JOHANNES KEPLER
UNIVERSITY LINZ

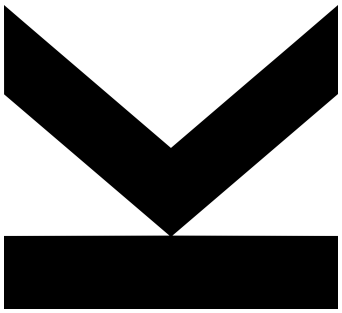
Author
Melissa Sen
k12102743

Submission
Institute for System Software

Thesis Supervisor
a.Univ.-Prof. Dipl.-Ing. Dr.
Herbert Prähofer

June 2024

A Component for Visualizing Sequence Diagrams in the Visual Debugger Tool JavaWiz



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Programming beginners often face difficulties when trying to understand how programs behave. As soon as code gets too complicated, novice programmers struggle with figuring out the purpose and flow of actions. This is when JavaWiz comes into play. JavaWiz is a tool that provides dynamic visualizations of a Java program's behavior. So far, there are visualizations available for a desk test, the stack and heap, flowcharts, linked lists, binary trees, arrays, input buffers and the console.

As object orientation is a relevant programming concept, this thesis presents a sequence diagram component for JavaWiz. Sequence diagrams show the chronological order of object interactions and help users to understand method invocations. Each step in the program results in the dynamic adaptation of the sequence diagram to reflect the current state. Moreover, users have the opportunity to collapse and thus hide the contents of method executions they are not interested in.

Kurzfassung

Programmieranfänger haben oft Schwierigkeiten zu verstehen, wie sich Programme verhalten. Sobald Code zu kompliziert wird, kämpfen unerfahrene Programmierer damit, den Zweck und den Ablauf von Aktionen nachzuvollziehen. Hier kommt JavaWiz ins Spiel. JavaWiz ist ein Werkzeug, das dynamische Visualisierungen des Verhaltens eines Java-Programms bereitstellt. Bisher stehen Visualisierungen für einen Schreibtischtest, den Stack und den Heap, Flussdiagramme, verkettete Listen, Binärbäume, Arrays, Eingabepuffer und die Konsole zur Verfügung.

Da Objektorientierung ein relevantes Programmierkonzept ist, stellt diese Arbeit eine Sequenzdiagrammkomponente für JavaWiz vor. Sequenzdiagramme zeigen die chronologische Reihenfolge der Objektinteraktionen und helfen dem Anwender, den Ausführungsablauf zu verstehen. Jeder Programmschritt führt zur dynamischen Anpassung des Sequenzdiagramms an den aktuellen Zustand. Außerdem haben die Nutzer die Möglichkeit, Methodenaufrufe zu kollabieren und somit die Inhalte, an denen sie nicht interessiert sind, zu verbergen.

Contents

List of Figures

1 Introduction

1.1 Motivation

Programming beginners are often more familiar with procedural techniques as it describes a straight-forward, top-down approach of writing code. This is helpful to understand imperative programs, however, it may be difficult to understand object-oriented programs. While procedural approaches focus on splitting a program into multiple functions, object orientation defines objects that interact with each other. Each object represents an instance of a class that contains closely related data and methods. Programs using the concept of objects allow modeling real-world entities as they incorporate ways of data hiding, abstraction and code reusability [1].

Sequence diagrams can support in understanding how objects interact with each other. They are defined in the Unified Modeling Language (UML) standard [2] for visualizing a system's behavior based on the chronological order of object interactions.

In UML, an actor initiates the execution of a program. The system is then visualized as a set of lifelines that interact with each other through messages. A lifeline represents an object or a class and method invocations result in arrows added to the diagram that go from one lifeline to another.

1.2 JavaWiz

JavaWiz is a tool that supports programming beginners by providing a visual debugger. That way, users can step through a program and observe dynamic changes at each execution step.

There is a web version [3] and a Visual Studio Code plugin [4] available. Both can be used by students, teachers and programming beginners in general for both educational and development purposes.

Depending on the user's needs, different dynamic visualizations are provided:

- A desk test
- A stack and heap visualization
- A console and input buffer visualization
- An array visualization
- A flowchart visualization (for more details see [5])
- A linked list and binary tree visualization (for more details see [6])

In this thesis, JavaWiz is extended by a sequence diagram visualization. The sequence diagrams are based on the UML standard [2]. The thesis focuses on the features of the additional component and how it was implemented.

1.3 Thesis structure

The structure of this thesis is as follows:

- Chapter 2 describes sequence diagrams as part of the UML standard and refers to dynamic visualization tools other than JavaWiz.
- Chapter 3 is dedicated to the tool JavaWiz and its architecture.
- Chapter 4 introduces the sequence diagram component that extends JavaWiz.
- Chapter 5 gives an overview of the actual implementation details.
- Chapter 6 summarizes the contribution of this thesis and gives my personal thoughts on the project.

2 Sequence Diagrams

Object orientation allows to define classes with fields and methods that are logically related. That way, programs are structured into basic components that are easier to comprehend [1]. Sequence diagrams are one way to show how object-oriented programs behave. They visualize the step-by-step object and class interactions that happen during the course of a program execution.

In this chapter, first sequence diagrams [2] as defined by the UML standard are presented. Then, existing tools other than JavaWiz that dynamically visualize sequence diagrams are reviewed.

2.1 The UML standard

The Unified Modeling Language (UML) [2] defines ways of visualizing the structure and functionality of programs. The software complexity is narrowed down by focusing on the basic elements and their interactions that can easily be understood [7]. Therefore, UML is not a programming but a modeling language. It serves as a means that is dedicated to humans instead of computers [8].

There are many different diagram types specified in the UML standard. One of them are sequence diagrams. They have the purpose of illustrating the interaction of objects and classes through method invocations. Instead of showing implementation details, sequence diagrams focus on simplifying the communication flow [9].

UML defines rules and guidelines on how sequence diagrams must be created. This standardized notation serves as the basis for the sequence diagram component in JavaWiz that is presented in Chapter 4.

Figure 2.1 shows parts of a sequence diagram as defined in the UML standard. An actor manually initiates the execution of a program. During this program, objects can be created which are represented as vertical lines with a label on top. Objects that do not exist anymore have a cross at the end of the line. Messages stand for method calls which are arrows that go from one object to another. However, messages do not have to connect two distinct objects, instead, method calls of the same object are possible too. When method calls return, return arrows are drawn. If needed, conditions can be added to the message labels within square brackets. A star (*) before square brackets indicates the start of an iteration.

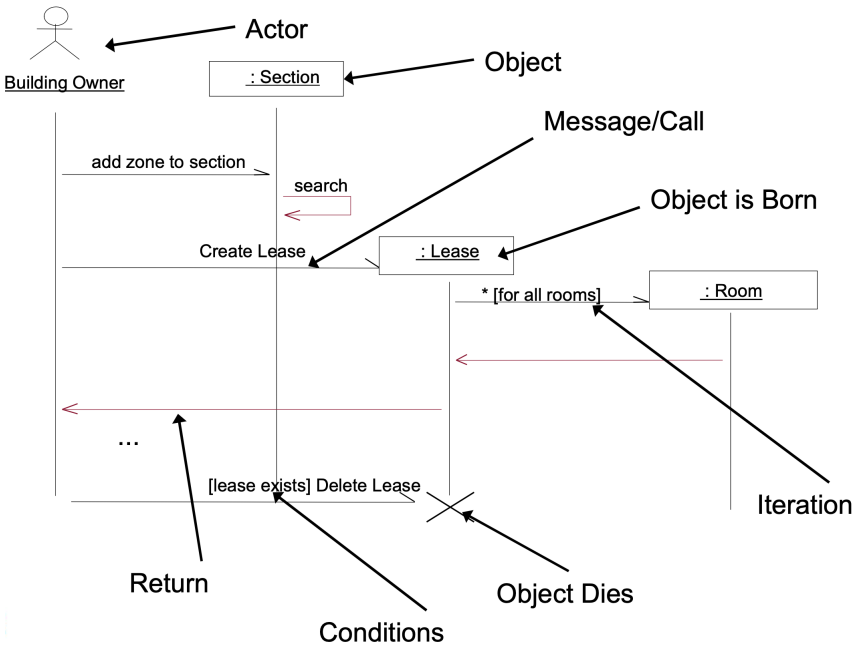


Figure 2.1: UML Sequence diagrams (taken from [10])

Figure 2.2 shows a sequence diagram that is defined based on a use case for drones taken from [11]. The use case describes the scenario of victim confirmation by a human operator. For this, the drone (UAV) raises an alert which is sent to the drone commander (Flight Management Service). The human must then inspect the video stream provided by the UAV Video Feed and confirm the victim. The sequence diagram shows the interaction between the contributing units. Method calls are drawn as solid lines while return arrows are drawn as dashed lines.

Compared to Figure 2.1, there are boxes for each call arrow that indicate the start and end of a method call. In UML, these boxes are called activations.

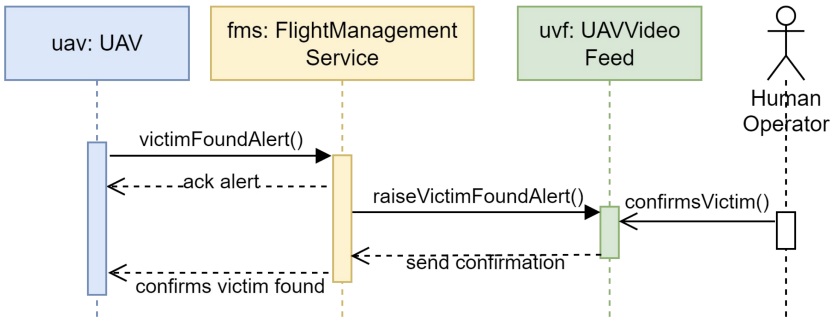


Figure 2.2: Sequence diagram use case example (taken from [11])

2.2 Existing visualization tools

JavaWiz is only one tool that provides visualizations for better code understanding. Various other approaches exist that contain functionalities similar to JavaWiz. Tools for dynamic visualizations with sequence diagrams are presented in the following.

2.2.1 SCENE

SCENE stands for *SCENario Environment* and describes a tool that visualizes the behavior of a system based on the use of scenario diagrams [12]. This visualization is based on the predecessor of the UML standard called *Object Modeling Technique (OMT) [13]*.

Scenario diagrams in OMT reflect the idea of sequence diagrams in UML. Both notations define objects as lifelines that are represented with vertical lines. Further, messages are represented through arrows that go from one object to another. The tool shows dynamic interaction diagrams by browsing through the source code of a program [12]. That way, it is possible to observe the interaction between objects in the program.

Figure 2.3 shows how a scenario diagram looks in SCENE. On the left side, the user that invokes the main method is shown first. The other objects are listed horizontally. Method calls are drawn as single-lined arrows. Return statements are shown as dashed arrows. The double-lined arrows define collapsed method calls that can be expanded to reveal further content.

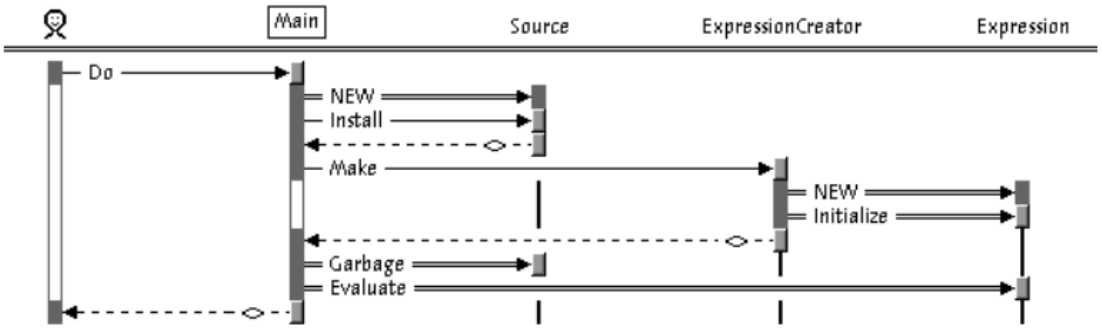


Figure 2.3: Scenario diagrams in SCENE (taken from [12])

2.2.2 JAVAVIS

JAVAVIS [14] is another tool that provides automatic program visualizations. Compared to SCENE, it is based on the UML standard. The dynamic changes in code behavior are shown in multiple object diagrams and one sequence diagram [14].

Figure 2.4 shows how a sequence diagram looks in JAVAVIS. The visualization reflects the state of the main method after executing two iterations of a for-loop. The application continuously extends a list as in each iteration a List object is created and appended. The blue boxes represent method executions. For each method call, there is an arrow and a box. Eventually, a return arrow is created when the method ends.

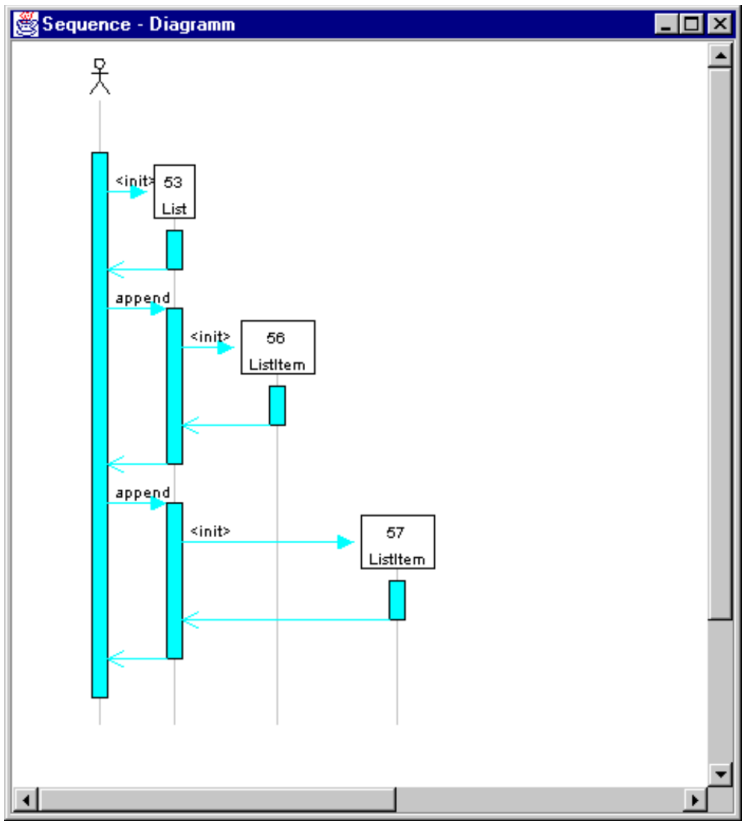


Figure 2.4: Sequence diagrams in JAVAVIS (taken from [14])

Overall, the JAVAVIS sequence diagram visualization together with the collapsing feature of SCENE is close to what the sequence diagram component in JavaWiz provides. The way JavaWiz combines different functionalities is presented in Chapter 4.

3 JavaWiz Visual Debugger Tool

JavaWiz is a tool developed by the Institute of System Software at the Johannes Kepler University in Linz. JavaWiz allows visualizing Java programs which is essentially dedicated to programming beginners. For a detailed description to the JavaWiz tool see [4].

This chapter gives a short introduction to JavaWiz. First, an overview is given. Subsequently, the execution flow of different commands is described based on the architecture of the tool. The centre of this architecture is the trace. Its structure is shown in Section 3.3.

3.1 Overview

JavaWiz provides visualizations while executing a Java program. Users can either download the VS Code Extension or simply use the Web version which runs in a browser and requires the backend to be started manually [4]. For the visualizations provided in this thesis, the Web version was used.

Figure 3.1 shows the initial view of JavaWiz after compilation. The toolbar at the top contains the control buttons. The left side contains buttons to step into, over and back in the execution. The icons in the right corner can be used to show or hide visualization components.

Below the toolbar, the code editor can be seen on the left side and the visualizations on the right side. When stepping, the current line in the code is highlighted in the code editor and the visualizations dynamically show the current program state.

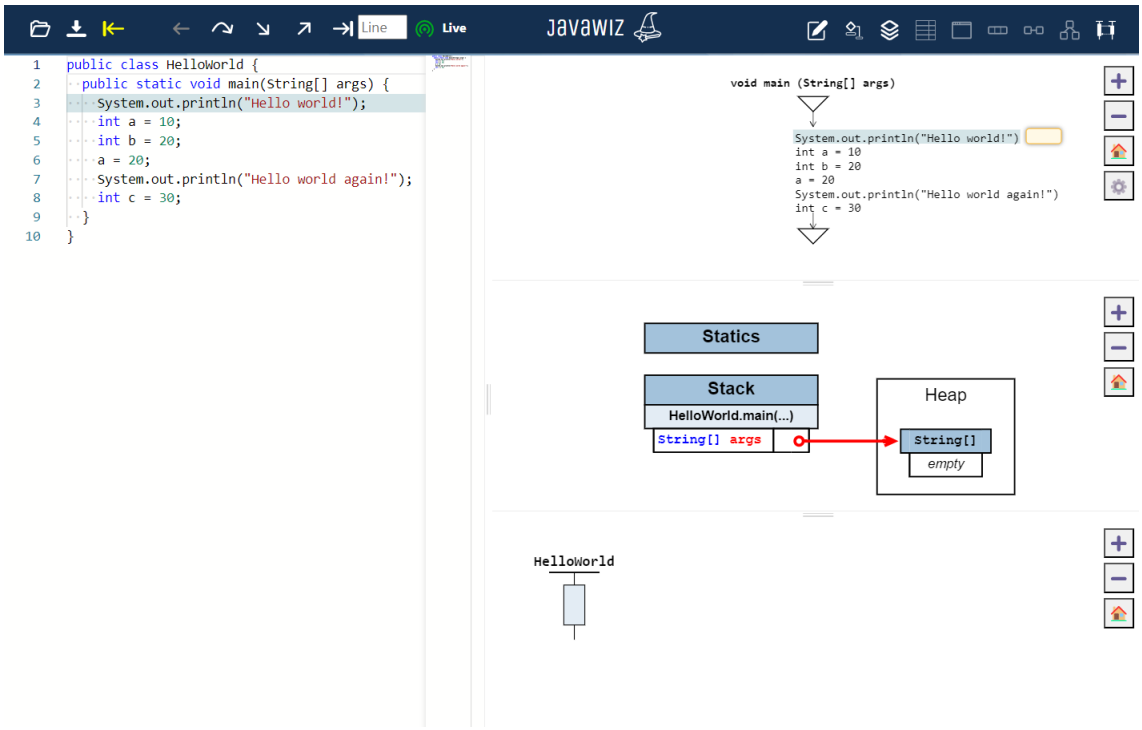


Figure 3.1: JavaWiz Screenshot

The following visualizations are available currently:

- **Desk Test**: This component serves as a support for checking the correctness of a program. This is done by providing a table that contains all lines of code together with the associated current values of variables [4]. Users can benefit from this tool as it helps to understand the basics of programming.
- **Console**: Statements or runtime exceptions are printed to the console. It contains a dedicated area for outputs and a text input field.
- **Input buffer visualization**: This visualization is closely linked to the console. Whenever a user has to provide some kind of input to the console, the input is shown in the input stream visualization. This allows to understand which elements are currently in the buffer and which element will be processed with the next input request.

- **Stack and heap**: This component shows the contents of both stack and heap as well as static variables that exist during the entire lifetime of the program. Local variables are stored on the stack, while objects that are created during program execution are located on the heap. This difference can easily be figured out by looking at the visualization.
- **Flowchart visualization**: Flowcharts are used to provide a 2-dimensional diagram of a program. The execution steps are made visible which offers clarity for decisions and the order of method calls in a program. In general, flowcharts consist of start and end points as well as decisions that determine the execution flow. Additionally, tasks represent actions and steps in a process [5].
- **List visualization**: This visualization shows the structure of linked lists. Normally, one must only save the first element of a list as all other successors are accessible through references in each node. However, the list visualization in JavaWiz aims to give an idea of the order of nodes for programming beginners [6].
- **Binary tree visualization**: Binary trees are similar to lists, however, multiple successor references inside a single node can exist. The hierarchies that this data structure internally creates are visualized in the binary tree visualization [6].
- **Array visualization**: The array visualization can be used to show the processing of arrays.

The sequence diagram component represents a new addition to the already existing visualization components. Sequence diagrams help to understand which objects are involved in a program execution and how they interact by method calls.

3.2 Architecture

The JavaWiz architecture consists of a backend and a frontend. The backend component runs the Java program and provides state information in each execution step which is used by the frontend component for the visualizations. Figure 3.2 shows the cooperation of the two architecture components in JavaWiz which will be explained in this upcoming section.

In the web version, the backend must be started manually. When the backend is running, the green start button on the left side of the toolbar starts the compilation. For this, the program is sent to the backend in a *compile* command. The backend compiles the code and starts a debugger after successful compilation. In JavaWiz, a Java Virtual Machine (JVM) contains the program code. The Java Debug Interface (JDI) serves as the debugger interface [6].

When the debugger is running, step requests can be accepted. A step sends a *step* command to the backend. The debugger then distinguishes between step into, step over and step back. This is only possible as the backend always sends the current state of the program back to the frontend. The so-called trace that can be seen as part of the frontend in Figure 3.2 then contains all previous states up until the most current one in chronological order.

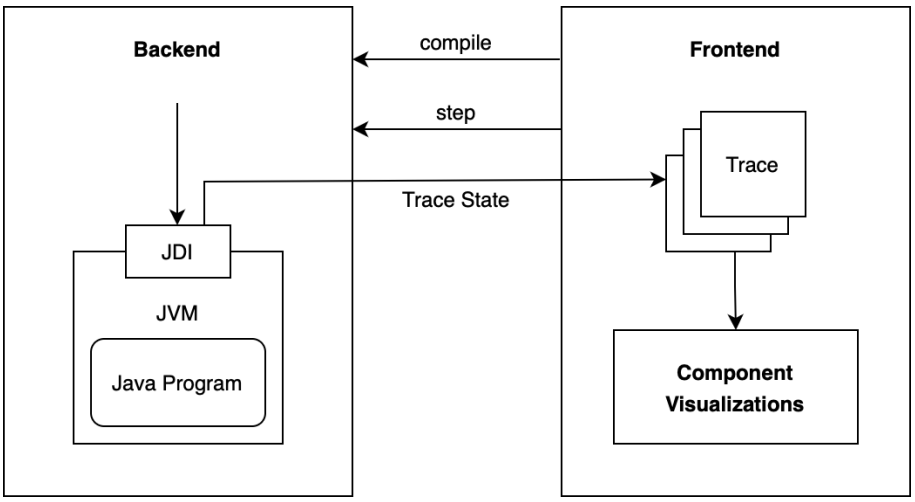


Figure 3.2: JavaWiz Architecture

3.3 Trace

So far, it has been mentioned that the trace describes a collection of states that can be associated to different step commands. The backend sends a list of trace states back.

Figure 3.3 shows how a trace state is structured. The different parts are as follows:

- **sourceFileUri**: This string contains the name of the source file that is currently being processed.
- **line**: This field stores the current line number in the source file.
- **stack**: The stack array collects all method calls that are currently active. Each array element is defined as a `StackFrame`, which contains information such as the associated class, the local variables and the method signature.
- **heap**: The heap array collects the active objects that are created during program execution. Each array element is defined as a `HeapItem`, which stores the `id`, `kind` and `type`. Based on `kind`, additional information is provided. For instance, a `HeapArray` kind stores its elements, a `HeapObject` kind contains its fields and a `HeapString` kind stores its associated string.
- **loadedClasses**: This array contains all classes that have been loaded. For each `LoadedClass`, the class name and the static fields are stored.
- **output, error and input**: If an output or error message was printed as well as if an input was entered to the console, the associated trace state must contain this information. Otherwise, empty strings are stored.
- **inputBufferInfo**: This field is only necessary for the input buffer visualization. It contains information about the current position in the input buffer and which elements are yet to be processed.

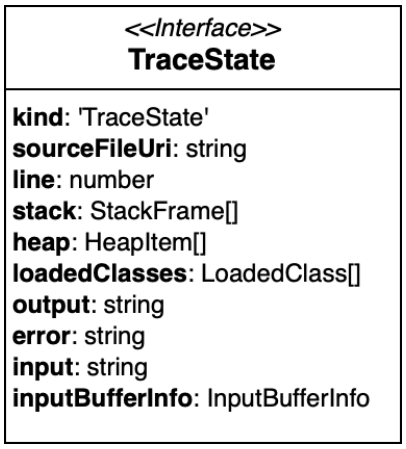


Figure 3.3: TraceState Interface

As explained in Section 3.2, each step or input request results in a new trace state that is provided by the backend to the frontend. The entirety of trace states form the trace that is used by the JavaWiz components to retrieve data for the visualization.

The data sampling process for the sequence diagram component based on the analysis of the trace is explained in Section 5.3.

4 Sequence Diagrams in JavaWiz

This chapter shows how the Sequence Diagram Component in JavaWiz visualizes method call behavior of Java programs. Further, the different features provided for the user, like collapsing and expanding, will be shown.

4.1 Simple method call

The basic functionality is explained by introducing a Java program that describes a simple method call between two classes.

Listing 4.1 shows two classes, one of which represents the main application that creates a new object instance of class `A` and calls the method `foo`.

```
1  public class Main {
2    public static void main(String[] args) {
3      A a = new A();
4      int z = a.foo();
5    }
6  }
7
8  class A {
9    int foo() {
10     return 2;
11   }
12 }
```

Listing 4.1: Simple code example

Every program starts at the beginning of the `main` method. The initial state of each sequence diagram looks like in Figure 4.1. The visualization reflects the uniform way of drawing sequence diagrams which consist of **lifelines, activations** and **method calls**. Lifelines are created for each new class or object that is present in an associated method invocation. As shown in Figure 4.1, the horizontal line provides space for the label and is always positioned at the top of the vertical line, which indicates the lifetime of the corresponding class.

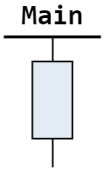


Figure 4.1: Main starting point

While lifelines identify classes or objects, activations show the execution of methods. The box characterizes the method that is active until it returns or the program ends. When executing the code in JavaWiz, one can decide to step into the method `a.foo` and see the dynamic changes in the visualization.

The steps of calling method `foo` and its return are shown in Figures 4.2 and 4.3. A second lifeline with the label `a:A` is introduced. The first letter of `Main` is written in uppercase, while `a:A` refers to the variable itself and adds the name of the corresponding class after the colon. The syntax of the labels is chosen in this way to show the difference between classes with static methods and objects with object methods.

The method call is represented as a solid arrow that goes from one activation to the start of another. When the method ends, one returns back to the previous activation which is indicated by a dashed arrow.

As more elements are added to the sequence diagram, the visualization dynamically adapts already existing components. This can be seen as the heights of the lifelines and activations are extended depending on the latest introduced units.

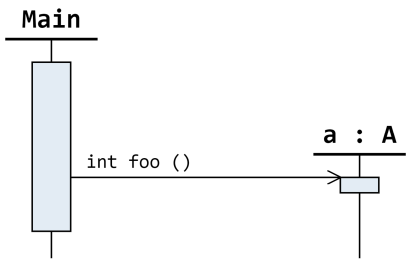


Figure 4.2: Method is called

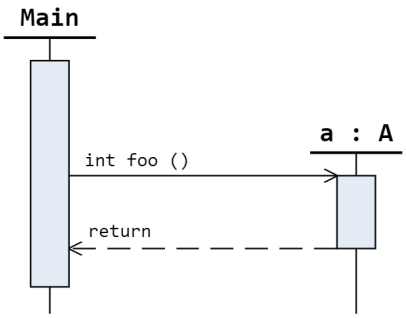


Figure 4.3: Method returns

Figure 4.4: Basic dynamic visualization changes

When the scope of an object ends, the lifelines must include an indication for no longer being active. This is illustrated by a cross that is placed at the bottom of the vertical line.

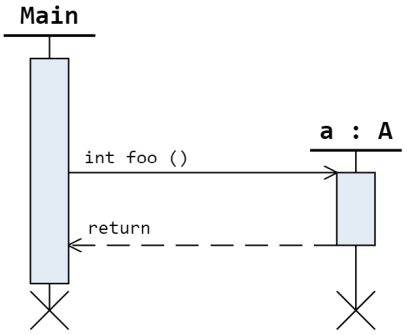


Figure 4.5: Program ended

4.2 Self calls

In the previous example, only straight arrows between different lifelines exist. A method call of the same object or a static method of the same class is represented by a so-called self call. To show this, Listing 4.2 introduces a new code example.

```
1  public class Main {
2    public static void main(String[] args) {
3      foo();
4    }
5    static void foo() {
6      int z = bar();
7    }
8    static int bar() {
9      return 1;
10   }
11 }
```

Listing 4.2: Self call code example

So far, method calls are drawn as straight arrows, however, having only one lifeline makes this impossible. Figure 4.6 shows how this issue is resolved in the visualization. In addition to the regular call and return arrows, a new arrow kind with a different shape is introduced. Self call arrows build a circle starting from the current activation and return to the same lifeline with a new activation. These activations must be stacked in a way to represent the method executions in the correct order.

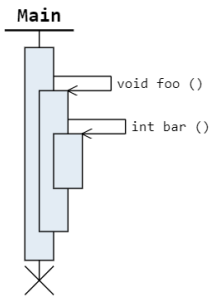


Figure 4.6: Multiple self calls

4.3 Collapsing, hiding and expanding activations

All prior figures show expanded activations that are colored in light blue. Collapsing an activation is done by double-clicking on the corresponding box.

Listing 4.3 is an example program with a main method and two objects of classes `A` and `B`. It will be used to demonstrate stepwise execution and the corresponding visualization states.

```
1  public class Main {
2    public static void main(String[] args) {
3      A a = new A();
4      a.foo();
5      a.foo();
6    }
7  }
8
9  class A {
10   B b = new B();
11   void foo() {
12     b.bar();
13   }
14 }
15
16 class B {
17   void bar() {
18     test();
19   }
20   int test() {
21     return 2;
22   }
23 }
```

Listing 4.3: Code example for collapsing activations

Figure 4.7 shows the visualization when stepping into the first two method calls. Lifelines are created for the used objects and activations represent the start of method execution.

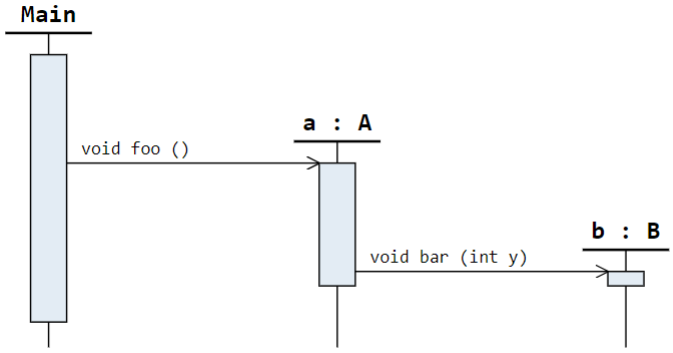


Figure 4.7: Stepping into first a.foo()

The method returns are shown by dashed arrows to indicate the end of execution. This can be seen in Figure 4.8.

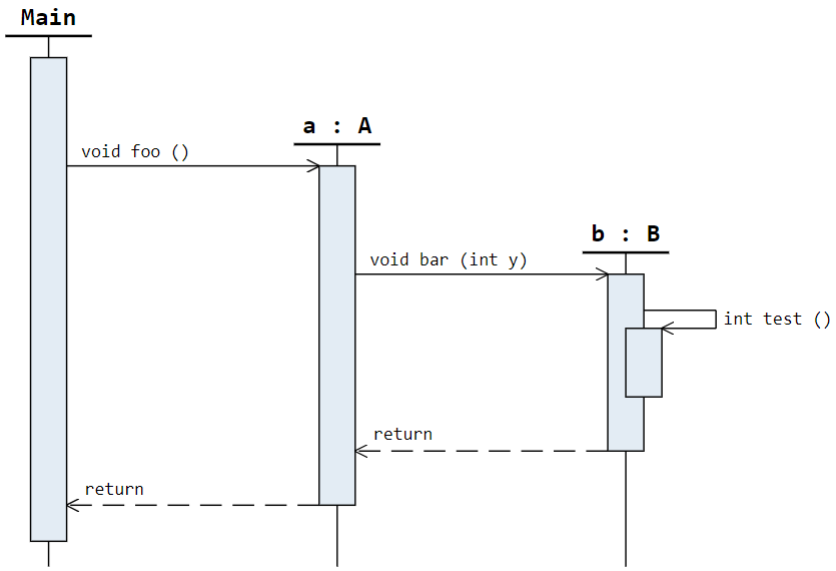


Figure 4.8: a.foo() returns

The entire sequence diagram that is created for the program is illustrated in Figure 4.9. As explained in Section 4.2, self calls are drawn inside method `bar` to visualize a method invocation of the same object.

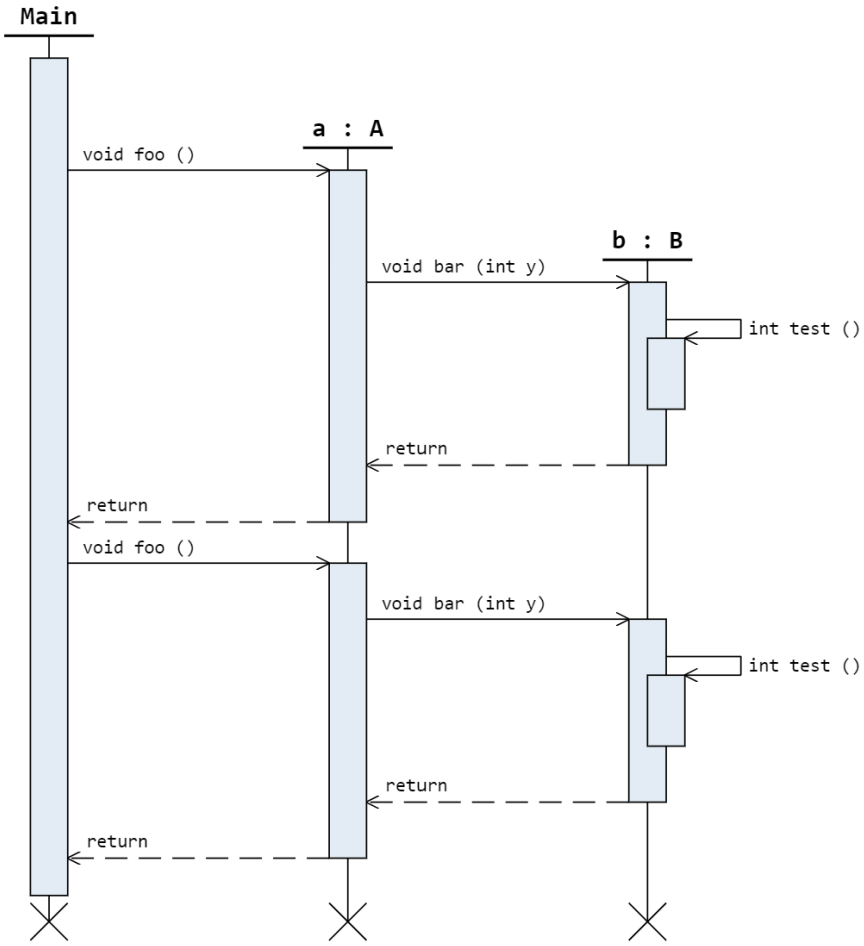


Figure 4.9: Visualization of both a.foo() calls

In Figure 4.10 the first activation for object `b` has been double-clicked and therefore collapsed. Collapsed activations are specified with a shade of blue that is slightly darker than what is used for expanded ones. Furthermore, there are three dots that indicate that the method contents are hidden.

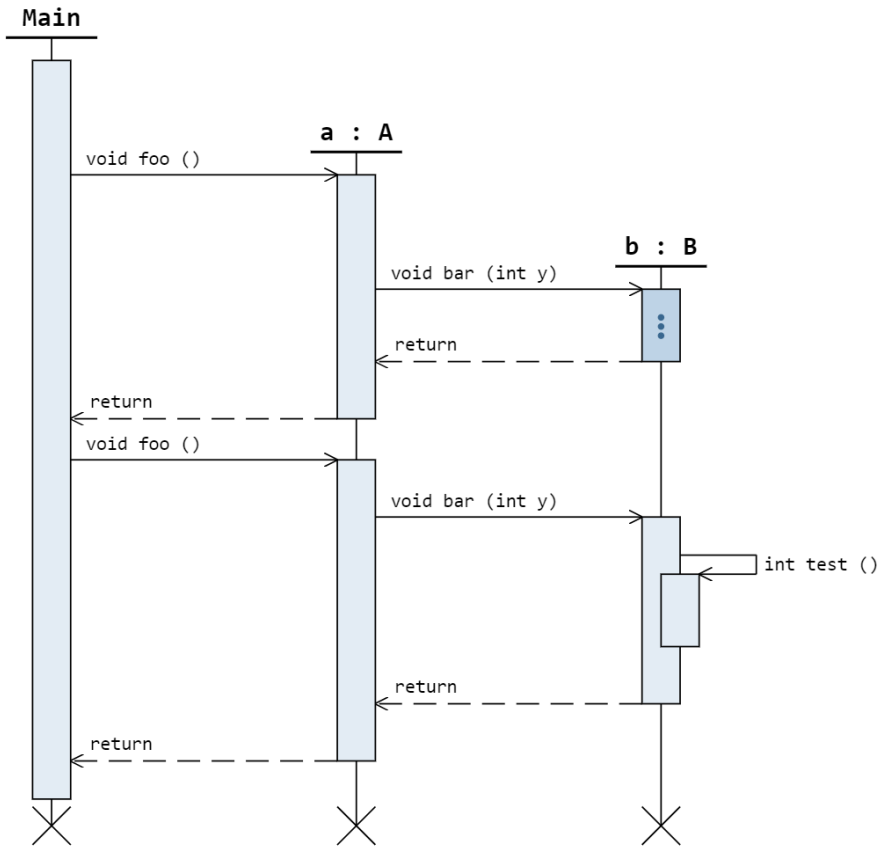


Figure 4.10: Collapsing first b.bar()

Figure 4.11 shows the visualization after the first activation of object `a` has been collapsed. Hidden activations are not displayed which is noticeable as the activation for the first `bar()` call is not shown in the figure. The same applies for method call arrows that link to hidden activations.

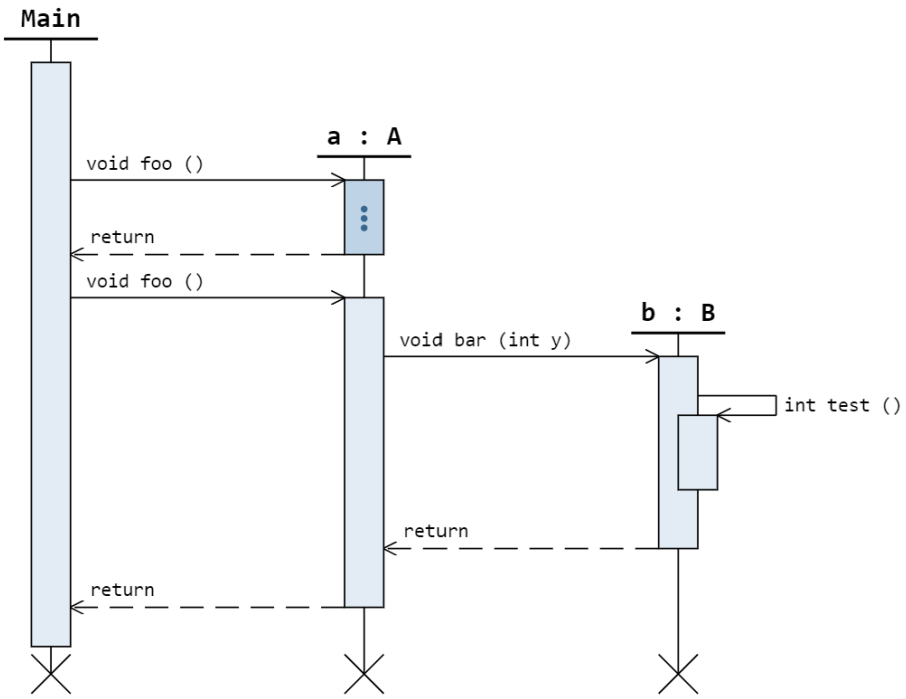


Figure 4.11: Collapsing first a.foo()

Collapsed activations shrink in size which has the same effect on the height of all lifelines. Additionally, the activations that contain the collapsed method execution adjust their height accordingly. All other method calls that occur after the collapsed one are shifted upwards in order to fill the space of any hidden elements. This dynamic change shows that the y-coordinates of arrows and activations can change depending on the height of any surrounding activations.

Another important feature is that when expanding `a.foo` again, the previously collapsed `b.bar` in Figure 4.10 must still be collapsed. This indicates that expanding always results in the state before collapsing.

So far, the visualization examples resulted from stepping into each method call. However, the Sequence Diagram Component allows to *step over* methods which results in collapsing the activation and hiding the contents inside the scope.

Figure 4.12 shows the sequence diagram after stepping over both `a.foo` calls. Although three lifelines are created throughout the program, stepping over the method calls leads to only two of them being visible. As `b:B` is only created within the execution of `a.foo`, the third lifeline is hidden together with any activations and arrows connecting to it. A hidden lifeline located between two visible ones would lead to more changes. The remaining lifelines created after the hidden one would be shifted upwards and to the left to fill the empty space.

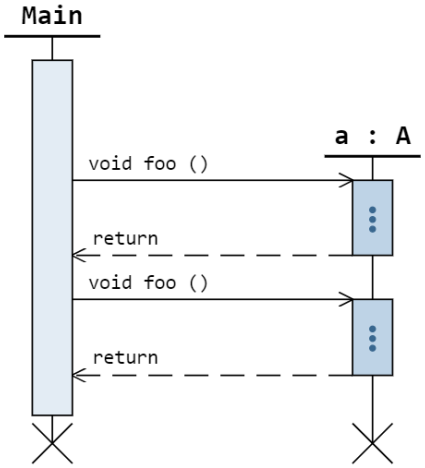


Figure 4.12: Step over collapse

4.4 Example program

Listing 4.4 shows a more complex example to point out the use of previously explained features. The program consists of a class system for shapes. The base class `Shape` declares the abstract methods `getLeft`, `getTop`, `getWidth`, `getHeight` and `draw`. The subclasses `Rect`, `Circle` and `Group` implement those methods. The class `Primitive` is a common super class for `Rect` and `Circle`. In the main class `Shapes` a shape of a snowman is created and the methods `getLeft`, `getWidth` and `draw` are called.

The method `draw` in class `Group` initiates the drawing process for each sub-element of the snowman. However, instead of actually drawing the snowman, strings containing information about the shape of an element are printed.

```
1  public class Shapes {
2    private static final int DIST = 10;
3
4    public static void main(String[] args) {
5      Rect armL = new Rect(60, 210, 40, 30);
6      Rect armR = new Rect(200, 210, 40, 30);
7      Circle body = new Circle(150, 250, 60);
8      Circle head = new Circle(150, 160, 30);
9      Group snowMan = new Group(armL, armR, head, body);
10
11     Out.println(snowMan.getLeft() + snowMan.getWidth());
12     snowMan.draw();
13   }
14 }
15
16 class Shape {
17   public int getLeft() {return 0;}
18   public int getTop() {return 0;}
19   public int getWidth() {return 0;}
20   public int getHeight() {return 0;}
21   public void draw(){}
22 }
23
24 class Primitive {
```

```
25    private final int x;
26    private final int y;
27
28    public Primitive(int x, int y) {
29      this.x = x;
30      this.y = y;
31    }
32    public int getX() {return x;}
33    public int getY() {return y;}
34  }
35
36  class Rect extends Primitive {
37    private final int width;
38    private final int height;
39
40    public Rect(int x, int y, int width, int height) {
41      super(x, y);
42      this.width = width;
43      this.height = height;
44    }
45
46    @Override
47    public int getLeft() {return getX();}
48    @Override
49    public int getTop() {return getY();}
50    ...
51  }
52
53  class Circle extends Primitive {
54    private final int radius;
55
56    public Circle(int x, int y, int radius) {
57      super(x, y);
58      this.radius = radius;
59    }
60
61    public int getRadius() {return radius;}
62    @Override
63    public int getLeft() {return getX() - radius;}
```

```
64    @Override
65    public int getTop() {return getY() - radius;}
66    @Override
67    public int getWidth() {return radius * 2;}
68    @Override
69    public int getHeight() {return radius * 2;}
70    ...
71  }
72
73  class Group extends Shape {
74    private final Shape[] subShapes;
75
76    public Group(Shape... subShapes) {
77      this.subShapes = subShapes;
78    }
79
80    public Shape[] getSubshapes() {return subShapes.clone();}
81    @Override
82    public int getLeft() {
83      int min = Integer.MAX_VALUE;
84      for (Shape s : subShapes) {
85        if (s.getLeft() < min) {
86          min = s.getLeft();
87        }
88      }
89      return min;
90    }
91    ...
92    @Override
93    public void draw() {
94      for (Shape s : subShapes) {
95        s.draw();
96      }
97    }
98  }
```

Listing 4.4: Shape classes

Line 11 in Listing 4.4 contains two separate method calls. The method invocation that occurs first is `snowman.getLeft`. Figure 4.13 shows that during execution method `getLeft` is called for each existing object. The calls of `getLeft` of the last three objects in the figure are collapsed to save space.

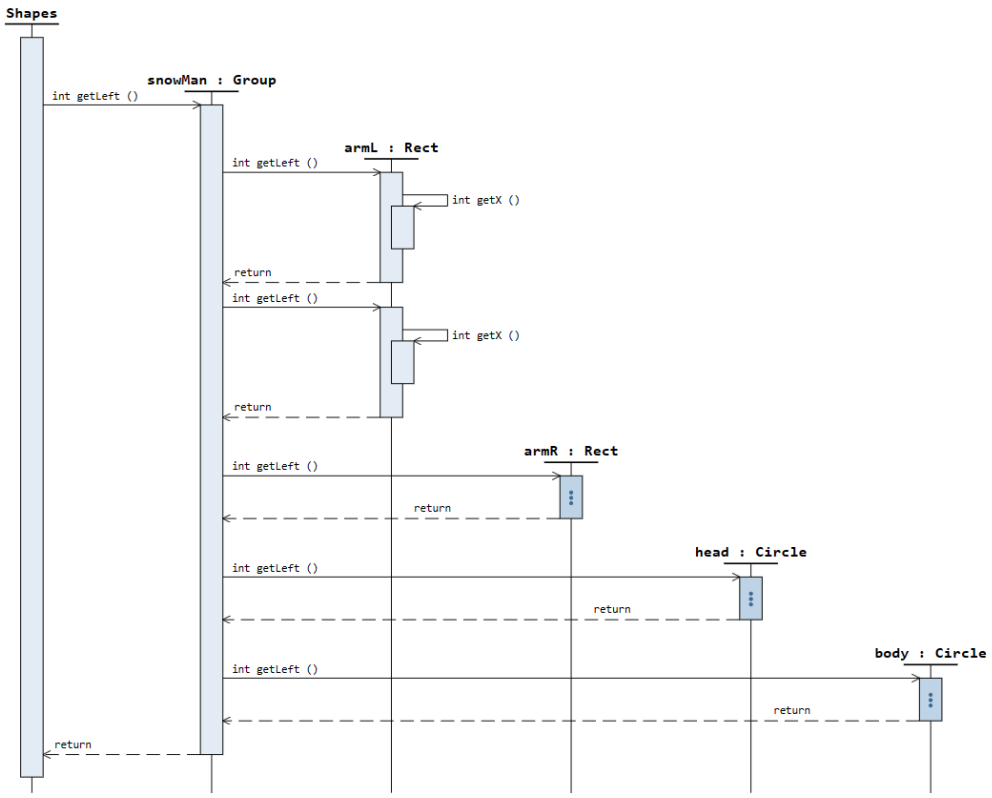


Figure 4.13: Visualization after snowman.getLeft()

In Figure 4.14 the main method has been collapsed by double-clicking on the corresponding box. After expanding the visualization would go back to its state in Figure 4.13.

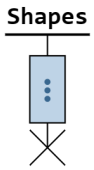


Figure 4.14: Visualization after collapsing the main method

The visualization in Figure 4.15 shows that the contents of `snowman.getLeft` have been collapsed to reduce the size of the sequence diagram. The next method call is `snowman.getWidth`. During execution the methods `getLeft` and `getWidth` alternate for each existing object. A self call is invoked before the method returns. The method `draw` is stepped over as we know that each object only prints a string.

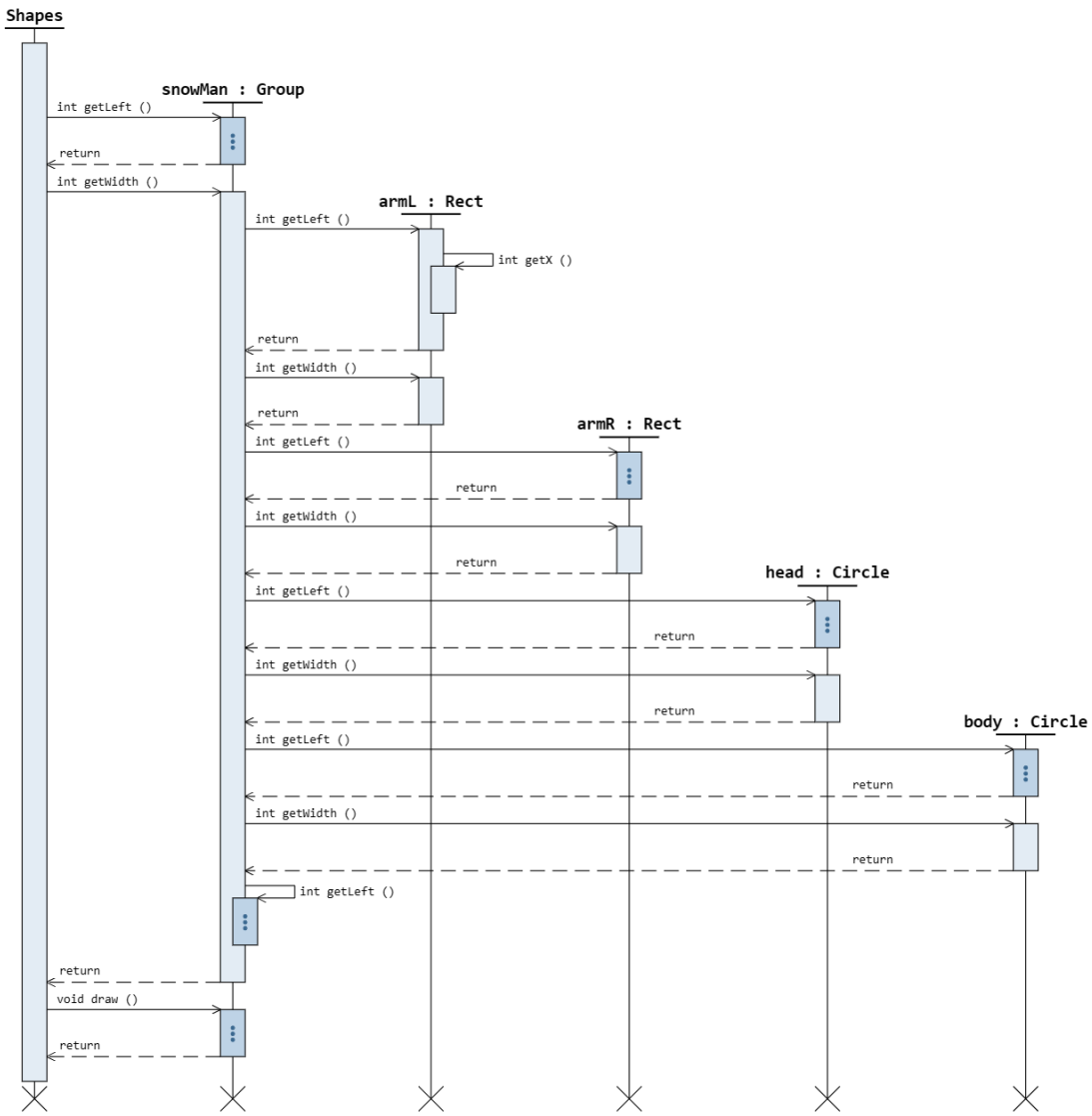


Figure 4.15: Visualization after program ended

5 Implementation

In this chapter, an introduction to the implementation of the sequence diagram component is given. Besides standard HTML, CSS and JavaScript, the sequence diagram component uses the Vue.js framework following the same template as all other JavaWiz components. Vue.js works component-based and relies on reusable instances for developing new user interfaces [15].

The code is written in TypeScript integrating the D3.js library to support visualizations with smooth animations. D3 stands for data-driven documents as the document object model (DOM) is modified [16]. This chapter shows how the data is sampled and linked to the component to trigger rendering.

5.1 Component structure

Figure 5.1 shows the interaction between the involved components. The backend compiles the code and runs the debugger that provides trace states for each *step* request. The trace contains all trace states up until the most current one. Whenever new trace states occur, the trace as well as the `SequenceDiagramHistory` are updated. The `SequenceDiagramHistory` keeps track of three different collections for lifelines, arrows and so-called boxes which represent activations. These collections are updated or extended by analyzing the stack and heap of each new trace state to filter out indications for the creation or modification of data.

The sampled data is then linked to the sequence diagram component. When stepping through a program, new trace states are continuously added to the trace which means that the `SequenceDiagramHistory` is updated regularly. As the collections for lifelines, arrows and boxes are adapted and extended, the visualization is re-rendered.

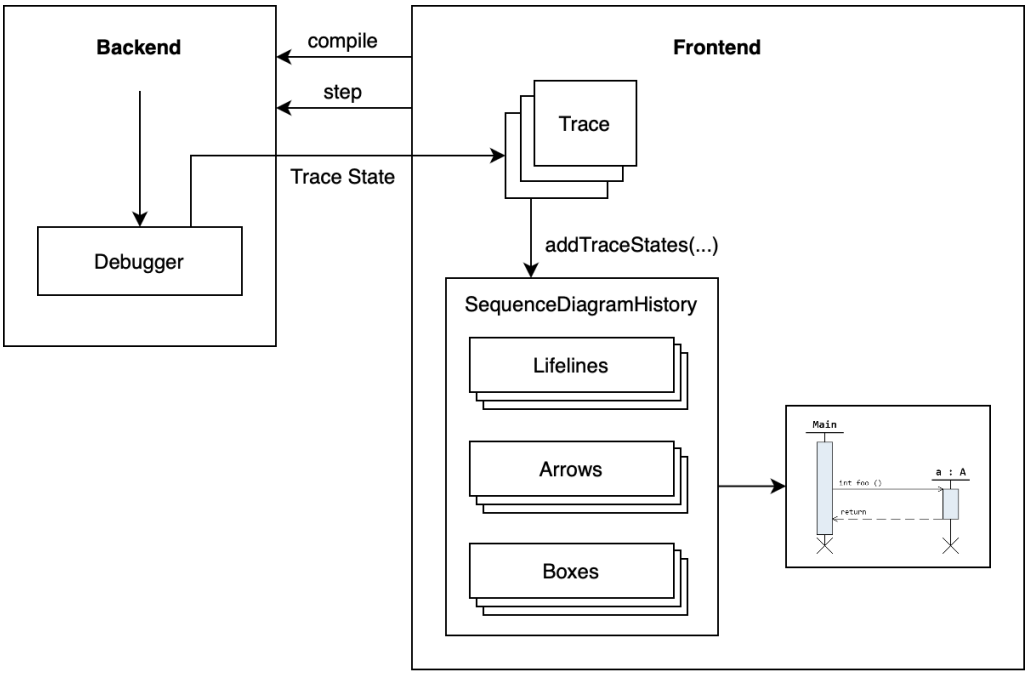


Figure 5.1: SequenceDiagramHistory as part of the architecture

Figure 5.2 shows a class diagram that explains the relationship between the trace and the components relevant for the sequence diagram visualization. As explained in Section 3.3, the trace consists of multiple trace states. There exists one `SequenceDiagramHistory` inside the trace. Internally, there are three separate collections containing multiple lifelines, arrows and boxes. The structures of the collection data types `LifeLine`, `Arrow` and `Box` are shown in Section 5.3.

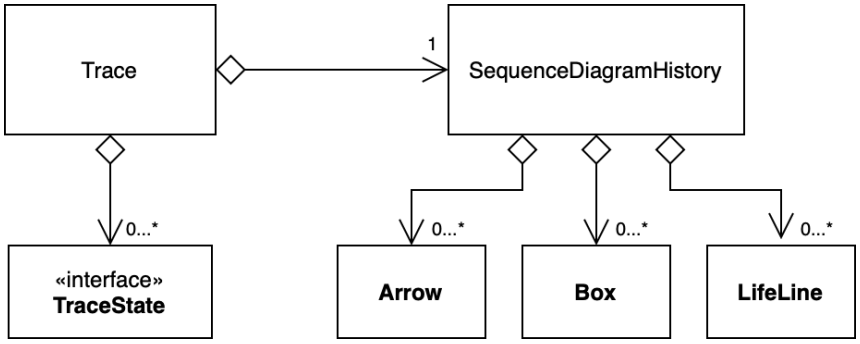


Figure 5.2: Class diagram

For the visualization, the HTML-based template syntax provided by Vue.js is used to create an initial Scalable Vector Graphic (SVG) that contains three separate groups for lifelines, arrows and activations.

Listing 5.1 shows what is specified in the template and compiled to JavaScript code. Vue.js can easily figure out the smallest number of required changes in the DOM as reactivity makes the interconnection between data and user interface possible [17].

```
<svg id="sequence-diagram-svg"
    width="120%"
    height="120%"
    viewBox="0 0 600 10000"
    preserveAspectRatio="xMidYMin slice">
    <g id="sequence-diagram">
        <g id="life-lines" />
        <g id="boxes" />
        <g id="arrows" />
    </g>
</svg>
```

Listing 5.1: Main SVG

Figure 5.3 shows the main Vue.js component for the sequence diagrams. The props contain the data provided by the `SequenceDiagramHistory`. The sequence diagram component links the gathered information about lifelines, arrows and activations to the corresponding groups in the SVG. Event indices indicate the order of events in the data sampling process with `eventIdx` in the props being the most current one. The `stateIdx` represents the current state in the program execution.

The `data` function provides the initial state of the Vue.js component including properties that are referenced and modified within the template. These properties include the SVG and data collections for `hiddenIntervals` and `activeEventIndices` which are progressively updated. The purpose and underlying algorithm are elaborated in Sections 5.2.1 and 5.4.

The `redraw` method is invoked whenever data changes and the DOM requires to be re-rendered. D3.js is incorporated to ease the visualization process.

TheSequenceDiagram.vue

```
name: 'TheSequenceDiagram',
components: {...},
props: {
        lifeLines: {...},
        arrows: {...},
        boxes: {...},
        eventIdx: {...},
        stateIdx: {...},
        ...
},
data: function () {
        return {
                svg: select('#sequence-diagram-svg'),
                hiddenIntervals: [],
                activeEventIndices: [],
                ...
        }
},
...
methods: {
        redraw() {...},
        ...
}
```

Figure 5.3: The Sequence Diagram Component

5.2 Layouting

Figure 5.4 shows how the SVG is structured for the visualization. The structure is based on two axes. While lifelines are added along the horizontal line, method calls happen at different event indices that cause the visualization to grow downwards. Both lifeline and event indices are maintained through distinct counters that are incremented with new events.

The sequence diagram that is shown in Figure 5.4 is taken as an example to further explain the meaning of the indices. Each lifeline has a unique lifeline index that determines the horizontal position. The distance between two lifelines in the visualization is constant and therefore always the same. Longer labels that overlap with other elements are abbreviated with three dots.

Event indices represent the order of events with events being

- the creation of lifelines for objects or classes
- the end of lifelines
- method calls represented by solid arrows
- the return of method calls represented by dashed arrows
- the creation of activations
- the end of activations

In comparison to the lifeline indices, the distances between two event indices differ in the visualization. This is the case as the height and distances between elements in the visualization are calculated dynamically. Based on the event index that belongs to a certain position, the y-coordinates for lifelines, arrows and boxes are determined with different equations that add or multiply constants to the event index. This approach allows to define distances without having to stick to constant arrangements. For example in Figure 5.4, the distance between the start of a lifeline and the start of the first activation is smaller than other distances in the visualization. This creates a more compact solution that uses space more effectively.

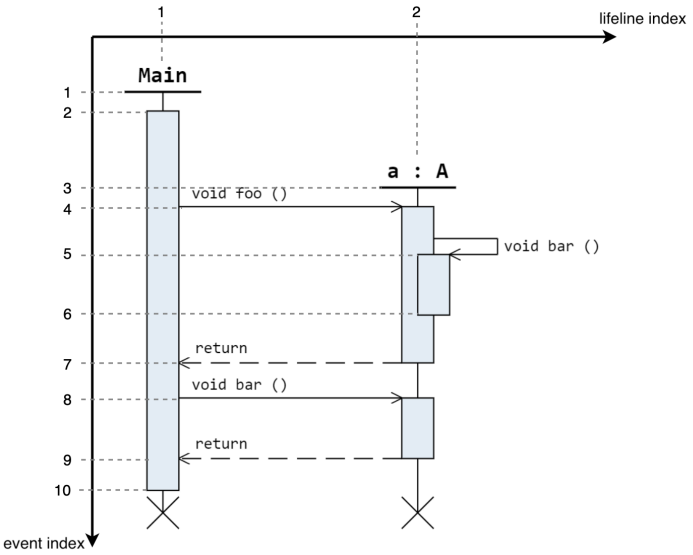


Figure 5.4: SVG structure based on two axes

5.2.1 Hidden intervals and active event indices

Collapsed activations are managed in order to adapt the positions and heights of any visible elements. The global data called `hiddenIntervals` contains all collapsed method calls and is updated whenever a box is double-clicked. The indices `activeEventIndices` represent all event indices that are visible. Whenever the `hiddenIntervals` change, the `activeEventIndices` must be updated to reflect the current state.

Taking the previous sequence diagram in Figure 5.4 as an example, it has already been explained that each activation has a start and end associated, each arrow is created at a certain event and each lifeline has a starting event index. The activation for `a.foo` starts at event index 4 and ends at event index 7. No box is collapsed, therefore, there are no `hiddenIntervals`. The `activeEventIndices` go continuously from 1 to 10.

In Figure 5.5 the activation for `a.foo` is collapsed. The activation is pushed to the `hiddenIntervals` as the contents are no longer visible. This indicates that the event indices 5 and 6 are not active anymore. They are deleted from the `activeEventIndices`. The event indices 1-4 and 7-9 are the ones that are still active.

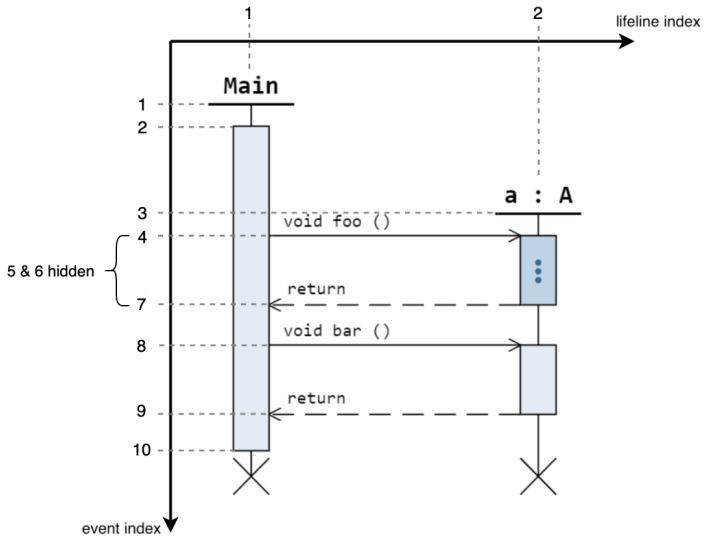


Figure 5.5: Active event indices after collapsing

Based on the position of an event index in the collection of active indices, the dynamic adjustment of the sequence diagram is made possible. By using transition animations provided by D3.js, the visualization changes smoothly.

5.3 Data sampling in SequenceDiagramHistory.ts

The `SequenceDiagramHistory` that is created by the `trace` receives information about new trace states as the method `this.sequenceDiagramHistory.addTraceStates` is invoked. `SequenceDiagramHistory.ts` internally keeps track of collections containing lifelines, arrows and boxes. In the following, the data sampling process based on the analysis of the trace at each step event is explained in detail.

5.3.1 Creation of lifelines

The stack must be examined to create lifelines for classes or objects. As mentioned in Section 3.3, the stack of a trace state consists of multiple stack frames.

Figure 5.6 shows the structure of one stack frame. For each new trace state, one must check the `this` of the first stack frame. Then, the `name` and `heapId` of an object is searched in the `localVariables` as well as the `heap` of each stack element. `this` being null defines a class, otherwise, the new lifeline must define an object.

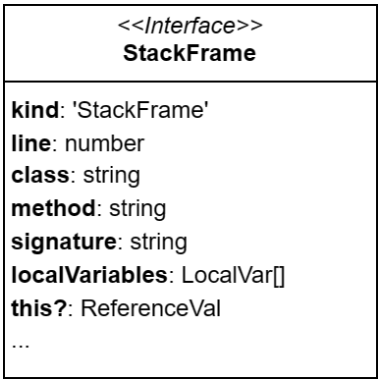


Figure 5.6: Structure of one stack frame

Figure 5.7 shows the information that must be gathered to create a new lifeline. The properties of a lifeline are explained in the following:

- **index**: Each new lifeline receives a unique identification.
- **start**: This field refers to the starting event at which the lifeline is created.
- **end**: A new lifeline is always active at creation. The ending event is set when the object does not exist any longer.
- **label and className**: The class name found in the stack is stored. Object classes require both the field name and the class name as part of the label. Static classes simply store the class name twice.
- **heapId**: Only objects have an associated heap id that must be maintained.
- **programEnd**: When the program ends, lifelines that are still active must terminate. There exists a dedicated field which indicates that all lifelines must end and the crosses have to be drawn.
- **isHidden and wasHidden**: When lifelines are created at a time that is inside the scope of a collapsed box or no methods of the lifeline are invoked, then `isHidden` becomes true. `wasHidden` stores the previous state of the lifeline.

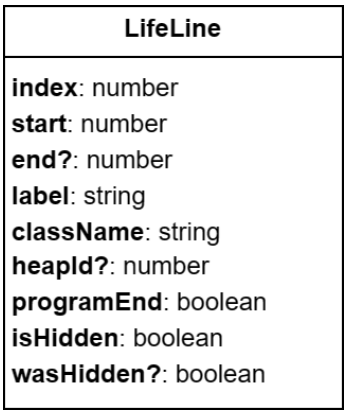


Figure 5.7: Lifeline data type

5.3.2 Creation of arrows and boxes

After creating and possibly terminating lifelines, the next step requires the introduction of arrows and boxes for new method calls. Arrows are created by comparing the stack of each trace state to the previous one. A new `StackFrame` indicates a method call and thus an arrow has to be drawn. If the stack does not contain this `StackFrame` anymore and the corresponding method call is not a constructor call, then a return arrow has to be drawn.

Figure 5.8 shows the data type that is used for arrows:

- **from and to**: An arrow goes from one box to another. The lifelines associated to them must be stored.
- **fromBoxIndex and toBoxIndex**: The unique identifications of the two affected boxes are saved.
- **kind and direction**: There are three arrow categories that are treated differently during data sampling. *Call* arrows are created in combination with boxes. *Return* arrows are assigned to already existing boxes. Running the main method creates a *CallMain* arrow that is never visibly drawn and leads to a main method execution that is active until the program ends. Basic arrows can point to the left or right, however, self call arrows have no particular direction.
- **label**: Call arrows are labeled with the method signature. For return arrows the label is simply *return*.
- **time**: This field defines the event index at which the arrow is created.
- **methodCallId**: Each method call receives a unique id that at some point has one call and one return arrow as well as one box associated.
- **fromDepth and toDepth**: As boxes can be stacked on top of each other, the depth must be considered for correct visualizations.
- **isHidden and wasHidden**: Arrows are hidden when they are created inside the scope of collapsed boxes. `wasHidden` stores the previous state of the arrow.

- **changed**: This flag is only used for zooming purposes in order to keep track of already processed arrows.

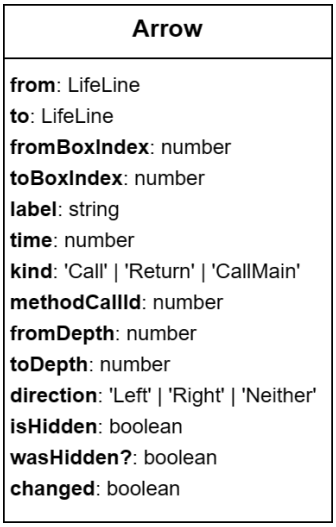


Figure 5.8: Arrow data type

As explained, call arrows are created for each new method call together with an associated box that marks the start of a new method execution. Boxes represent activations and are structured as shown in Figure 5.9. The information that must be gathered is as follows:

- **index**: Each box has a unique index for identification.
- **lifeLineIndex**: The unique id of the associated lifeline is maintained.
- **start**: The starting event at which the box is created is stored.
- **end**: When a method returns, the ending event has to be set accordingly.
- **callArrow, returnArrow and methodCallId**: The method call id, call and return arrow must be stored to associate arrows and boxes to one another.
- **depth**: As activations can be stacked, the depth must be preserved.
- **currState and prevState**: Boxes can be *expanded*, *collapsed* or *hidden*. When boxes change their current state, `prevState` defines the state before modification.
- **isDrawn and wasDrawn**: As collapsed boxes can either be visible or not, flags must be kept to have access to this information during visualization.

- **stepOver**: This field is only needed to determine whether a box must be stepped over or not.

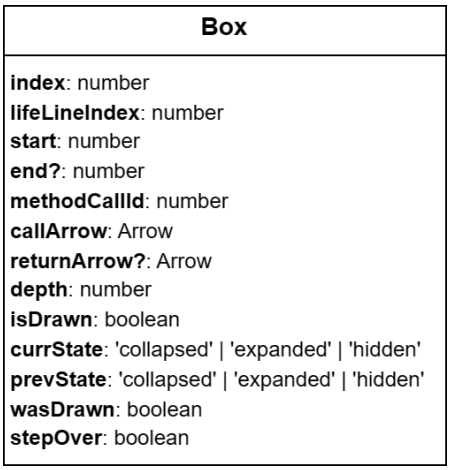


Figure 5.9: Box data type

5.4 The redraw method

The sequence diagram is rendered by invoking the `redraw` method. This is done automatically whenever the associated props change.

This is straight-forward, however, methods stepped over need special treatment. Figure 5.10 shows a sequence diagram where none of the methods are stepped over and thus no activations are collapsed. As shown in the figure, method `foo` is started with event index 4 and ends with event index 7. In-between event indices 5 and 6 are the start and end of method call `bar`. An internal map is used that assigns event indices to their associated state indices. This is shown in Figure 5.11.

Now, we assume that method `a.foo` has been stepped over. Figure 5.12 shows that the method is collapsed now. For recognizing that a method invocation has been stepped over and the activation has to be collapsed, the mapping from event indices to state indices is used. As can be seen in Figure 5.13, now there is a gap in the state indices. This is the indication that the method `foo` has been stepped over and the collapsed activation contains method calls not shown.

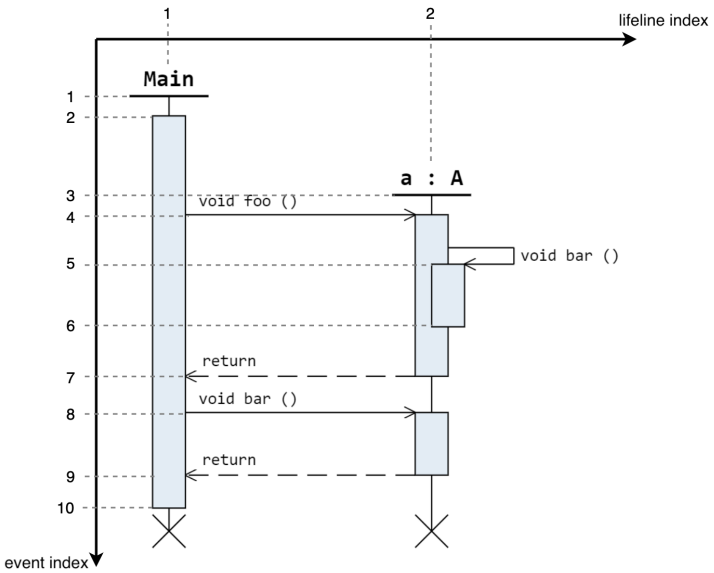


Figure 5.10: Sequence diagram example

mapping for activation a.foo()
eventIdx: 4 --> stateIdx: 3
eventIdx: 5 --> stateIdx: 4
eventIdx: 6 --> stateIdx: 5
eventIdx: 7 --> stateIdx: 6

Figure 5.11: Mapping

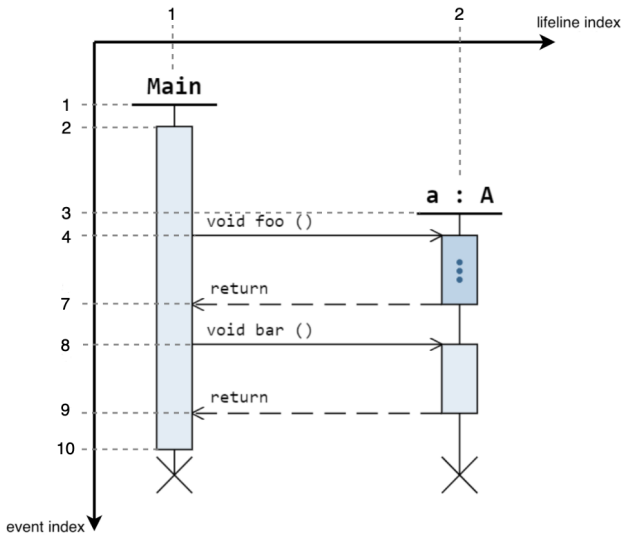


Figure 5.12: Step over a.foo()

mapping for activation a.foo()
eventIdx: 4 --> stateIdx: 3
eventIdx: 7 --> stateIdx: 6

Figure 5.13: New mapping

6 Conclusion

This thesis has presented the development of a visualization component for sequence diagrams within the JavaWiz debugging tool. First, it introduced sequence diagrams as part of the UML standard and presented tools other than JavaWiz that follow similar visualization approaches for sequence diagrams. An introduction to JavaWiz was given by explaining the architecture and other fundamental features. JavaWiz is extended by the sequence diagram component that allows users to follow the chronological sequence of interactions between objects. The most important implementation details are described in Chapter 5.

The visualization component receives trace information from the JavaWiz backend. It extracts relevant information from the trace, which is then used for visualization. The component is implemented as a Vue.js component using the D3.js visualization framework. As programming language, TypeScript has been used.

A highlight of the implementation include the collapsing of method calls. The possibility to collapse method calls allows to break down a program into its pieces and focus on those parts that are currently of interest.

6.1 Personal thoughts

To get used to both TypeScript and D3.js, I first started off by implementing the input buffer visualization. That way, I was able to quickly understand the concept behind the trace and how I could filter information for my own purposes. This smaller project introduced me to most of the needed skills for the sequence diagram component and, therefore, facilitated an easier start.

Overall, I can definitely say that this project challenged me in various ways as I had to work with new technologies and expand my programming knowledge. Using D3.js to visualize the sequence diagram was an entertaining and creative task as it required trying out different design ideas. As a student researcher at the Institute for System Software, I had the opportunity to ask colleagues to explain specific concepts regarding certain technologies. This allowed me to quickly proceed with the project.

Bibliography

[1] *Java OOP (Object-Oriented Programming)*. URL: https://www.w3schools.com/java/java_oop.asp. (accessed: 25.04.2024).

[2] *Unified Modeling Language 2.5.1*. URL: https://www.uml.org. (accessed: 30.04.2024).

[3] Schenk F. *Realization of a JavaWiz backend system running on a JavaScript runtime*. Master Thesis, Johannes Kepler University Linz, Austria, 2024.

[4] Kern K. *JavaWiz - A Visualization Tool for Software Development Education*. Master Thesis, Johannes Kepler University Linz, Austria, 2023.

[5] Schloemicher A. *Flowchart Visualization in JavaWiz*. Bachelor Thesis, Johannes Kepler University Linz, Austria, 2023.

[6] Schenk F. *Eine Komponente zur Visualisierung von Java-Methoden für lineare Listen und binäre Bäume*. Bachelor Thesis, Johannes Kepler University Linz, Austria, 2022.

[7] Eriksson H. and Penker M. *UML Toolkit*. New York: John Wiley Inc., 1998.

[8] *Unified Modeling Language Class and Sequence Diagrams*. URL: https://open.oregonstate.education/setextbook/chapter/uml-class/. (accessed: 25.04.2024).

[9] Martin Hitz and Gerti Kappel. *UML@ Work: von der Analyse zur Realisierung*. dpunkt-Verlag, 1999.

[10] Egyed A. and Grünbacher P. *Design Modeling*. Lecture Notes Software Engineering, Johannes Kepler University Linz, Austria, 2023.

[11] Ratiu C. Marchezan L. and Rynkiewicz M. *E3: Define Architecture and Design*. Exercise Notes Software Engineering, Johannes Kepler University Linz, Austria, 2023.

[12] Kai Koskimies and Hanspeter Mössenböck. "Viewing Object-Oriented Programs Through Scenario Diagrams". In: *Proceedings of the symposium on software engineering, Visegrad, Hungary*. 1995.

[13] Mary ES Loomis, Ashwin V Shah, and James E Rumbaugh. "An object modeling technique for conceptual design". In: *European conference on object-oriented programming*. Springer. 1987, pp. 192–202.

[14] Rainer Oechsle and Thomas Schmitt. "Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi)". In: *Software Visualization: International Seminar Dagstuhl Castle, Germany, May 20–25, 2001 Revised Papers*. Springer. 2002, pp. 176–190.

[15] *Vue.js Guide*. URL: `https://vuejs.org/guide/introduction.html`. (accessed: 02.04.2024).

[16] *What is D3?* URL: `https://d3js.org/what-is-d3`. (accessed: 02.04.2024).

[17] *Reactivity in Depth*. URL: `https://vuejs.org/guide/extras/reactivity-in-depth.html`. (accessed: 02.04.2024).

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 11.06.2024
Ort, Datum

Melissa Lee
Unterschrift