt, 2002]

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

Lifeline A essential component of sequence-diagrams. Indicates when, during execution, an object is active in a program.

Model View Controller A design pattern defining an architecture where data is kept separate from the logic that is responsible for the modification of data, as well as the components that present the data.

Object Management Group An international, open membership, non-profit computer industry standards consortium. Most notably responsible for the development of the UML standard.

Trace Viewer Plugin Tool to visualize and analyze communication of parallel message passing programs.
[Köckerbauer et al., 2010]

Trace-Oriented Debugger Trace-Oriented Debugger. A debugging tool that executes queries on program traces.
`http://pleiad.cl/tod/`
[Pothier et al., 2007]

Unified Modeling Language A language describing the components needed to create models of a system, as well as defining how these models should look.
`http://www.uml.org/`

Whyline A query-based debugger that provides an easy way to find out why things are as they are.
`http://www.cs.cmu.edu/~NatProg/whyline-java.html`
[Ko and Myers, 2009]

Acronyms

CLI Command Line Interface

CSV Comma Separated Values

GDB GNU debugger

GEF Graphical Editing Framework

GUI Graphical User Interface

HCI Human–Computer Interaction

IDE Integrated Development Environment

JIVE Java Interactive Visualization Environment

MVC Model View Controller

OMG Object Management Group

UML Unified Modeling Language

XML eXtensible Markup Language

Bibliography

[Czyz and Jayaraman, 2007] Czyz, J. and Jayaraman, B. (2007). Declarative and Visual Debugging in Eclipse. *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 31–35.

[De Pauw and Vlissides, 1998] De Pauw, W. and Vlissides, J. (1998). Visualizing Object-Oriented Programs with Jinsight. *Object-Oriented Technology: ECOOP'98 Workshop Reader*, pages 541–542.

[Deline and Rowan, 2010] Deline, R. and Rowan, K. (2010). Code Canvas : Zooming towards Better Development Environments. pages 207–210.

[Gestwicki and Jayaraman, 2005] Gestwicki, P. and Jayaraman, B. (2005). Methodology and architecture of JIVE. *Proceedings of the 2005 ACM symposium on Software visualization*, 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 et al., 2006] Ko, A., Myers, B., Coblenz, M., and Aung, H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on 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*, pages 1569–1578.

[Köckerbauer et al., 2010] Köckerbauer, T., Klausecker, C., and Kranzlmüller, D. (2010). Scalable Parallel Debugging with g-Eclipse. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, pages 115–123. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Larkin and Simon, 1987] Larkin, J. H. and Simon, H. A. (1987). Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive science*, 11(1):65–99.

[Lessa and Jayaraman, 2010] Lessa, D. and Jayaraman, B. (2010). Temporal Model for Debugging and Visualizations. *128.205.32.53*.

[Lessa et al., 2010] Lessa, D., Jayaraman, B., and Czyz, J. (2010). Scalable Visualizations and Query-Based Debugging. *coggsworth.cse.buffalo.edu*.

[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). In *Software Visualization*, pages 176–190.

[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. T. 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 on Computer science education*, pages 510–514.

[Vaishnavi and Kuechler, 2004] Vaishnavi, V. and Kuechler, B. (2004). Design Science Research in Information Systems.

Appendix A

Accessing the source code

A.1 JIVE

The source code of JIVE can be acquired by following the installation instructions at `http://www.cse.buffalo.edu/jive/download.html`. After the installation is complete, a set of jar-archive files can be found in Eclipse's plugin folder, which is located in the user-directory. The plugin is divided into several archives, representing the plugin fragments. The source code for each fragment can be found within the respective archives, and can be extracted with an appropriate program. After extracting the code, it can be imported into Eclipse as a project, note nat all fragments are needed to successfully run the plugin after import.

A.2 The modified version

The modified source code can be found at the following repository at Github: `https://github.com/Skabbkladden/Meister/`, within the folders `code/JIVE/`.