JOHANNES KEPLER
UNIVERSITÄT LINZ

Flowchart Visualization in JavaWiz

Author
Andreas Schlömicher

Submission
Institut für Systemsoftware

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

Assistant Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc.

April 2024

Bachelor's Thesis

to confer the academic degree of
Bachelor of Science

in the Bachelors's Program
Informatik

Abstract

A flowchart is a graphical representation of a program which shows the operations and the control flow in a two-dimensional figure. They are used for developing software and other processes that work with a sequence of operations.

JavaWiz is a tool for helping programming beginners understanding the behaviour of Java programs and its data structures. It works like a debugger for Java applications and is available as a Visual Studio Code plugin.

In this work, flowcharts have been developed as a further visualization component of JavaWiz. Interactive flowcharts allow students to follow the step-by-step execution of a program. Flowcharts in JavaWiz support rendering a method as a flowchart, interactively stepping through the operations of the flowchart, collapsing and expanding nodes and inlining called methods at the call statement.

Kurzfassung

Ein Ablaufdiagramm ist eine grafische Darstellung eines Programms, die die Operationen und den Kontrollfluss in einer zweidimensionalen Figur zeigt. Sie werden für die Entwicklung von Software und anderen Prozessen verwendet, die mit einer Abfolge von Operationen arbeiten.

JavaWiz ist ein Werkzeug, das Programmieranfängern hilft, das Verhalten von Java-Programmen und deren Datenstrukturen zu verstehen. Es funktioniert wie ein Debugger für Java-Anwendungen und ist als Plugin für Visual Studio Code verfügbar.

In dieser Arbeit wurden Ablaufdiagramme als eine weitere Visualisierungskomponente von JavaWiz entwickelt. Interaktive Ablaufdiagramme ermöglichen es Schülern, die schrittweise Ausführung eines Programms zu verfolgen. Ablaufdiagramme in JavaWiz unterstützen die Darstellung einer Methode als Ablaufdiagramm, das interaktive Durchlaufen der Operationen des Ablaufdiagramms, das Kollabieren und Expandieren von Knoten und Inlining von aufgerufenen Methoden an der Aufrufstelle.

Contents

1 Introduction

According to [1] a flowchart is a graphical representation of the definition, analysis, or method of solution of a problem in which symbols are used to represent operations, data, flow, equipment, etc. They are used for developing software and other processes that work with a sequence of operations.

The purpose of a flowchart is to have a clear and structured view of how a program or process works. It helps to visualize the sequence of steps, decision points and possible outcomes within the process.

Flowcharts are built from different elements:

- **start and endpoints**: flowcharts begin with a start point and end with an endpoint. These represent the beginning and end of the process or system being represented. The start point can be shown in different ways, either as a rounded rectangle or a triangle.
- **decisions**: decisions in a flowchart are represented by diamond-shaped symbols. They indicate points in the process where a decision must be made based on certain conditions or criteria. Depending on the outcome of the decision, the flow of the diagram follows different paths. Typically, the decision symbol will have outgoing arrows. Each arrow is labelled with a condition or criteria that determine the path to be followed.
- **tasks**: tasks, also known as processes or actions, represent the specific actions or steps performed in the process. These can include calculations, data processing, input and output, or any other action required to complete the process. Tasks are presented with a textual description, possibly within a box.

The symbols described above are most commonly used in business process flowcharts, as flowcharts used to represent software may require more differentiation in the visualization of certain elements [2].

Flowcharts are valuable tools for documenting, analysing and improving processes. They are useful for the identification of bottlenecks, inefficiencies or areas for optimisation. They also aid communication, training, and troubleshooting by clearly and intuitively representing a process.

In the specific context of imperative programming, flowcharts are used to show statement sequences, branches and loops [3]. In particular, it shows the control flow of procedures. This thesis focuses exclusively on flowcharts for visualizing procedures (Java methods).

1.1 JavaWiz

JavaWiz is a tool for visualizing fundamental aspects of programming and is designed to help beginners understand the behaviour of Java programs and data structures. It can be used as a debugger for Java applications and is available as a Visual Studio Code plugin or in a web page.

JavaWiz is already able to create the following visualizations:

- **Desk test**: a tabular view showing executed statements and variable values. For each step, a row is created which makes it easy to see the changes of variables and conditions.

- **Stack and heap visualization**: It is a representation of the current memory with the data in the stack and the heap.
- **Array visualization**: The purpose of this visualization is to show the manipulation of arrays.
- **List visualization**: Special visualization of linked lists.
- **Tree visualization**: This visualization is used for visualizing binary trees.

1.2 Motivation

Computer programming education faces the problem of introducing students to complex programming concepts, as it can be overwhelming for beginners to understand these concepts, syntax and logic. By integrating flowcharts into JavaWiz, learners can benefit from a more engaging learning experience and are provided with a valuable visual aid that helps them to trace program flow and identify logical errors more effectively. Interactive flowcharts allow students to follow the step-by-step execution of a program.

Flowcharts should allow:

- **Render a flowchart of a method**: The flowchart should show the method signature, the sequence of statements, branches and loops, as well as exception handling.
- **Step through the flowchart**: This includes the ability to execute the method step by step, following the path of the flowchart. The next statement to be executed should be highlighted. The visualization should always include the method currently being executed.
- **Collapse and expand nodes**: Flowcharts can become complex, especially for larger methods. The ability to collapse nodes would be helpful to simplify the view and focus on specific sections. Collapsing a node would hide the details within it, making the flowchart more concise. Conversely, expanding nodes would reveal the hidden details, providing a more detailed view of the method's logic.
- **Inline methods in the flowchart**: To inline methods is to include the flowchart of called methods directly in the diagram. Instead of displaying method calls as a separate view, the flowchart of the called method is expanded inline within the calling method's flowchart. This helps to visualize the complete execution flow in a single diagram, without navigating to different parts of the flowchart for each method call.

1.3 Structure of the thesis

This thesis is structured in the following way:

- Section 2 discusses work related to this thesis.
- Section 3 is a user guide for JavaWiz flowcharts. It shows various features by examples.
- Section 4 gives an overview of the implementation and guides the developer in implementing new features and maintaining the code base for the flowcharts.
- This thesis is finally summarised in Section 5.

2 Related Work

Flowcharts have been around for a long time. Creating flowcharts from source code is not a new idea either. This chapter takes a look at the rich history of flowcharts, explores the various variants and notations that exist, and highlights the range of tools available for creating flowcharts by hand or from source code.

2.1 History of flowcharts

People like graphical visualizations and they help to share and discuss complex technical structures and processes. They can also be used to document a process. According to Stritzinger [4], flowcharts always were a popular way for algorithm design. However without any existing standard many different variants exist, the only commonality being the use of arrows to show the flow.

An alternative to flowcharts were to use the Nassi-Shneiderman diagrams. They were easy to layout and, at the time, easy to print on a matrix printer [2]. Moreover, they do not allow GOTOs and thus promote a structured programming approach.

A short-lived standard was DIN 66001. This standard was designed for general data flows and was not specific to programming.

2.2 Variants

As already mentioned in the previous section, there are several variants of flowcharts. Some of them are:

- **Activity diagram in UML**: Flowcharts are not part of the Unified Modelling Language (UML)[5]. The primary focus of UML is on the modelling of software systems using a set of standardised diagrams. However, if you want to represent flow-like behaviour in UML, you can use activity diagrams. Activity diagrams in UML are used to show the flow of activities, actions and decisions within a system. Although they are not exactly the same as traditional flowcharts, they do provide a visual representation that is similar. While activity diagrams can represent flow-like behaviour, they also provide additional features such as parallelism, making them more expressive than traditional flowcharts.
- **Structogram**: The structogram, also known as the Nassi-Shneiderman diagram, is a graphical representation used in software engineering to show the flow of control in a program. They provide a structured way of representing algorithms and program logic[2].
- **DIN 66001**: As already mentioned, the German standard DIN 66001 is obsolete because it does not support the new structured programming paradigm of the 1980s. This paradigm banned the use of GOTO statements.
- **ISO 5807**: This standard defines symbols and provides application guidance. It is no longer in use, but has had a major influence on other standards.
- **Visualization by Stritzinger and Blaschek**: According to Strizinger and Blaschek, the existing technology for rendering algorithms was not sufficient, so they developed their own visualization [3], [4]. This visualization has been largely adopted in this thesis. They adopted a flowchart visualization from Rechenberg[6].

2.3 Tools

There are several tools available for creating flowcharts. Most require manual drawing of nodes and arrows, but some are able to create flowcharts from source code.

Some tools are:

- **diagrams.net**: Also known as draw.io is a web app for creating diagrams [7]. It is a diagram editor that requires content to be added manually and has very good integration with various cloud services and export and import options.
- **Lucidchart**: A tool similar to diagrams.net
- **Microsoft Visio**: A tool from Microsoft for the creation of different types of diagrams, including flowcharts. It is part of Microsoft Office.
- **Code Iris**: A plugin for the IntelliJ IDEA IDE that supports Groovy and Java visualization of modules, packages and classes.
- **Code2Flow**: A tool that creates flowcharts from pseudocode. It does not support Java.
- **Dia**: A popular flowchart editor. Last released in 2014, it is similar to Microsoft Visio. It is open source and currently under active maintenance.
- **LJV**: An extension for Jswat created by John Hammer [8]. Jswat is a graphical Java debugging front-end. It is possible to include it as a dependency in a Java project [9]. It has been discontinued as of 2013 [10].
- **Aaron**: Aaron is a structure-oriented editor for Modula-2 that uses a flowchart editor instead of source code. Aaron is able to generate Modula-2 source code from the generated flowchart [3]. This IDE was developed by Günther Blaschek with the help of Alois Stritzinger and Johannes Sametinger around 1984-1987.

  Project Aaron was divided into several parts. One part was a truncated syntax tree, which has a similar purpose to the backend used in this thesis. Another part is the flowchart visualization made by Stritzinger [4], which is similar to the frontend part of this thesis.

  They use very similar algorithms to place elements on the diagram, as they also use the centre, height and width of the element to position it. They also perform multiple recalculations of the placements to fit the available space.

Many of these tools have been around for a long time and may no longer be actively maintained.

3 JavaWiz Flowcharts

The JavaWiz Flowcharts component is able to create flowcharts of code using the basic Java language features. When the debugger is started, the flowchart of the **`main`** method is created and extended as the user continues to execute the program as seen in Figure 1.

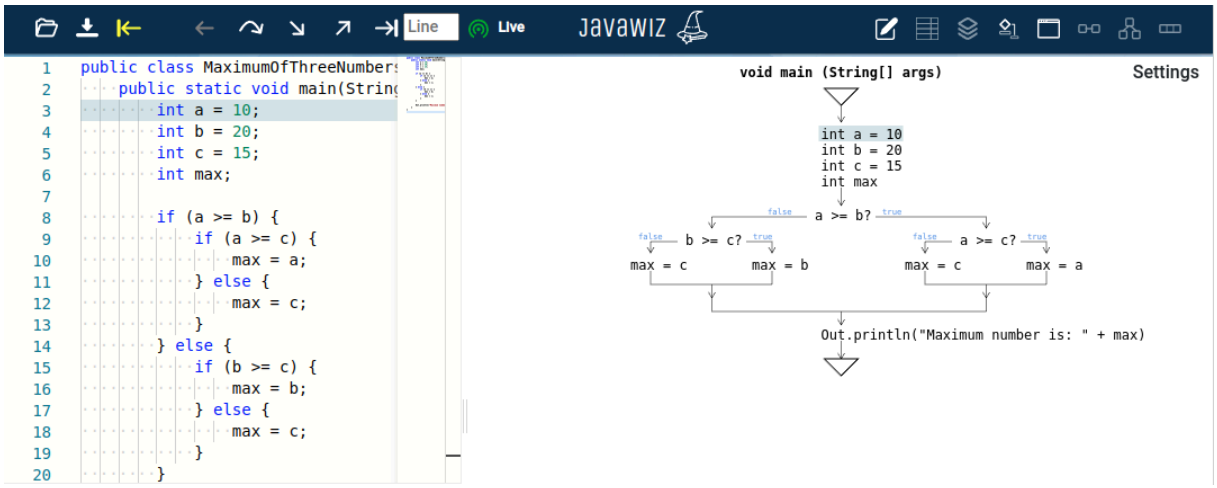


Figure 1: JavaWiz flowchart showing the main method

In this chapter, the visualisation of the different language elements of Java and the supported features are presented.

3.1 Method

The JavaWiz flowcharts focus on rendering methods and their content.

The first example Listing 1 is the well-known “Hello World” in Java.

```
public static void main(String[] args) {
  System.out.println("Hello world!");
}
```

Listing 1: *Hello world* program

In Figure 2, one can see the flowchart for this method. At the top, the signature of the entry method **`main`** is displayed. Below the signature is the entry point symbolized as a triangle with an arrow below that points to the first statement. At the end of the chart, the exit point is symbolized by a triangle pointing down with an arrow on top.

A light blue background highlights the next executed statement.

When the program jumps to another method, the flowchart is removed and the new method is rendered.

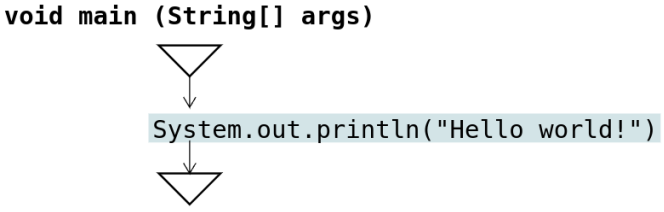


Figure 2: *Hello world* program

3.2 Conditionals and loops

The rendering of if-statements and all loops is implemented. The internal handling between if-statements and loops is very similar and is therefore combined in this section.

3.2.1 If-statement

In this section, if-statements are shown with all their variants and options.

As an example of the simple if-statement, the Listing 2 has a condition, a then-branch that sets **x** as the maximum and an else-branch that sets **y** as the maximum. Both branches print the maximum to the output.

```
if(x > y) {
  Out.print("x ist max");
  max = x;
} else {
  Out.print("y ist max");
  max = y;
}
```

Listing 2: Finding the maximum between **x** and **y**

The Figure 3 shows the flowchart of the if from Listing 2. The condition indicated by the question mark is the starting point. On the left side marked by the **false** keyword is the else-branch and on the other side is the then-branch marked by the **true** keyword. Arrows enter and leave the block of statements in each branch.

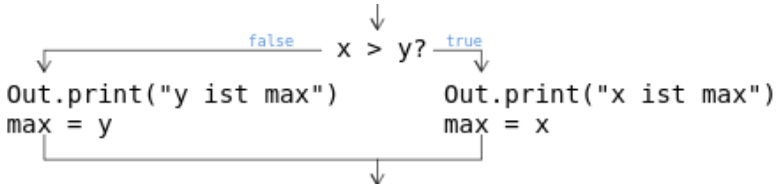


Figure 3: Finding the maximum between **x** and **y**

The rendering position of the branches can be swapped in the settings.

The default rendering is the true branch on the right, as shown in Figure 3. After changing the position of the branches in the settings, as seen in Section 3.6.1, the rendering of if-statements changes immediately, which is shown in Figure 4.

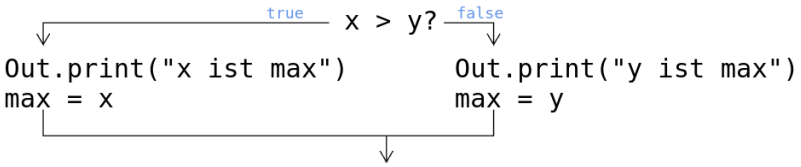


Figure 4: Then-branch on left side

A form of if-statements is the **if-statement without else-branch**.

In Listing 3 the program checks if the divisor **n** is not zero in order to avoid a division by zero exception. If it is not zero, **q** is set to **x / n**.

```
if(n != 0)
  q = x / n;
```

Listing 3: If-statement as a divide by zero guard

In Figure 5 is shown that if-statements without an else-branch have a continuous line to the end of the if-statement instead of the block. The line has a right-turned *U*-shape. The size of the *U* is set to fit the **false** keyword at the top of the line.

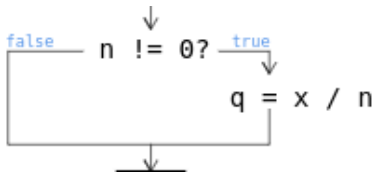


Figure 5: If-statement as a divide by zero guard

When working with **nested if-statements** the visualization of if-statements is essential for keeping it clear and easy to read.

The Listing 4 uses nested if-statements for the comparison of three numbers.

```
if(a > b) {
  if(a > c) {
    max = a;
  } else {
    max = c;
  }
} else {
  if(b > c) {
    max = b;
  } else {
    max = c;
  }
}
```

Listing 4: Nested if-statements finding the maximum in three numbers

The two-level nesting is clearly visible in Figure 6. Depending on the first comparison the nested if-statement is executed.

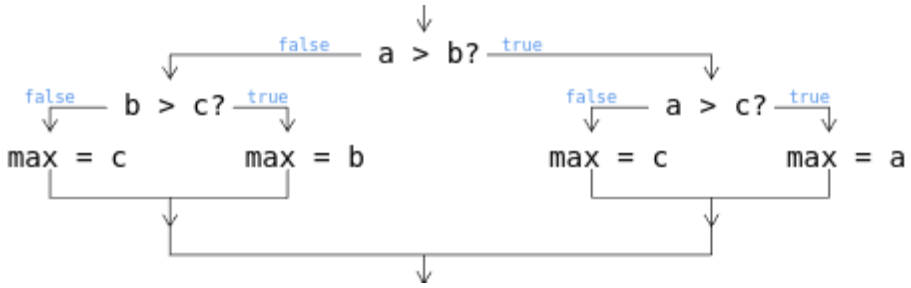


Figure 6: Nested if-statements finding the maximum in three numbers

The importance of the visualization of if-statements is also highlighted in a **if-statement cascade**.

In Listing 5 is the value check of **x** shown, which is implemented with cascading if-statements.

```
if(x > 1000) {
  Out.println("x is very big");
} else {
  if(x > 100) {
    Out.println("x is big");
  } else {
    if(x > 0) {
      Out.println("x is positive");
    } else {
      if(x < 0) {
        Out.println("x is negative");
      } else {
        Out.println("x is zero");
      }
    }
  }
}
```

Listing 5: Cascading if-statements of checking the value of **x**

As shown in Figure 7 this results in a deeply nested flowchart that is leaning heavily on one side.

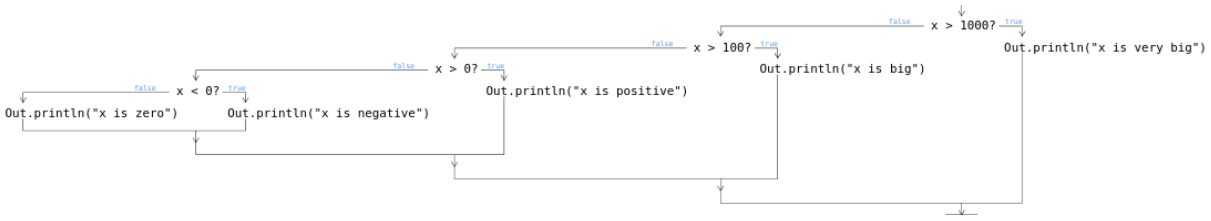


Figure 7: Cascading if-statements of checking the value of **x**

If-statements with returns are shown differently. First a text followed by a triangle representing the return statement. Then a dotted line indicates that the if does not continue. If both sides do not continue, the arrowhead is omitted and a dotted line connects both branches.

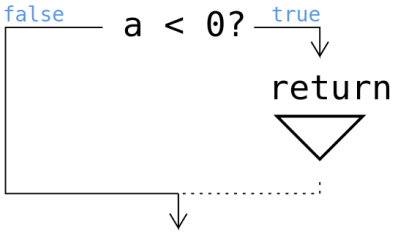


Figure 8: Return-statement on one side

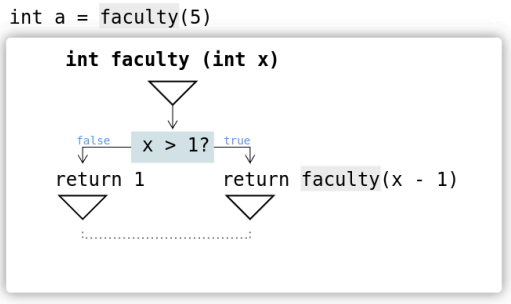


Figure 9: Return-statement on both sides

3.2.2 While-loop

JavaWiz supports all the basic loops used in Java and one of them is the while-loop.

In the example as shown in Listing 6, the user can enter numbers until he finishes by entering a non-number. Each number is added up to a total.

```
int x = In.readInt();
int sum = 0;
int n = 0;
while (In.done()) {
  sum = sum + x;
  n++;
  x = In.readInt();
}
```

Listing 6: Sum of several integers entered by the user

Figure 10 represents the flowchart for Listing 6. The while-loop has a diamond at the beginning that contains the condition of the loop inside. An arrow points from the diamond to the first statement. The loop body statements are in the center of the loop and at the bottom is an arrow pointing to the statement after the while-loop. In this arrow, a circle marks the start of another arrow that returns to the condition. The width of the diamond adapts to the length of the condition.

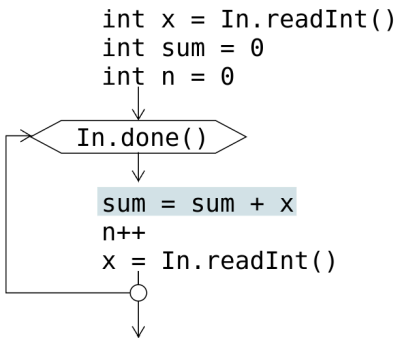


Figure 10: Sum of several integers entered by the user

A very similar loop is the **do-while-loop**. Listing 7 contains the code for asking the user for a number input that is greater than zero. The do-while-loop is used to enforce a retry until a valid input is entered.

```
int number;
do {
  Out.print("Please enter a number greater than 0: ");
  number = In.readInt();
} while (number <= 0);
```

Listing 7: User input of a number greater than 0

The flowchart in Figure 11 shows that the diamond is at the bottom of the loop and the arrows with the circle are at the beginning of the loop.

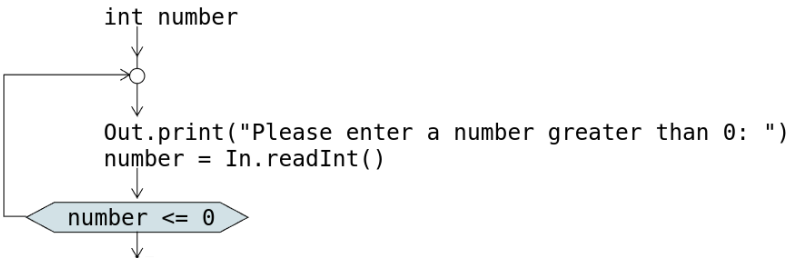


Figure 11: User input of a number greater than 0

3.2.3 For-loop

The visualization of for-loops is the same as while-loops, except that the header diamond contains the initialisation, comparison and update-statements, separated by semicolons.

The code in Listing 8 shows how to sum up all numbers from 1 to **n** and the resulting flowchart is shown in Figure 12.

```
int n = 10;
int sum = 0;
for (int i = 1; i <= n; i++) {
  sum = sum + i;
}
```

Listing 8: Sum up all numbers from one to ten with a for-loop

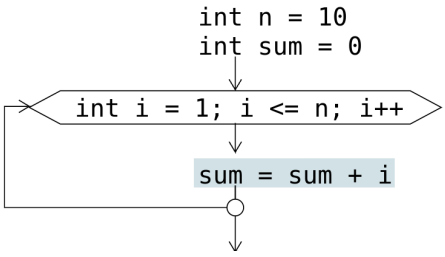


Figure 12: Sum up all numbers from one to ten with a for-loop

An interesting example is Listing 9, where a $n \times n$ multiplication table with $n = 4$ is printed. Here two nested for-loops are used. The inner loop prints the columns and the outer loop is responsible for the rows.

```
int n = 4;
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
      Out.print(String.format(" %3d", i * j));
  }
  Out.println();
}
```

Listing 9: Multiplication table of 4

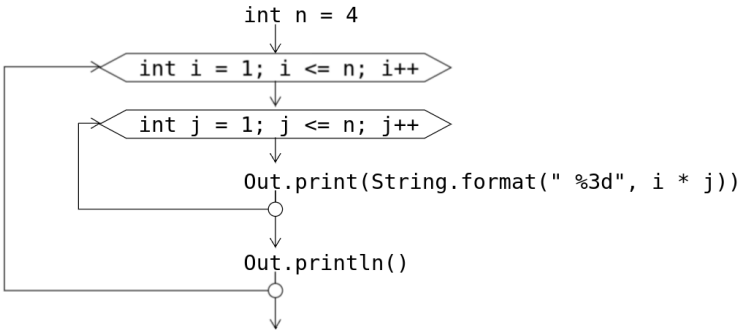


Figure 13: Multiplication table of 4

3.3 Switch-statement

JavaWiz flowcharts also supports the visualization of switch-statements. However, switch-expressions are shown as source code only.

A basic example of a switch-statement is shown in Listing 10, where the user input is used to print the day of the week. If the input is not between one and seven, the default case prints an error message.

```
int day = In.readInt();
switch (day) {
  case 1:
    Out.print("It's Monday");
    break;
  case 2:
    Out.print("It's Tuesday");
    break;
  case 3:
    Out.print("It's Wednesday");
    break;
  case 4:
    Out.print("It's Thursday");
    break;
  case 5:
    Out.print("It's Friday");
    break;
  case 6:
    Out.print("It's Saturday");
    break;
  case 7:
    Out.print("It's Sunday");
    break;
  default:
    Out.print("Wrong Input: " + day);
}
```

Listing 10: Print the name of a day with a day index as input

In Figure 14 the flowchart of the switch-statement of Listing 10 is shown. On top is an arrow-like rectangle showing the selector of the switch-statement. Then, the cases are chained from top to bottom. On the right side of a case the statements are positioned. At the bottom, a

smaller arrow-like rectangle and an arrow form the end of the switch-statement. The selector rectangle size is adapted to the length of the switch-selectors name. The centers of the blocks inside each case may vary as they are relative to the case's conditions.

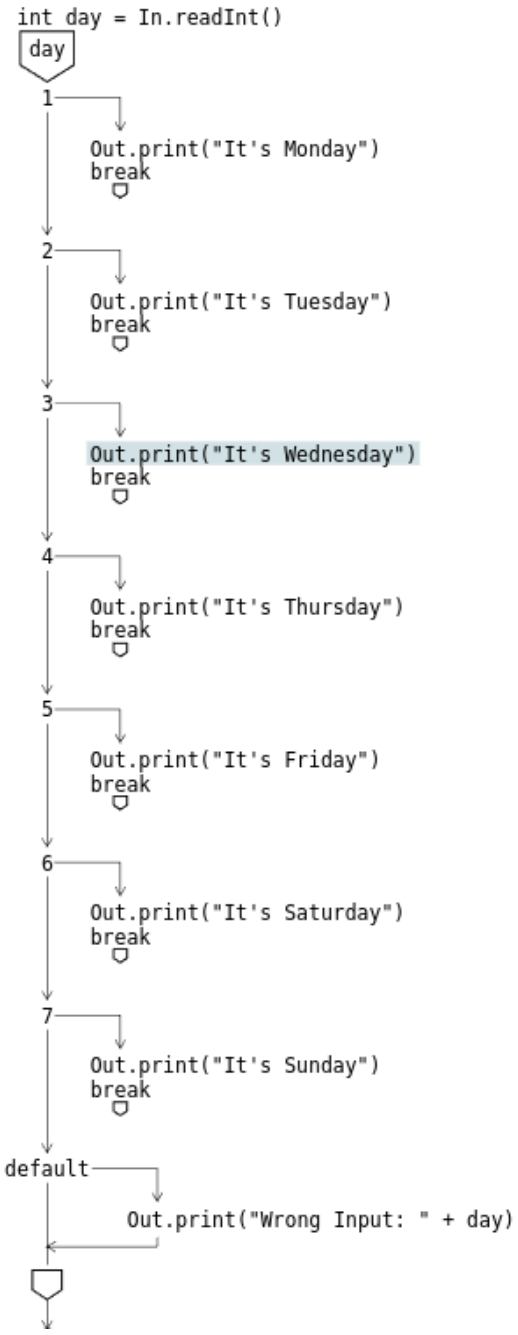


Figure 14: Print the name of a day with a day index as input

There are a few more things to consider for switch-statements. Figure 15 shows that when multiple cases are combined the case conditions are joined by a comma. It also shows that if the break or return-statement is omitted, an arrow goes back into the cases conditions chain.

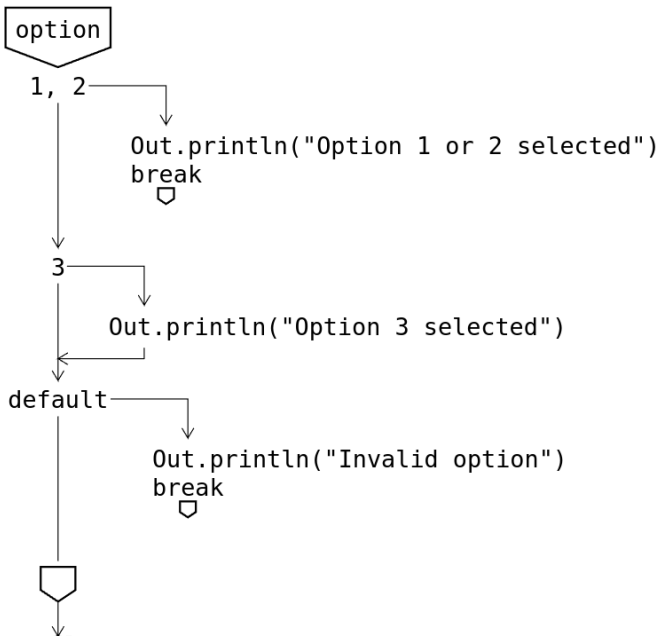


Figure 15: Handle multiple options with a switch-statement

3.4 Try-catch-finally-statement

Exceptions are very disruptive to the flow of programs. The design goal of catch blocks was to minimize the required space until an exception is thrown. The try-block and finally-block have minimal changes compared to an ordinary block to keep the user's focus on the normal control flow.

Listing 11 shows how to handle the exception of a division by 0 by a try-catch-statement. The division is surrounded by a try-block followed by a catch-block for the **`ArithmeticException`** and then the general catch-block for all exceptions, both printing an error message.

```
try {
    int result = divide(10, 0);
    Out.println("Result: " + result);
} catch (ArithmeticException ex) {
    Out.println("ArithmeticException occurred: " + ex.getMessage());
} catch (Exception ex) {
    Out.println("Exception occurred: " + ex.getMessage());
}
```

Listing 11: The division of $\frac{10}{0}$ enclosed by an try-catch-statement

The chart in Figure 16 shows the try-block on the left side and the catch clauses on the right side and a thick black line in between. There are thin grey lines at the top and bottom to show that the program is able to run on both sides of the chart.

The try-block is displayed as the blue **`try`** keyword on the left and arrows pointing to the first statement and at the try-block.

The chain of catch-clauses starts with the blue **`catch`** keyword and an arrow pointing right to the first catch-clause and continues to the right. Catch-clauses are collapsed by default, but are automatically expanded when the relevant exception is caught, or by double-clicking on **`{…}`**. Figure 17 shows the expanded state, consisting of the name and type of the exception, the statements in the catch clause, and the arrows at the beginning and end.

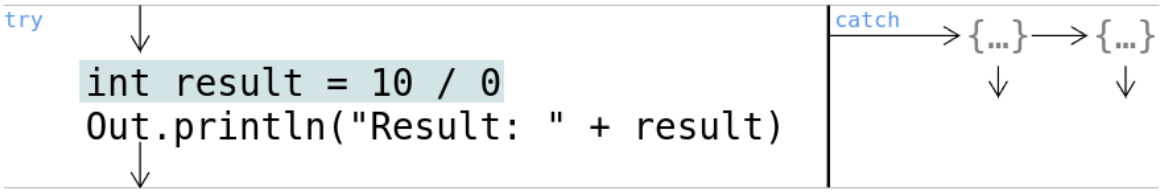


Figure 16: The division of $\frac{10}{0}$ enclosed by an try-catch-statement

When a finally-block like in Listing 12 is added, an area is added to the bottom that spans the entire try-catch-finally-statement, which can be seen in Figure 17. It also has a thin grey line at the bottom and is tagged by the **`finally`** keyword.

```
finally {
  Out.println("Finally block executed.");
}
```

Listing 12: The division of $\frac{10}{0}$ enclosed by an try-catch-finally-statement

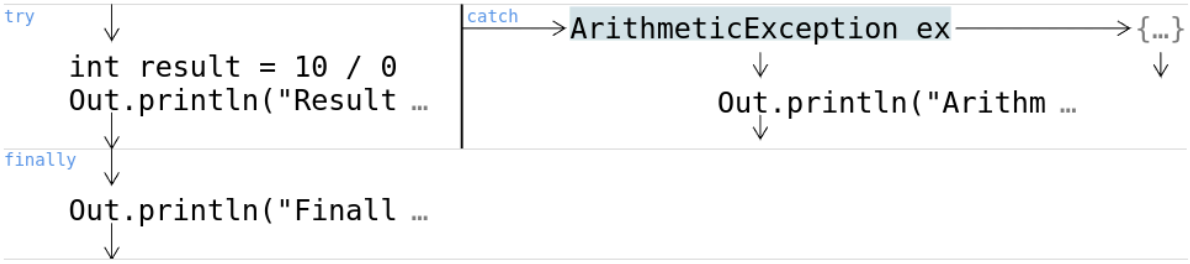


Figure 17: The division of $\frac{10}{0}$ enclosed by an try-catch-finally-statement

3.5 Statements

Statements that have no explicit JavaWiz implementation for flowcharts are represented as **`Statement`**. The default representation is a single-line text of the source code but some special cases exist. Depending on the statement, it is possible to render inlined methods and symbols that show the continuation of the flow.

- **default rendering:**

```
int b = faculty(3)
```

Figure 18: Default rendering of a Statement

in the default rending the source code is rendered in a single line. The center of the statements is at 20px to the left. Method call names of inlinable methods have a light grey background.

- **break:** has an empty circle below the statement. This represents the circle in loops where the loop is exited.

Figure 19: Return of **`value`** in e.g. a method

- **yield:** uses the same symbol as break-statements
- **throw:** has a thick black lightning bolt below the statement

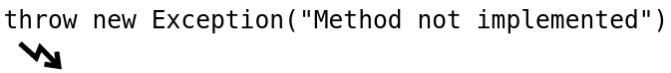


Figure 20: Throw-statement if a method is not implemented

- **return:** a triangle below the statements source code symbolizes the method exit. It looks the same as the exit of a method.

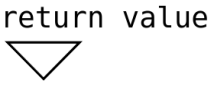


Figure 21: Return of **`value`** in e.g. a method

- **continue:** the diamond below the continue text symbolises the loop condition header.

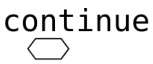


Figure 22: Contine-statement with diamond symbol

- **before a conditional or loop:** between a **`Statement`** and a conditional or loop condition an arrow appears.

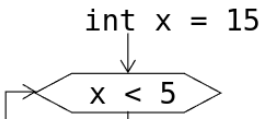


Figure 23: Arrow of statement pointing into while-loop condition

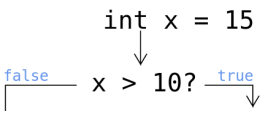


Figure 24: Arrow of statement pointing into if-statement condition

3.6 Settings and options

The flowchart visualization supports various settings and visualization options. These will be presented in this subsection.

3.6.1 Settings

The user can customise some rendering behaviours in the settings menu, as shown in Figure 25. It can be opened by clicking on **`Settings`**.

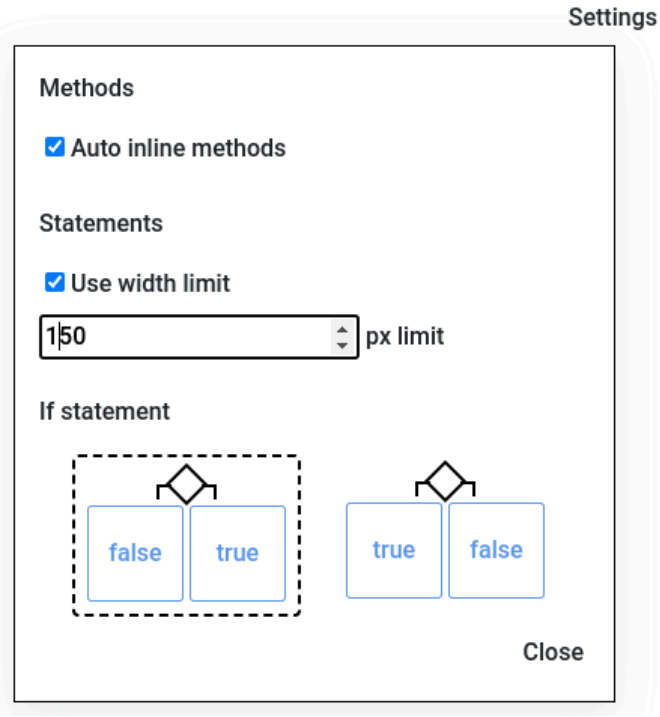


Figure 25: Settings dialog

The available settings are:

- auto inline methods, see Section 3.6.4
- width limit in statement and conditions, see Section 3.6.2
- placement of branches of if-statements, see below

It is possible to switch the placement of then and else-branches of if-statements. In the settings menu one can select one of those options. The dashed line around the placeholder shows whether the then or else-branch is rendered on the left or right.

3.6.2 Width limit of statements and conditions

In the settings, the user is able to set a width limit in pixels for statements and conditions. The default settings is *no width limit* and, if the limit is activated, the limit is preset to *150px*.

```
System.out.println( …
```

Figure 26: Statement rendered with a width limit at 150px of **`System.println.out("Hello world!")`**

```
System.out.println("Hello world!")⊟
```

Figure 27: Expanded state of statement with width limit

In Figure 26 the statement is longer than *150px* and cut off with an ellipsis. The user is able to expand a single statement by double-clicking the ellipsis. Expanded statements as seen in

Figure 27 end with a tick-box with a minus inside. By clicking the box the user can undo the expansion of the statement.

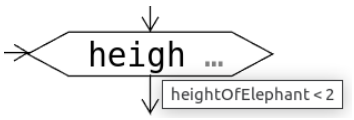


Figure 28: Loop with width limit

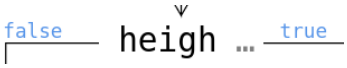


Figure 29: If-statement with width limit

Statements with a width limit show the full statement on mouse hover as a tooltip.

3.6.3 Collapsable elements

It is possible to collapse code blocks and exceptions in the chart to save space and make it easier to read.

The user can double-click on some parts of the chart to collapse or expand a code block or exception. Generally, it will toggle the collapse state on the first block of code or exception containing the element.

However, in loops and if-statements it is also possible to toggle the state by double-clicking on the condition, or in switch statements on the selector. If it is a condition or a switch selector, the code blocks within these elements are either collapsed or expanded depending on the current expansion state of the majority of these blocks.

Collapsed exceptions are shown as **{…}** and an arrow.

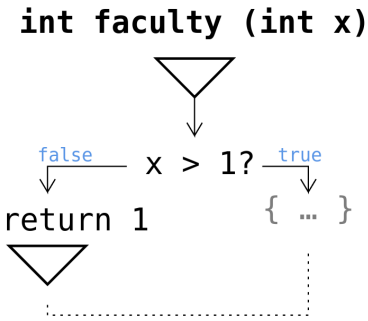


Figure 30: Collapsed block in an if-statement

In Figure 30 you see that collapsed blocks are shown as **{…}**. If the user clicks on the condition **`x > 1?`** the false-branch would also be collapsed.

When the program is executed, JavaWiz flowcharts opens all collapsed elements that should be visible in the flowchart. Visible elements are those in the AST directly between the root and the currently executed line. Their children can still be collapsed.

Only catch-clauses are collapsed at the start, and the others are only collapsed or expanded by user action.

3.6.4 Inlining methods

In order to open or close **inlined methods** the user can double-click on the method names with grey background in statements or conditions.

In Figure 31 the method **`faculty`** is rendered inline. Inlined methods have a rectangle with a shadow that surrounds the method. By double-clicking on **`faculty`** in **`int a = faculty(5)`** the inlined method can be closed.

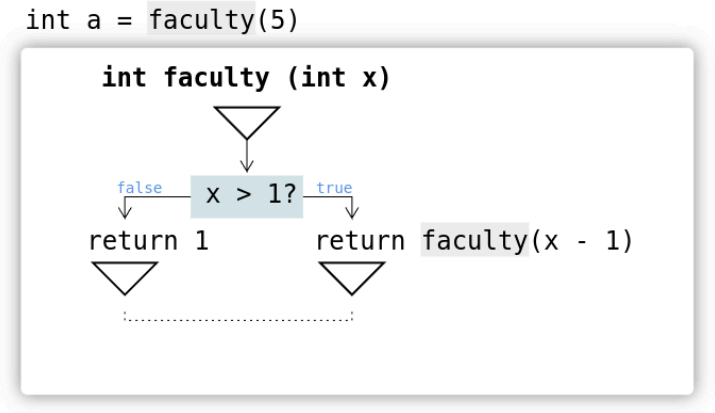


Figure 31: Method **`faculty`** rendered inline

By double-clicking on **`faculty`** in **`return faculty(x - 1)`** shown in Figure 31 it is possible to render an inlined method in an inlined method, as shown in Figure 32.

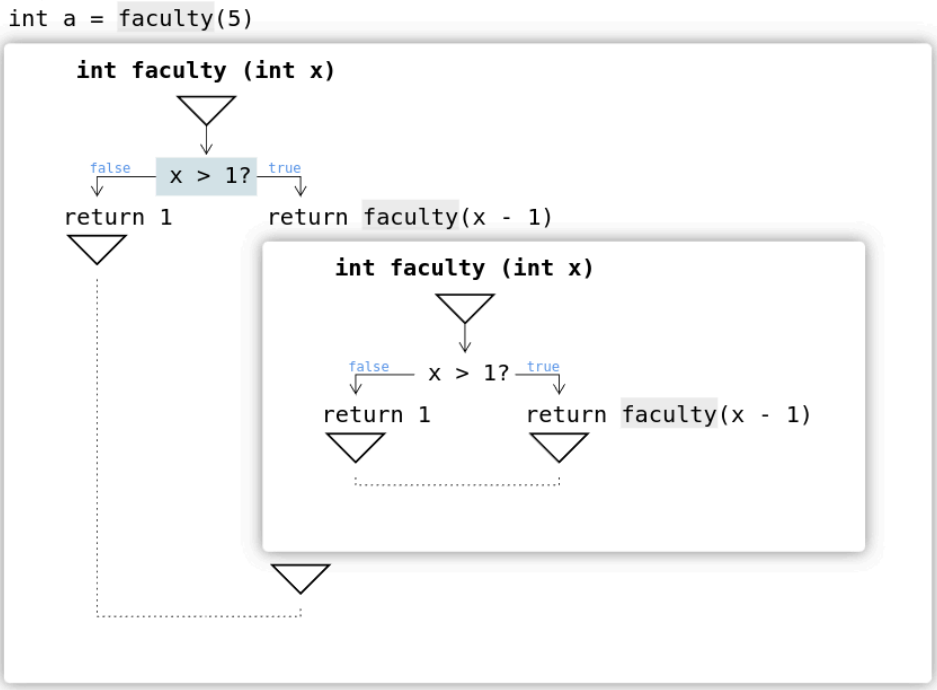


Figure 32: Recursive method **`faculty`** opened two levels deep

In the settings, the user can enable *Auto inline methods*. In this mode, JavaWiz analyzes the next executed statement in combination with the current stackframe and opens the next called method as an inlined method if this is possible. If an inlined method is exited, it is closed automatically.

3.7 Navigation

The JavaWiz Flowcharts component allows you to easily view parts of the flowchart.

It is possible to drag the chart with the mouse and scale it by scrolling. Double-clicking on an empty space restores the scale and position of the chart.

4 Implementation

This chapter focuses on the implementation and the used technologies.

First, the architecture of the system is explained, then an overview of the used technologies is given, and finally, the implementation of the backend and the layout of the flowcharts are presented.

4.1 Architecture

JavaWiz consists of a backend server and a frontend web application that can be embedded as a plugin in Visual Studio Code.

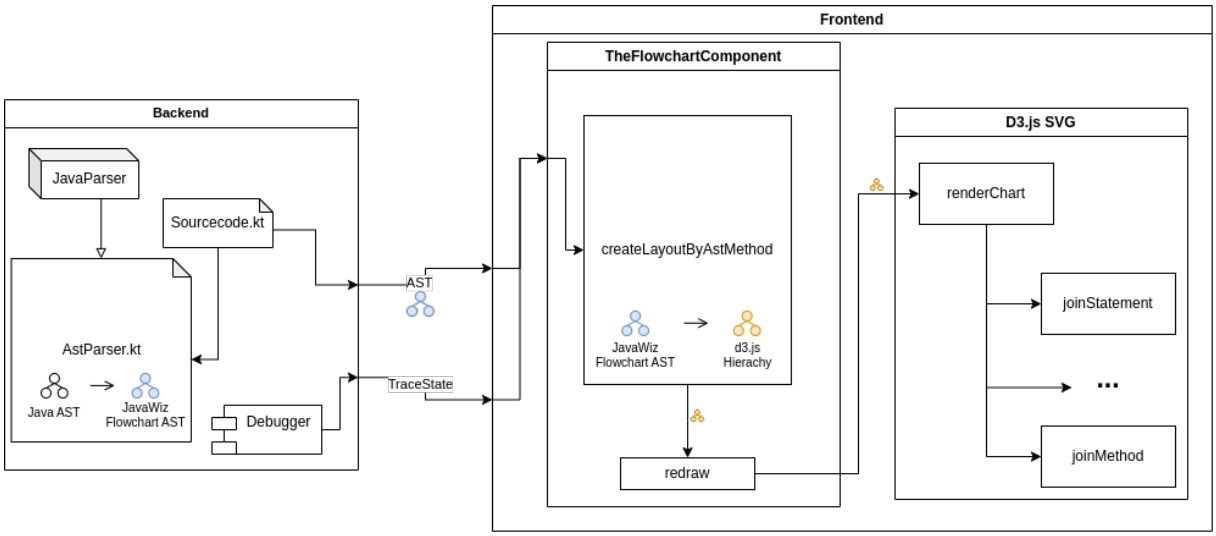


Figure 33: Architecture of JavaWiz with flowcharts

The backend is responsible for transforming the Java source code into data that can be used for rendering the flowchart. This is explained in more detail in Section 4.3.

The backend server and the frontend communicate via a websocket. The frontend sends commands to the backend. The most important for the flowcharts are the **`COMPILE`** and **`STEP_*`** commands. On compilation, the backend returns the ASTs and also the first trace state (class **`TraceState`**), which contains the current stackframes and the line number of the next statement to be executed. Each step command updates the trace state.

The frontend's job is to render the backend's data as a flowchart, as it has to transform the backend data several times to create a renderable layout. This is described in more detail in Section 4.4.

4.2 Technologies

The frontend uses modern web technologies such as Vue.js, a Javascript framework and D3.js, a visualization library. It is based on components, each containing Javascript, CSS and HTML. Typescript is used instead of Javascript because Typescript provides static type checking by the compiler. The result is better tooling and code quality.

4.2.1 D3.js

For the visualization of the flowchart, D3.js is used. It is used to create the *Scalable Vector Graphic* (SVG) of the flowchart. This is because D3.js manipulates the *Document Object Model* (DOM) [11]. Vue.js and other libraries also do this, but D3.js takes a more data-driven approach, hence the name Data-Driven Documents (D3).

In the following, an example will be shown to illustrate the basics.

Figure 34: Example of 5 circles rendered

Five circles with defined radius and colour will be created and arranged in a row to demonstrate how D3.js operates. Next, the circles' radius should change randomly, and with added animations, it will appear as if the circles are being shuffled.

Listing 13 creates five circles with a radius and colour set.

```
const data = [
  { id: 1, r: 3, color: "orange" },
  { id: 2, r: 4, color: "red" },
  { id: 3, r: 5, color: "green" },
  { id: 4, r: 6, color: "blue" },
  { id: 5, r: 7, color: "purple" }
];
```

Listing 13: Five circles with a radius and a colour

An SVG element as shown in Listing 14 is now added to the HTML document. In this element, the circles will be rendered. The viewport defines the SVG's visible space and is in this example 120px wide and 100px high.

```
<svg id="container" viewbox="0 0 120 100"></svg>
```

Listing 14: A SVG element used as a container

```
const container = d3.select("svg#container");
```
Listing 15: Select the container created in Listing 14

To initiate the rendering process of the circles, D3.js must obtain the SVG element from the Document Object Model (DOM). This is done via the **select** function as seen in Listing 15.

```
function render() {
  container
    .selectAll("circle")
    .data(data, (d) => d.id)
    .join((e) =>
      e
        .append("circle")
        .attr("cy", 50)
        .attr("fill", (d) => d.color)
    )
    .transition()
    .attr("cx", (d) => (data.indexOf(d) + 1) * 20)
    .attr("r", (d) => d.r);
}

render()
```
Listing 16: Rendering five circles

Now a function **render**, which handles the rendering process, is created as shown in Listing 16. The principle of D3.js is to select DOM elements and bind the data to these DOM elements. After this binding, D3.js allows to define which changes to the DOM are needed if a data item has been added, updated or deleted since the last render.

So the first thing to do is to select all circles in the container with the **selectAll** function.

Next is the **data** function, which binds the data to the previously created selection. The second argument is used by D3.js for the change detection and is a function returning the circle's ID in this case.

After this data binding is done, the **join** function is used to define the changes required for the create, update and delete events. The first parameter is for the create event, the second for the update event and the last one for the delete event. All parameters are optional, as they already have a predefined implementation in D3.js and all functions after the **join** function will always be executed.

In this example the create event is defined to create a circle SVG element and the static y-position and color are set and the other events do the D3.js default behaviour.

The x-position and the radius are updated every time so these attributes are set after the **join** function. Animations can be enabled by putting the **transition** function before changing these attributes. It is possible to modify the animation ease function, delay and more.

Now the render function is complete and is called at the last line in Listing 16.

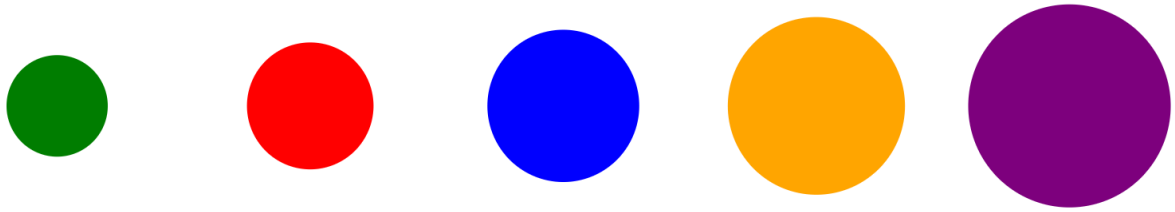


Figure 35: Example of the 5 circles shuffled and then rendered

To render the circles as shown in Figure 35 after rendering them as shown in Figure 34, we implement the following shuffle algorithm:

```
setInterval(() => {
  const a = Math.floor(Math.random() * data.length);
  const b = Math.floor(Math.random() * data.length);
  const tmp = data[a].r;
  data[a].r = data[b].r;
  data[b].r = tmp;
  data.sort((a, b) => a.r - b.r);

  render();
}, 250);
```

Listing 17: Shuffle two circles in a 250ms interval

The code shown in Listing 17 is an anonymous function called in a 250ms interval that, at first, selects two random indices **a** and **b** and then switches the radius between them. Next, the array is sorted by radius. To see how the data has changed, the **render** function is called again which could result in a view similar to Figure 35.

4.2.2 JavaParser

For analyzing the source code in the backend and retrieving the program structure for the flowchart representation, the JavaParser library is used [12]. The JavaParser creates an Abstract Syntax Tree (AST).

For flowchart visualization, this AST is traversed, the required information is retrieved and a specific AST structure is then sent to the frontend. For details see the next section.

4.3 Backend

The backend is a Kotlin implementation of a server that is used to analyse the provided source code and also runs a debugger program. The frontend needs specific information from the source code that will be rendered.

In Figure 33 the backend architecture component shows that the main work for the flowchart is done in component **AstParser**. The AstParser is used in **SourceCode**, which handles the compilation of the source code.

From the existing debugger running on the backend, the following data is used:

- **`currentFileUri`**: the URI of the current file is used to select the correct AST.
- **`stackframes`**: the stackframes are used for auto-inlining methods or highlighting the next executed statement in combination with multiple inlined methods that could be opened recursively.
- next executed statement line, also known as **`highlightLine`**: this number is used to highlight the next executed statement in the flowchart and to open inlined methods or expand collapsed elements.

This data is sent to the frontend via websocket at each step of the debugger.

At the start of the debugging session with JavaWiz, the source code is compiled and an AST is created by the JavaParser and also sent via the websocket.

4.3.1 `AstParser`

The JavaParser library creates a highly detailed AST from the source code. The AstParser is used to generate an AST specific to the flowchart visualization from the AST generated by the JavaParser. The result is a reduction in the size and complexity of the AST.

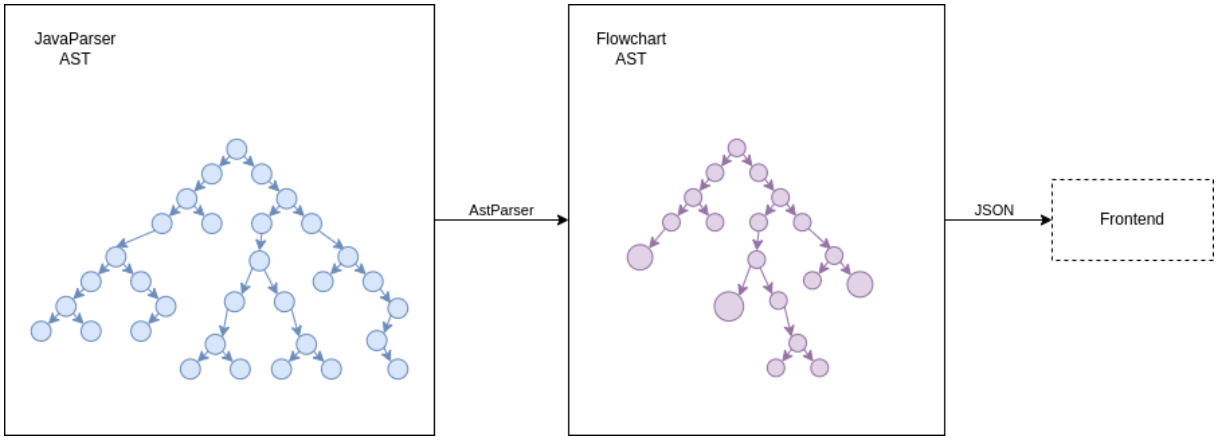


Figure 36: Transformation of the JavaParser AST into a compact AST in the **`AstParser`** class

The AST parser traverses the JavaParser AST. Each node is transformed into a new node with only the necessary information. For example, expression nodes are already contained in the statement node. This drastically reduces the size of the tree, as shown in Figure 36.

This process is started in **`SourceCode`**, which also provides the JavaParser AST via the already existing **`CompilationUnit`**.

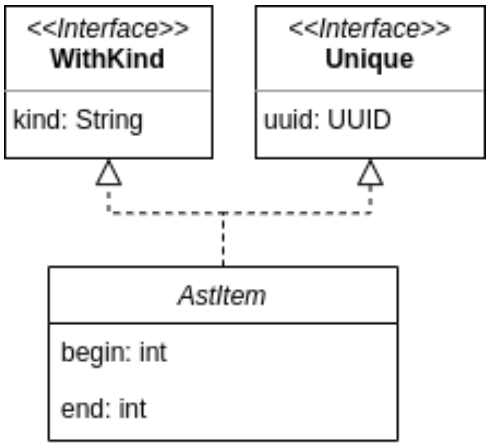


Figure 37: Class diagram of **`AstItem`**

The base class **`AstItem`** contains the beginning and ending line number, a UUID to identify a node, and a kind containing the class name of a node. As shown in Figure 37 the class implements the **`WithKind`** and **`Unique`** interfaces.

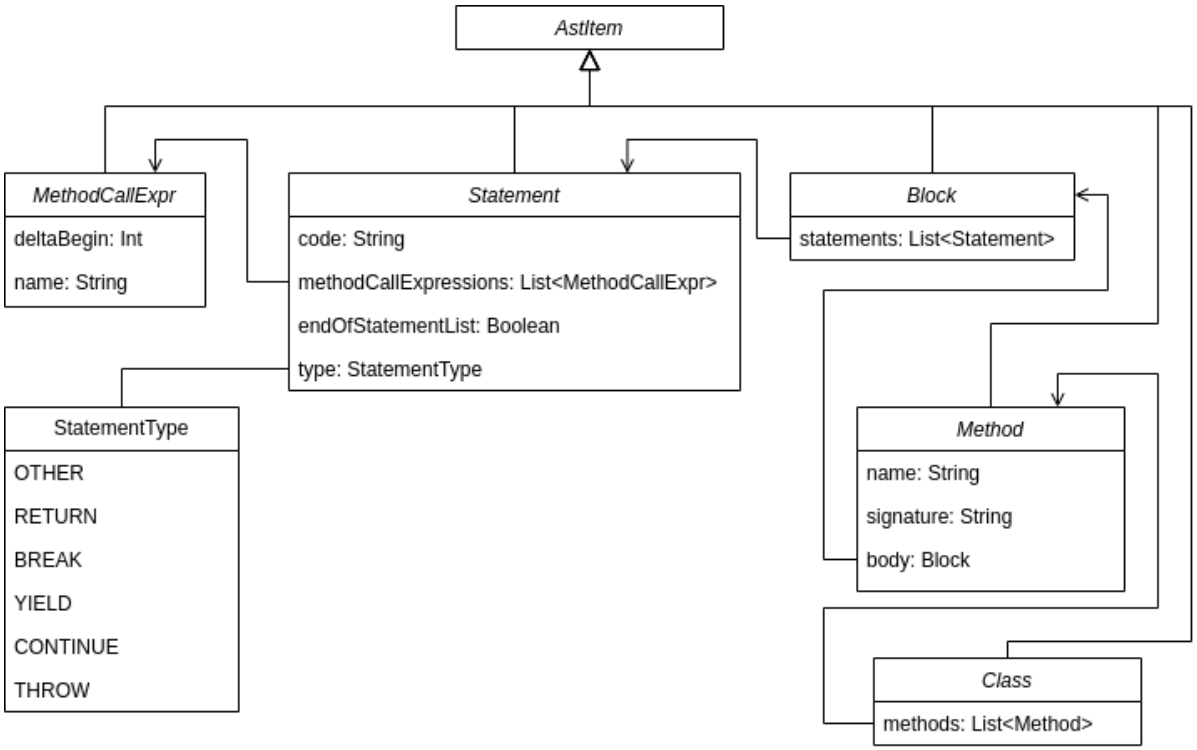


Figure 38: Class diagram of **`Statement`**, **`MethodCallExpr`**, **`Block`**, **`Method`** and **`Class`**

Figure 38 to Figure 41 depict the classes for **`AstItem.`** **`StatementType`** is an enum of specialisations of **`Statement`** and the default is **`"OTHER"`**. It is also shown that the **`MethodCallExpr`** has a **`deltaBegin`** which represents the character index in the line of source code where the expression begins, which is required to show the grey background for these method call expressions.

In **`Statement`** the **`endOfStatementList`** is calculated via the helper method **`checkLastInLine`** which checks if a **`Statement`** is the last one in the block of code or before an if-statement or loop.

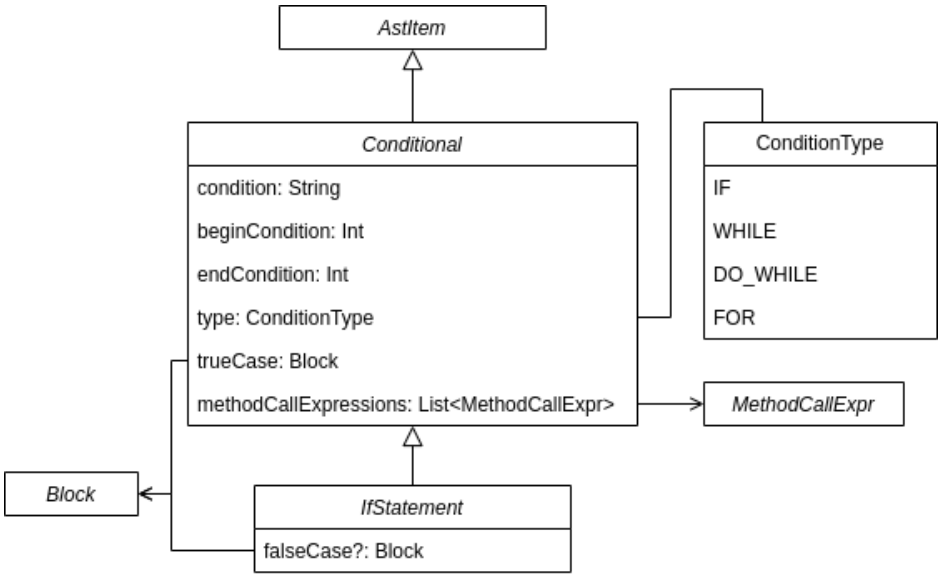


Figure 39: Class diagram of **`Conditional`** and **`IfStatement`**

In Figure 39 it is shown that **`IfStatement`** inherits from **`Conditional`**. This is important when working with the **`kind`** field as it does not reflect any inheritance and for if-statements, it is **`IfStatement`** and for any loop, it is **`Conditional`**. In short, the inheritance information is lost and must be known by the developer. The **`ConditionType`** can help in this case, as it represents a more specific type of **`Conditional`**.

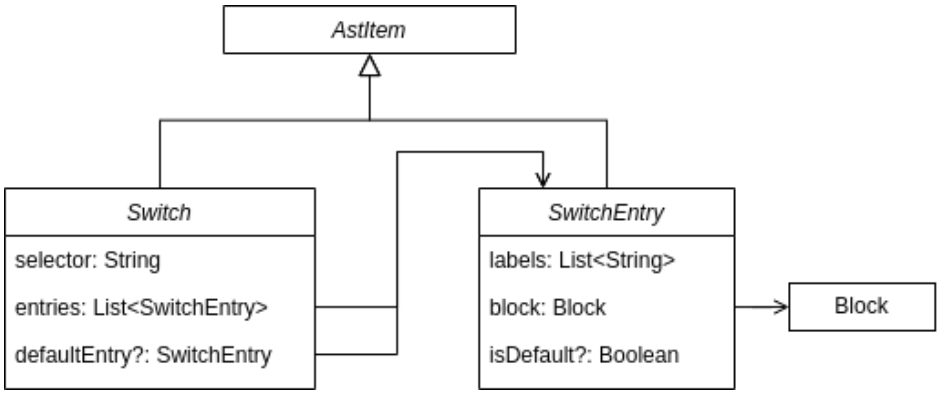


Figure 40: Class diagram of **`Switch`** and **`SwitchEntry`**

As shown in Figure 40, the **`Switch`** class has a separate field for the default entry, as it is not in the **`entries`** field. This is because the default entry is rendered below the line of labels and the other entries are rendered to the right.

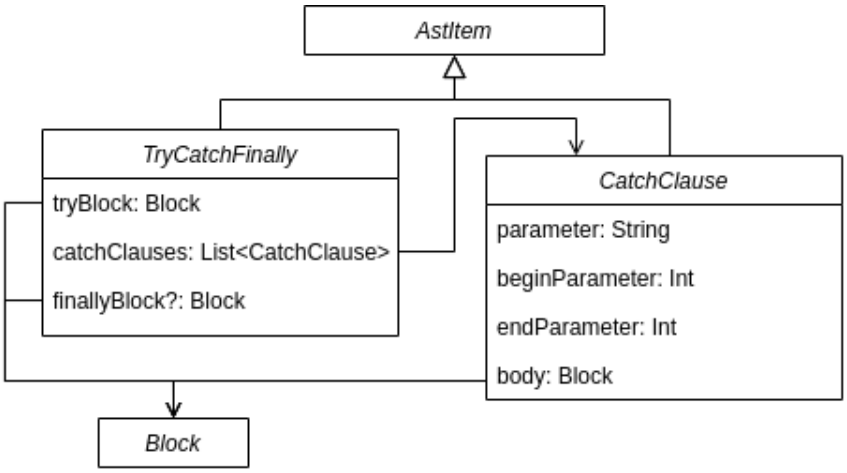


Figure 41: Class diagram of **`TryCatchFinally`** and **`CatchClause`**

In Figure 41 is shown that **`TryCatchFinally`** has an optional try-block and list of catch clauses. A **`CatchClause`** has the exception that is caught in the parameter as a string and the handler in the body.

There is also the overloaded **`findMethodCalls`** method, which finds inlinable methods for either an expression or a statement and is used to set the **`methodCallExpressions`** field, as shown in Figure 38 and Figure 39.

Listing 18 shows the source code for a main function to find the maximum of x and y and assigns the result to z. In this example, z will be 5 as y is the maximum.

```
public class FindMax {
    public static void main(){
        int z = 0;
        int x = 3;
        int y = 5;
        if(x < y){
          z = y;
        } else {
          z = x;
        }
    }
}
```

Listing 18: Program to assign the maximum of x and y to z

In Figure 42, the example AST of Listing 18 is shown. Important are the **`Block`** nodes containing the sub-nodes.

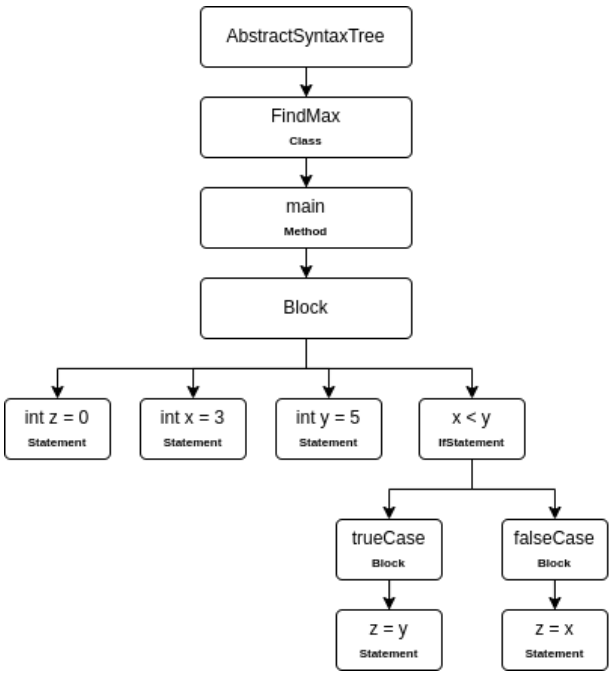


Figure 42: Object diagram of Listing 18

4.3.2 Stepping

After each step taken by the debugger, the visualization is updated. The next diagram gives an overview of the communication between the frontend and the backend.

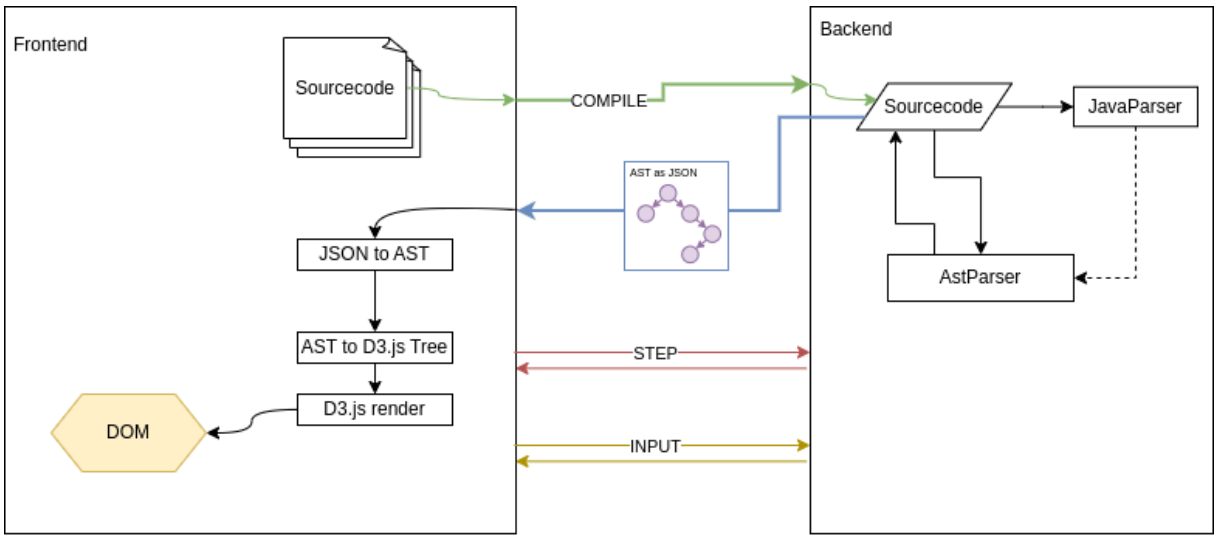
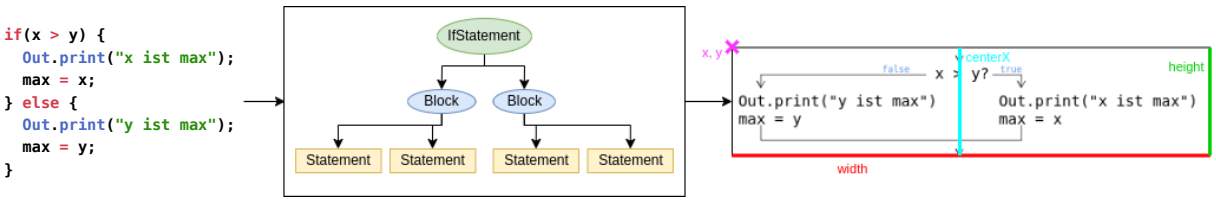


Figure 43: Communication between Frontend and Backend

As seen in Figure 43, sending the source code to the backend is the first step. This is done in the COMPILE task sent via websocket. The source code is compiled and then a debugging process is started. The parsed AST is sent back to the frontend as JSON data. This JSON data is parsed and transformed into a tree that can be rendered with D3.js.

The frontend sends STEP or INPUT commands to the backend to tell the debugger to continue. However, no further updates to the AST on the backend side are required.

4.4 Layouting



Listing 19: Source code to AST to Layout

This section describes the process of creating the flowchart layout required by D3.js. The result of this process is shown for an if-statement on the right side of Listing 19.

The source code is already transformed into the AST at the start of the process. For rendering it properly, this AST has to be extended by a few things.

4.4.1 Layout structure

Creating the additional information for the AST is done via a recursive algorithm.

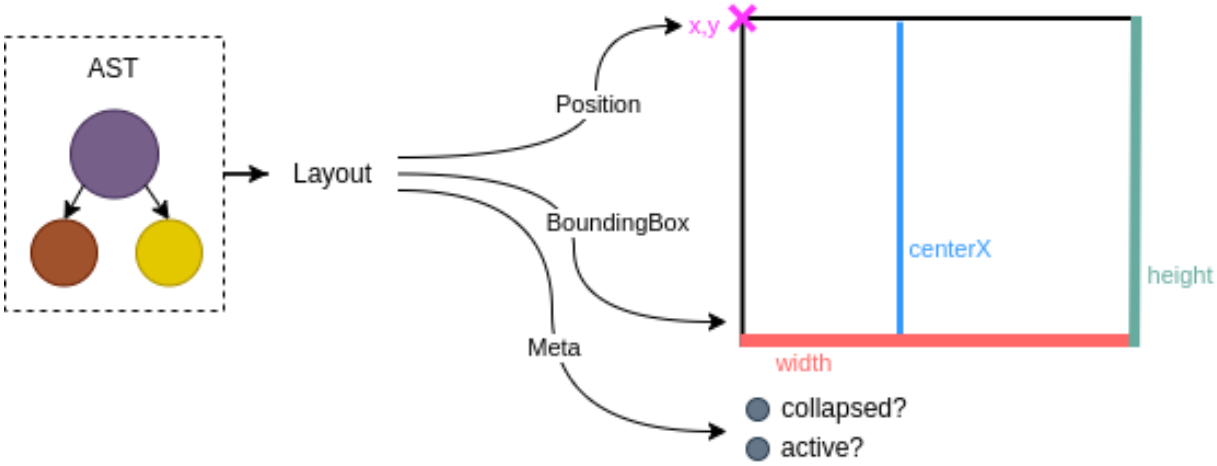


Figure 44: Source code to layout and metadata calculation process

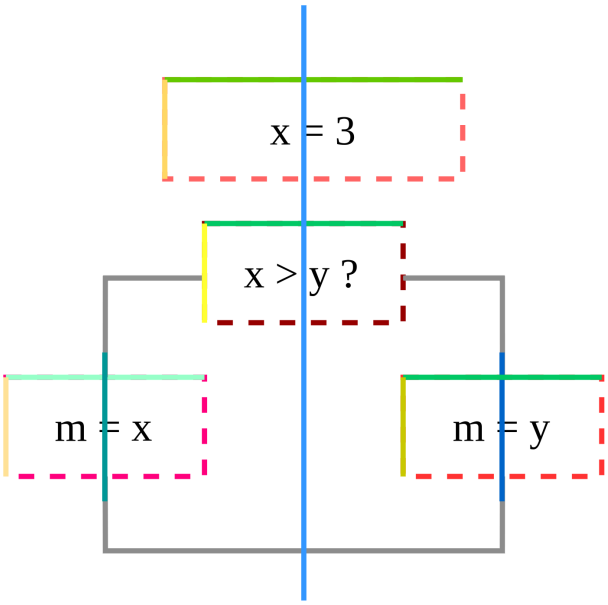


Figure 45: Source code to layout and metadata calculation process

Figure 44 shows that there are three different structures calculated for the layout and will be added to the AST:

- **BoundingBox**: A bounding box has a height, a width and a centre value. The centre of the horizontal axis is important for aligning the AstElement with other AstElements. Figure 45 shows how the BoundingBox is used to align the AST elements.
- **Position**: The layout stores the absolute x and y coordinates, starting from the top left. This is necessary for the global positioning of the AST elements, as shown in Figure 45.
- **Meta**: This consists of the boolean fields **`active`** and **`collapsed`**. **`active`** indicates that the debugger is currently stopped at this node. This is displayed with a light grey background. The **`collapsed`** flag is active when a node is collapsed, ignoring child nodes.

4.4.2 Layouting Algorithm

The algorithm extends the AST by bounding boxes, positions and metadata. This algorithm has to satisfy various conditions such as inline methods, collapsed nodes, the highlighted line and the screen position of the chart. To achieve this, the algorithm may calculate certain parts of the AST multiple times.

```
rootNode = getRootNode()
ast = createAstByMethod(rootNode)
collapsed = openCollapsedUntilHighlight(ast)
// recreate ast
ast = createAstByMethod(ast,collapsed)

layoutAst = createLayout(ast, { posX: 0 })
deltaCenterX = prevCenterX - layoutAst.box.centerX
// rerender layout AST
layoutAst = createLayout(ast, { posX: prevPosX + deltaCenterX })

prevCenterX = layoutAst.box.centerX
prevPosX = layoutAst.pos.x
```

Listing 20: Pseudocode of **createLayoutAstByMethod**

In Listing 20 the layout for the AST is created. First, the AST is selected from the ASTs sent. This **rootNode** is then transformed into a D3.js hierarchy. This hierarchy starts with the method node that matches the requested method name and node uuid. The next step is to open all collapsed nodes until the current highlighted line of code is found. Now the AST is recreated with collapsed nodes in consideration.

Now the layout is created as described later on in Listing 21. To avoid layout shifts caused by changes from for example collapsed nodes or inlined methods, a delta between the previous rendered layout and the new layout is calculated. Then the layout is recreated with the added delta width. The new x-position is saved for later renderings.

```
createLayout(astNodes, { posX }){
  astNodes.each(node => {
    node.box = getBoundingBox(node.data)
    node.pos = calculatePosition(node, { posX })
    node.meta = {
        collapsed: collapsed.has(node.data.uuid),
        active: hasActiveLine(activeLine, node, stackFrameMethods)
      }
  })
}
```

Listing 21: Pseudocode of **createLayout**

Listing 21 shows the process of adding bounding boxes, position, and metadata. The function **createLayout** traverses the AST from top to bottom. **getBoundingBox** is shown in detail in Listing 22. After calculating the bounding box, the position is computed.

The position of the nodes is absolute. The calculation starts with the parent node's top-left position as an anchor. This also has to be customized for all types of **AstElement**.

```
calculateBoundingBox (el: AstElement): BoundingBox {
  switch (el.kind) {
    // ...
    case 'Block': {
      childBoundingBoxes = el.statements.reduce((total, child)=>{
        return total + calculateBoundingBox(child)
      })
      return {
        width: childBoundingBoxes.width + PADDING,
        height: childBoundingBoxes.height + PADDING,
        centerX: childBoundingBoxes.centerX,
      }
    }
    // ...
  }
}
```

Listing 22: Pseudocode of **`calculateBoundingBox`**

The method **`getBoundingBox`** used in Listing 21 returns the cached result of **`calculateBoundingBox`**. Listing 22 shows how the recursive algorithm works by showing the calculation of the **`BoundingBox`** of a block. The aim is to combine the bounding boxes of the children and apply styling, such as padding, to the resulting bounding box. This calculation needs to be customized for all types of **`AstElement`**.

4.4.3 `redraw`

The **`redraw`** method is responsible for selecting the SVG in the DOM and then calling the **`renderChart`** function with all the necessary parameters. This contains the finished AST for the flowchart, toggle callbacks for collapsing and inlining methods and a **`FullWidthManager`**.

4.4.4 Collapse

The flowchart component stores the UUIDs of collapsed nodes as a set of strings. Changes to this set trigger the rerendering of the flowchart.

Some components have a double-click handler, which for one thing stops click propagation and prevents the default event handler, but it also calls the collapse function with the current UUID. The collapse function is in the Vue component and is passed through until it is needed.

Collapsed elements are automatically opened if they are in a direct line in the AST between the root and the highlighted statement. In this case, they cannot be closed.

Normally all elements are expanded except for catch-clauses. The set of collapsed elements has no history, so it forgets that the collapse state of a catch-clause has changed at any time. The set **`historicalCollapsedUuids`** contains all UUIDs of elements that have been collapsed at least once, and this tells JavaWiz which catch-clause should not be treated as a first-time collapse anymore.

4.4.5 Inline methods

Inlined methods are methods from the AST that are copied to other locations in the AST. In the Vue component, inlined methods are stored in **`inlinedMethods`**. This is a map that uses the method name as key and stores an object with the name **`methodAst`**, containing the AST element of type **`Method`** and **`uuids`**, a set of strings containing all method calls with that method as an open inlined method.

```
function openInlineInHightlightedLine(
  highlightedLine: number,
  ast: HierarchyNode<AstElement>,
  inlined: InlinedFnMap,
  stackFrameMethods: string[]
) {
  // find correct active statement (respect recursive method calls)
  let idx = 0
  const node = ast.find(node => {
    if (node.data.kind === 'Method' &&
          node.data.name === stackFrameMethods[idx]) {
      idx++
    }
    return node.data.begin === highlightedLine &&
           idx === stackFrameMethods.length &&
           node is (Conditional, Statement or IfStatement)
  })

  if (node) {
    // ...
    const meth = inlined.get(node.data.methodCallExpressions[...].name)
    // remove last inlined fn
    meth.uuids.delete(node.data.methodCallExpressions[...].uuid)
    // ...
    // open next method inlined
    meth.uuids.add(mc.uuid)
  }
}
```

Listing 23: Pseudocode of **`openInlineInHightlightedLine`**

The purpose of the **`openInlineInHightlightedLine`** method is to inline a method in the current highlighted statement when auto-inline is activated.

The pseudocode in Listing 23 shows the recursive lookup of the highlighted node in the first part of the function. To find the correct node, it is necessary to track the method order in the call stack, the highlighted line number, and the node type. The type is checked because only loops, if-statements, and statements contain inlinable method expressions.

If a node is found, the open inlined method is closed by removing its UUID from the open methods, and the node's UUID is added.

4.5 Visualization with D3.js

This chapter is about rendering with the D3.js library. It provides an overview for developers who maintain it or add new features.

In the next figure, the approach is illustrated by an example of an if-statement rendered with D3.js.

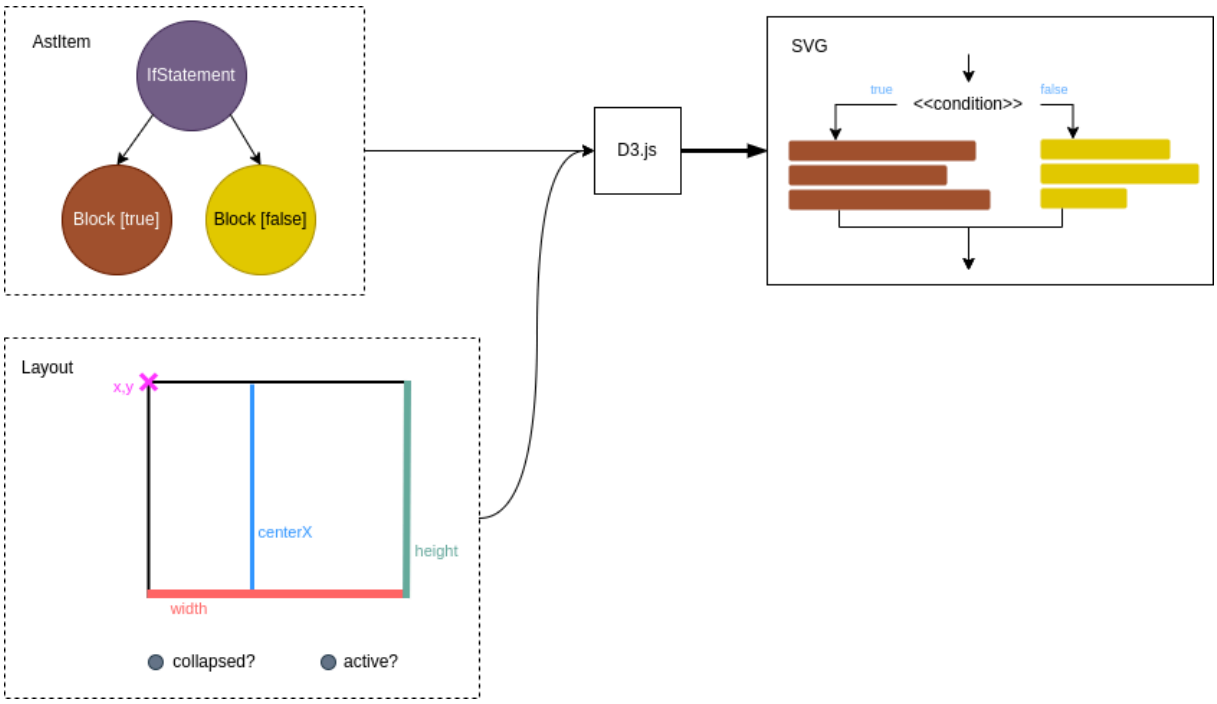


Figure 46: Render the data with D3.js as a SVG

The whole rendering process with D3.js starts in the **`renderChart`** method. As seen in Listing 24, each **`AstElement`** has its own **`join`** function, which is responsible for entering, updating and removing operations, including transitions.

```
renderChart (data: MetaHierarchyNode<AstElement>[]) {
  const methods = joinMethods(selection.selectAll('g.method'))
  const ifStatements = joinIfStatements(...)
  // ...

  d3.selectAll([methods, ifStatements, ...])
    .attr('id', (d: any) => uuidToDomId(d.data.uuid))
    .attr('translate', d => translate(d))
}
```

Listing 24: Pseudocode of **`renderChart`**

```
function joinIfStatements (selection, ...) {
  const g = selection.join(
    e => {
      const ifs = e.append('g').classed('if-statement', true)
      enterAnimation(ifs)
      return ifs
    }, u => u, e => exitAnimation(e))

  const conditions = g.selectAll('g.condition')
    .data(d => [d])
    .join(e => e.append('g').classed('condition', true)
      .on('dblclick', (ev: MouseEvent, d) => {
        collapse(d.data.uuid)
      }))
    .attr('transform', d => `translate(${d.box.centerX},0)`)

  renderRichText<IfStatement>(conditions, ... )


  // true-case lines
  // header to true case arrow
  addArrow(g, d => { ... }, 'true-top')

  // ... other arrows

  _addDebugInformation(g)
  return g
}
```

Listing 25: Pseudocode of **`joinIfStatements`**

The **`joinIfStatements`** function, shown as pseudocode in Listing 25, renders an if-statement similar to the one in Figure 46. The **`AstItem`** and layout metadata are provided to D3.js via **`MetaHierarchyNode<IfStatement>`**. The visualization is then created using D3.js via the select-data-join pattern. The visualization of child **`AstItems`** is not done in the parent visualization. For the if-statement in Figure 46, this means that the **`joinIfStatements`** function renders the entry and exit arrows of the blocks, the condition and the branch labels.

Spacing, proportions and other miscellaneous settings are stored in the **`ELEMENT`** object and used during rendering.

The **Animation** is done via the D3.js transitions. These transitions interpolate the selected attributes of a selection and can be adjusted in duration, delay and the easing function. The default animation duration for the JavaWiz flowchart is 1400ms, but some animations have a different duration, such as the exit of some selections at $\frac{1400\text{ms}}{5} = 280\text{ms}$.

Arrows indicate the direction of flow in JavaWiz flowcharts. These arrows have a chevron for the arrowhead and a solid line for the arrow tail, and, because the head and tail are connected, the arrow is a single SVG path.

An arrow can be generated with the **arrow** function. This function only returns a D3.js line if less than 2 points are provided. The last two points are used to calculate the points needed for the arrowhead. The head is the vector of these points rotated by 30 degrees and is 6 pixels long on each side.

In order to safely integrate the **arrow** function into the other flowchart renderings, the **addArrow** function is required. This function creates a path in the given selection and also provides a transition. This transition parses the current points of the path, if any, and interpolates them with the new points. It is important to note that the start and end points are not interpolated and are always the new points for a better-looking transition.

```
function zoomReset () {
  const svg = d3.select<SVGSVGElement, unknown>('#flowchart')
  const realWidth = svg.node()?.getBoundingClientRect().width || 0
  const x = -CENTER + realWidth / 2
  zoomBehavior.transform(svg, d3.zoomIdentity.translate(x, 0))
}

const zoomBehavior = d3.zoom<SVGSVGElement, unknown>()
  .scaleExtent([0.3, 3])
  .on('zoom', (e) => {
    d3.select('#flowchart g.chart')
      .attr('transform', e.transform)
  })

onMounted(() => {
  init()
  const chartRoot = d3.select<SVGSVGElement, unknown>('#flowchart')
  chartRoot.call(zoomBehavior)
    .on('dblclick.zoom', () => zoomReset())
})
```

Listing 26: Adding zoom to a D3.js SVG element

Zooming and panning is enabled by the D3.js zoom module as seen in Listing 26. It requires all the SVG content to be grouped together because the CSS style **transform** is applied to this group by this D3.js module.

By default, the D3.js zoom module zooms in on a double click, but this has been changed for the flowcharts to reset the zoom when the component is mounted. The **zoomReset** method must calculate the appropriate translation and scale, as the default zoom would position the content so that it is invisible most of the time.

For **visual debugging** the method **_addDebugInformation** is provided to find bugs in the rendered SVG. This method renders the **BoundingBox** of an **AstElement** in randomly chosen colours.

For elements such as if-statements and loops that need to check if the **block continues** to render the lines as seen in Section 3.2.1, the **blockContinues** function checks this recursively.

The function **renderRichText** is responsible for creating text that has clickable rectangles for opening inline methods and it also provides the width limit feature for the text.

4.6 Flowchart settings

The settings are stored in **autoInline** and **ELEMENT** and the component emits **update** after the settings have been changed. The **TheFlowChart** component redraws the flowchart whenever the settings are updated.

The settings are volatile and get lost after closing JavaWiz. It is possible to save the settings in a future version.

5 Summary and Conclusion

This thesis described the flowchart visualization component for JavaWiz. JavaWiz, a visual debugger designed to help students learn programming, can now display Java source code flowcharts. The flowchart component allows rendering a method as a flowchart, interactively stepping through the operations of the flowchart, collapsing and expanding nodes and inlining called methods at the call statement.

The thesis describes how to use JavaWiz with flowcharts, its architecture and gives an overview of its implementation. The flowchart program consists of several components: In the backend of JavaWiz, which is a Java system implemented in Kotlin, an AST is created; The frontend consists of a visualization system written in TypeScript and using the UI framework vue.js and the visualization library D3.js.

It can be said that the objective of incorporating flowcharts as an interactive aid for students has been achieved. The current version of JavaWiz, which is published as a Visual Studio Code plugin, already includes the flowcharts.

The flowcharts in JavaWiz have been used in the classroom since October 2023 to demonstrate the behaviour of, for example, recursive algorithms. Interacting with the flowchart seems fun for the user and makes the algorithm easier to understand. Students are supported in learning programming.

Currently, most Java control structures are supported by the JavaWiz flowchart, only some special elements of Java are not supported, e.g., switch and lambda expressions. In the future, more information may be added to the flowchart. This could be variable changes, stackframe information or previous inputs and outputs.

Bibliography

[1] [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:5807:ed-1:v1:en

[2] I. Nassi and B. Shneiderman, “Flowchart techniques for structured programming,” *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12–26, Aug. 1973, doi: 10.1145/953349.953350.

[3] G. Blaschek, “Computerunterstützter Programmentwurf mit Ablaufdiagrammen.” Dissertation, Johannes Kepler Universität Linz, 1987.

[4] A. Stritzinger, “Zeichnen von Ablaufdiagrammen für Modula-2-Programme.” Diplomarbeit, Johannes Kepler Universität Linz, 1985.

[5] O. M. G. (OMG), “Unified Modeling Language (UML) Specification,” Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML

[6] G. Blaschek, H. Mössenböck, and G. Pomberger, “Peter Rechenberg – Forscher, Lehrer, Mensch.” [Online]. Available: https://ssw.jku.at/Research/Papers/Moe03a/BlaMoePom03.pdf

[7] [Online]. Available: https://app.diagrams.net/

[8] J. Hamer, “A Lightweight Visualizer for Java,” 2004.

[9] [Online]. Available: https://github.com/atp-mipt/ljv

[10] [Online]. Available: https://github.com/nlfiedler/jswat

[11] M. Bostock, “D3.js - Data-Driven Documents.” Accessed: May 26, 2023. [Online]. Available: https://d3js.org/

[12] [Online]. Available: https://javaparser.org/

List of Figures