

A student-ready implementation of trace-based debugging for Java

Jolidon Bastien & Kappeler Kelvin
Supervisors: Pr. Pit-Claudiel Clément, Shardul Chiplunkar

Repository : PrintWizard

December 18, 2025

Abstract

Traditional debuggers, with their breakpoints and run-time inspection, are invaluable tools for software development. However, in complex scenarios, these tools can fail to provide a complete view of program execution. Tracing debuggers offer a unique approach by capturing and logging all execution events, allowing developers to interactively explore program behaviour post-runtime. This report presents the enhancement of *PrintWizard*[Ser24b], a tracing debugger prototype for Java originally developed as part of Erwan Serandour’s master’s thesis[Ser24a].

The project focuses on improving the usability and accessibility of the debugger’s web interface. Key advancements include collapsible event blocks, a consistent visual design, and enhanced interactivity to navigate traces and inspect objects’ states.

By addressing usability challenges and streamlining the debugging workflow, the project transforms *PrintWizard* into a practical and effective debugging tool. The enhanced interface is designed for students and novice developers, making it easier to debug Java applications. This work contributes to the broader effort to create developer-friendly tools to understand program execution and solve software issues efficiently.

Contents

1	Introduction	3
1.1	Timeline	5
2	Interface Presentation	7
2.1	Design Principles	7
2.2	Features and Functionalities	8
2.2.1	Trace Features	8
2.2.2	Inspector Features	9
3	Debugging Workflow	10
4	Future Works	12
4.1	Setup	12
4.2	Backend	12
4.3	Frontend	12
5	Conclusion	14
6	Appendix	15
6.1	Original proposal	15
6.1.1	Introduction	15
6.1.2	Current State	15
6.1.3	Proposed Solution	16
6.1.4	Timeline	16
6.2	Boids Example	17

1 Introduction

Developers make mistakes. One way to fix bugs in a program is to read its code. Unfortunately, this can quickly become complicated, or even impossible, as the code becomes more complex. This is why a large majority of developers use two different methods to solve their problem:

- The first is called logging and allows displaying and collect a record of events during the execution of the program. Although this method can be excellent (and especially very fast) to know certain values of a particular method for example, it is absolutely no longer considered when this method can be called several hundred times per second. Logging floods the user with information, who must therefore resolve to use the second option.
- Using a traditional debugger[Ros96] allows you to pause the execution of a program at a desired location (called breakpoints) in order to know its state at that moment. This type of tool is very effective when the user knows where the error is, and is much less effective if they have to click a hundred times to resume the execution of the program until they find the bug.

These two tools each have their flaws and their qualities. Not happy with the possibilities offered by these, Erwan Serandour implemented the beginnings of a new type of tool called tracing debugger[JLL23]. A tracing debugger allows you to explore the entire execution of a code in order to be able to display or search for the information required to correctly resolve the bug. This tracing debugger is called *PrintWizard*. The initial version of *PrintWizard* provides a foundational framework to inspect traces of program execution through a web-based interface. This tool enables developers to explore the trace interactively by offering key features, including:

- Displaying the complete execution trace of the program.
- Allowing users to click on statements to expand them and inspect the operations executed inside.
- Showing the return values of methods and the values of variables at different points in the trace.
- Enabling object inspection by clicking on objects, which would open a dedicated information box.

Although the prototype contains the essential functionality required for trace inspection, it has significant limitations that hampered its use. The main problem is the lack of interactivity with the trace. Data is often displayed inconsistently, and clicking on an object to retrieve details causes information to appear in seemingly random places, making it difficult for users to effectively follow the debugging process.

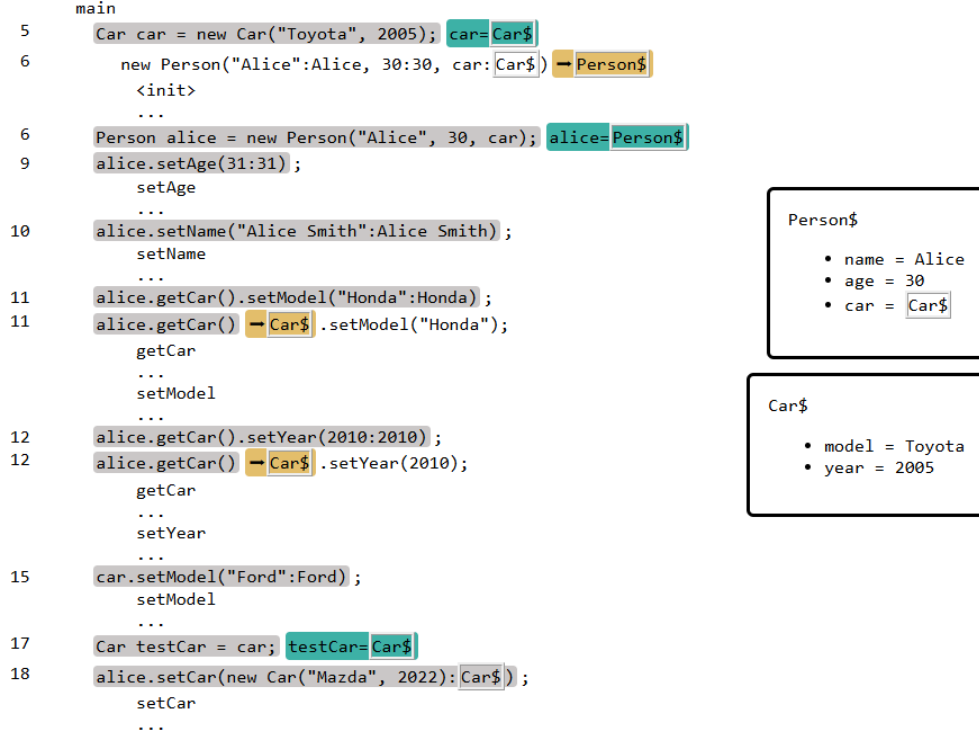


Figure 1: Initial version of *PrintWizard*

The motivation for improving the tool was to transform it into a fully user-friendly solution. The goal is to design a clear and intuitive interface that consistently presents information, eliminating confusion caused by the previous layout. Additionally, new frontend features, such as a search functionality within the trace, are planned to further enhance the debugging process and make the tool accessible to a broader audience, including novice developers and students.

The architecture of the initial tracing debugger, *PrintWizard*, is designed to capture comprehensive information about program execution and provide an interactive interface for exploring the resulting trace. The system is divided into two main components: the backend, implemented in Java, and the frontend, developed in HTML/CSS/JavaScript. This separation facilitates modularity and allows for independent enhancements of each component.

To further organize the system, the tool is decomposed into five functional modules:

- **Instrumentation Module:** Responsible for injecting logging instructions into the target program during compilation. This is achieved using a Java compiler plugin, which modifies the program’s abstract syntax tree (AST) to include logging capabilities. This approach ensures the preservation of detailed information that would otherwise be lost in the bytecode.
- **Logging Module:** Records program events and stores them in a structured event log. Events are organized hierarchically, with group events marked by start and end indicators, and execution steps represented as discrete entries. The logs are stored in JSON format, enabling easy access and interoperability with the frontend.
- **Object Data Store Module:** Manages the storage and retrieval of object states throughout the program’s execution. Each object state is identified using a combination of its reference

hash code and timestamp, allowing the debugger to retrieve the correct version of an object at any point in the trace.

- **Source Code Formatting Module:** Simplifies the execution trace by linking each event to its corresponding source code expression. This reduces redundancy in the logs and enables a more intuitive mapping between the code and its execution behaviour.
- **Display Module:** Serves as the bridge between the backend and the user, rendering the execution trace in the graphical user interface.

This modular design ensures flexibility and allows the tool to provide an interactive and detailed debugging experience. The backend generates three JSON files: *EventTrace.json*, containing the logged execution events; *ObjectData.json*, holding object states; and *SourceFormat.json*, linking events to their source code. These files are used by the Display Module to render the execution trace in the graphical user interface. However, the Display Module required significant enhancements to improve usability and clarity. Challenges included inconsistencies in the presentation of data, a lack of seamless interaction for navigating traces, and limited visual organization. Our efforts in this project primarily focused on addressing these issues by refining the Display Module to offer a more intuitive and user-friendly interface for debugging so that it can be used by second-year EPFL IC Bachelor students during their *Software Construction*[EPF24] course.

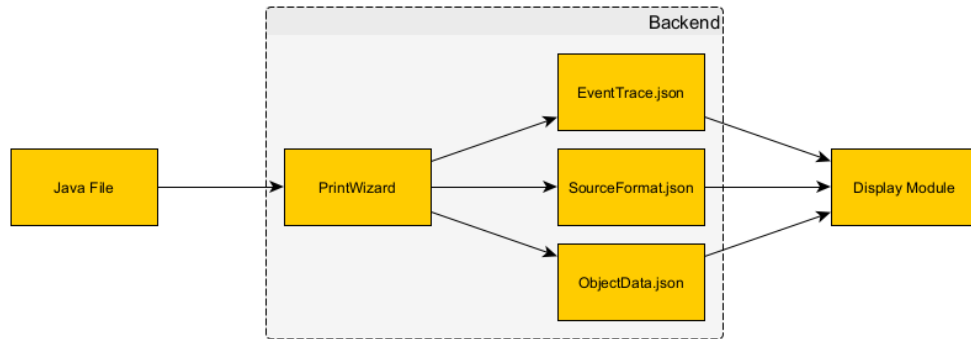


Figure 2: Short view of the architecture of *PrintWizard*

1.1 Timeline

The timeline 6.1.4 for the project was originally divided into three main phases:

1. **Familiarization and Planning (Weeks 1–3):** The initial phase focused on understanding the existing version of PrintWizard and brainstorming the features to implement. This involved analysing the tool’s architecture and identifying areas for improvement.
2. **Feature Implementation (Weeks 4–11):** The second phase was dedicated to implementing the selected features. This eight-week block was aimed at significantly enhancing the interface’s usability and functionality.
3. **Finalization and Documentation (Weeks 12–14):** The final phase involved refining the tool, fixing backend problems, and preparing comprehensive documentation, including the report and user guides.

While the timeline was generally respected, feature implementation extended slightly beyond the planned eight weeks due to the complexity of some enhancements and our lack of experience in web development. Unfortunately, due to time constraints, we were unable to improve the backend as initially intended, prioritizing the completion of frontend features instead.

2 Interface Presentation

The web interface of the tracing debugger was developed to provide users with an intuitive and efficient way to explore program execution traces. The interface acts as a bridge between the backend-generated trace data and the user, presenting complex debugging information in an organized and interactive manner. Key enhancements were made to improve clarity, usability, and interactivity, ensuring that the tool supports users at various skill levels.

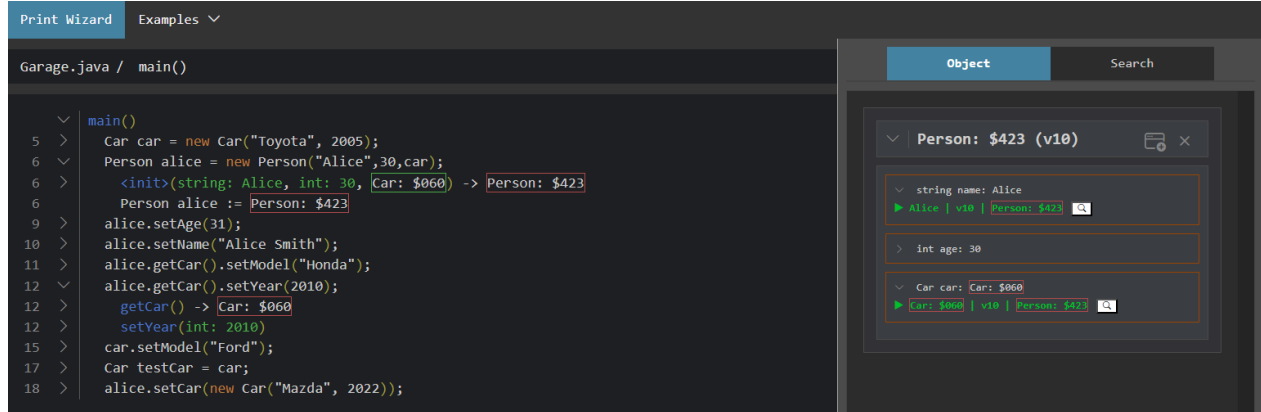


Figure 3: The new *PrintWizard* interface

2.1 Design Principles

The design of the web interface focuses on dividing its functionality into distinct, interconnected components to simplify debugging. To achieve this, the interface was structured around three main components:

- **Trace:** Designed to provide a hierarchical representation of program execution, allowing users to explore events at various levels of detail.
- **Inspector:** Created to offer a focused view of detailed information about objects while maintaining the trace as a high-level overview.
- **Toolbar:** Developed for easy access to essential controls such as resetting the interface or switching traces.

These components were designed to work seamlessly together, ensuring clarity and usability. Features such as collapsible trace elements, interactive object inspection, and search functionality were included to streamline navigation and enhance user experience.

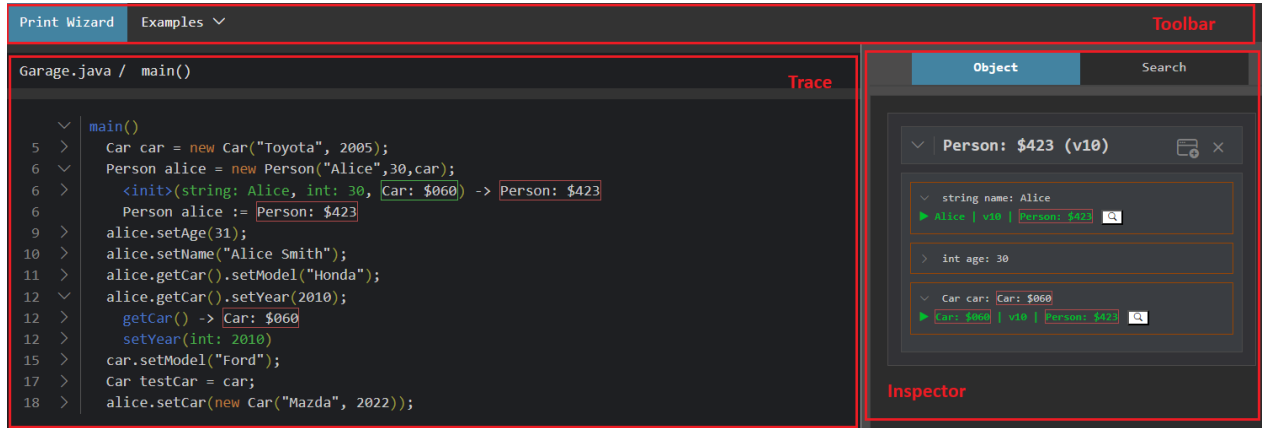


Figure 4: The three main components of the interface

2.2 Features and Functionalities

The web interface of the tracing debugger offers a variety of features designed to enhance the debugging experience. These functionalities are grouped into two main areas: the trace and the inspector.

2.2.1 Trace Features

The trace is the core component of the interface, presenting the program's execution as a hierarchical tree. It combines interactive elements with visual enhancements to provide a comprehensive view of the program's behavior:

- **Expandable Statements and Substatements:** Users can click on a disclosure triangle icon to expand or collapse statements and substatements, revealing detailed information about their execution. By holding **Shift** while clicking on a disclosure triangle, users can also expand or collapse a statement along with all its child elements, enabling the user to navigate the trace at different levels of granularity.
- **Value Highlighting:** Return values are displayed in red, while argument values are shown in green, providing a clear visual distinction between input and output data.
- **Object Pointers:** Pointers to objects are displayed within a bordered box (green for arguments, red for return values). These pointers are clickable, allowing users to open the corresponding object version in the inspector for detailed exploration.
- **Loop Handling:** To address the verbosity of loops with many iterations in the initial version, loops structure have been modified to allow hiding iterations. Users can expand a loop statement to selectively extend the iteration of interest. Iteration-specific details, such as condition evaluation and incrementation, are displayed comprehensively, with iteration numbers clearly marked in the trace.

```

4  ✓   for (int i = 0; i < 5)
4      int i := int: 0
4  ✓   i0 for (int i = 0; i < 5) -> i < 5 -> bool: true
5  >   ++compteur;
4      int i := int: 1
4  >   i1 for (int i = 0; i < 5) -> i < 5 -> bool: true
4      int i := int: 2
4  >   i2 for (int i = 0; i < 5) -> i < 5 -> bool: true
4      int i := int: 3
4  >   i3 for (int i = 0; i < 5) -> i < 5 -> bool: true
4      int i := int: 4
4  >   i4 for (int i = 0; i < 5) -> i < 5 -> bool: true
4      int i := int: 5
4      i5 for (int i = 0; i < 5) -> i < 5 -> bool: false

```

Figure 5: The trace of a for loop

2.2.2 Inspector Features

The inspector provides detailed views of objects and allows users to search within the trace. Its key functionalities include:

- **Object Inspection:**

- The *Object Tab* is dedicated to inspecting object fields and their history during program execution.
- Multiple objects can be opened simultaneously, enabling side-by-side comparisons.
- Each object inspection box includes:
 - * A header displaying the type, pointer, and version of the object.
 - * The object's field values and a history of each field's changes throughout execution.
 - * Buttons that allow users to jump directly to specific versions of the object in the trace.
- Object inspection boxes can be detached from the inspector, providing flexibility, and can be closed when no longer needed.

- **Search Functionality:**

- The *Search Tab* allows users to execute commands and search for specific values or patterns in the trace.
- Available commands include:
 - * **help**: Displays a list of available commands.
 - * **sf**: Searches for specific fields or values.
 - * **value**: Finds occurrences of a particular value in the trace.

These features collectively make the interface a powerful and flexible tool for debugging, enabling users to interactively explore traces, inspect objects in detail, and efficiently search for relevant information.

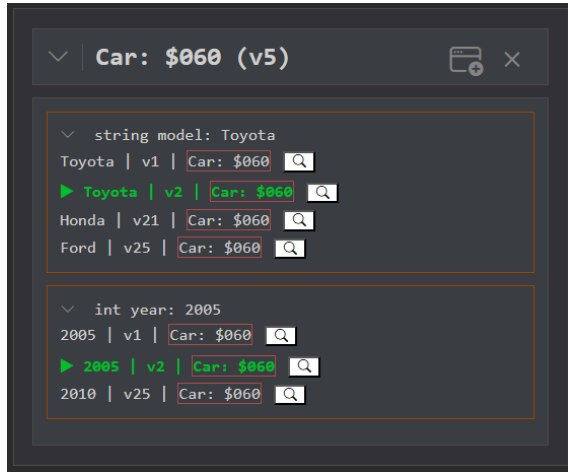


Figure 6: Inspection panel of an object

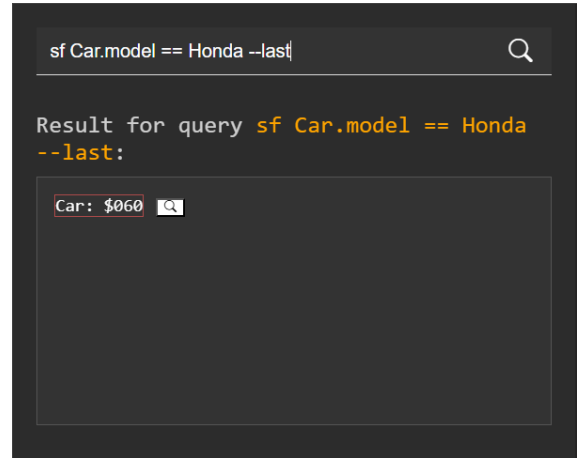


Figure 7: The search inspector

3 Debugging Workflow

PrintWizard becomes very good when you have to debug programs where the user does not know where the error is. An example from *PrintWizard* is based on a simulation of a flock of boids whose code is in the appendix 6.2, inspired by a simulation of Craig Reynolds[Rey87]. Each Boid has a velocity and a position that will be updated at each tick of the simulation, depending on the other Boids. After starting the program, it turns out that the position of some Boids returns a value of *NaN* instead of a number. As the `tick` method is called often (and there are 1888 different Boids!), it is very difficult with a traditional method to find the error. Thanks to the new *PrintWizard* web interface, it is much easier to find it by following this procedure:

- **Bug Identification** : By going to search and writing values `Vector2.x`, we can display all the different values that the `x` field takes for all `Vector2` objects. We notice that it sometimes takes the *NaN* value.

```
756.0968, 757.7905, 763.23254, 766.8296, 772.11304,
773.754, 777.57825, 786.58875, 788.3142, 799.0387,
8.265847, 8.54599, 8.659088, 8.74456, 801.1891, 809.75287,
815.91455, 820.458, 830.7658, 835.5066, 845.42505,
845.7431, 855.2281, 864.91406, 874.581, 891.54254,
9.631592, 908.87024, 950.70557, 961.75793, 976.9952,
990.167, 992.4153, 999.3043, NaN
```

Figure 8: One value taken by a `Vector2` is *NaN*

- **Finding the source of the bug** : It is possible using the command `sf Vector2.x == NaN --first` to find the first interaction in the code where there is a *NaN* value for the `x` field of a `Vector2`. The command tells the user that the vector `$038` is the first *NaN* in the program. Through the interface, it is possible to see that this is the return value of the `totalForce` method.

```

165 > tickBoid(Boid: $020, ArrayList: $526, Physics: $690) -> Boid: $674
149 > Geometry.Vector2 acceleration = totalForce(thisBoid, allBoids, physics);
149 > totalForce(Boid: $020, ArrayList: $526, Physics: $690) -> Vector2: $038

```

Figure 9: Location of the first *NaN*

- **Manual investigation** : After going through the first lines of the `totalForce` method, it turns out that the first *NaN* is at the level of the calculation of the position of the centre in the `cohesionForce` method. Indeed, a division by 0 is present when there are no Boids in the perception radius of a Boid!

```

totalForce(Boid: $020, ArrayList: $526, Physics: $690) -> Vector2: $038
List<Boid> withinPerceptionRadius = boidsWithinRadius(thisBoid, allBoids.stream(), physics);
Geometry.Vector2 cohere = cohesionForce(thisBoid, withinPerceptionRadius);
cohesionForce(Boid: $020, ListN: $604) -> Vector2: $003
    Geometry.Vector2 sum = Geometry.Vector2.zero();
    for(int i=0; i<boidsWithinPerceptionRadius.size(); ++i)
    Geometry.Vector2 center = sum.scale(1 / (float) boidsWithinPerceptionRadius.size());
        boidsWithinPerceptionRadius.size()() -> int: 0
        1 / (float)boidsWithinPerceptionRadius.size() -> float: NaN
        scale(float: NaN) -> Vector2: $086
    Vector2 center := Vector2: $086
    return center.minus(thisBoid.position());

```

Figure 10: Screenshot of the bug

Using this example, it is therefore possible to debug a code much more easily than with a traditional method.

4 Future Works

There are three main areas where *PrintWizard* can be improved: the setup, the backend, and the frontend. This project is not about improving the setup or the backend, but here are the different possible improvements (these three lists are of course not exhaustive) :

4.1 Setup

- **Simplifying the setup** : Currently, the setup to start *PrintWizard* is complex and long: you have to start two different scripts (that you have to modify manually for each program) and twice a traditional debugger in order to then start the web server to access the data. One possibility would be to transform all this into a single script with a single command.

4.2 Backend

- **Standardization of .json files** : Currently, .json files (mainly the source code one) change drastically when a statement is written on multiple lines. For example, some loops do not contain all the information at the line level. Additionally, some information could be added about the content of the structure (loop, condition, functions..) in the eventTrace.json because it was complex to translate the 3 .json into a uniform structure for the web interface.
- **Increased in the number of possible syntaxes** : Some syntaxes in Java are not possible using *PrintWizard*, because it crashes the debugger. The most annoying example is `i++` (while `++i` works just fine).
- **Access to mutable data** : It is currently not possible to see all states of a mutable object (nothing changes). It would be nice to be able to use *PrintWizard* on mutable code!

4.3 Frontend

- **Enhancing trace interactions with query language integration** : Currently, the implemented search works as a command system, but it would also be possible to improve the usability of *PrintWizard* by allowing all interactions with the trace to be expressed as queries where users can seamlessly transition between exploring trace data and forming precise queries. For instance, inspecting an object in the Object Inspector would automatically fill the search bar with a corresponding query, such as *inspect object \$123*. Conversely, entering a query like *sf Vector2.x == 10* would directly highlight matching instances in the trace.

We imagined a CSS-like query language for its simplicity and readability. Similar to CSS selectors, queries could target specific elements of the trace based on attributes. For example: *#totalForce* to locate all invocations of the *totalForce* method, or *Vector2.x[value=NaN]* to identify variables with a specific value. This bidirectional link between the query bar and the trace would ensure that users can explore and manipulate traces intuitively and efficiently.

- **Code refactoring** : The project code could use some improvement to make it more robust and have less repetition. This was the first HTML/CSS/Javascript project we implemented, so it could use some code quality improvement.
- **Increase in the number of search commands** : Only two commands exist at the moment: *sf* and *values*. A greater diversity of commands would be nice, for example with a command allowing to directly access the *i*-th iteration of a loop.

- **Implementation of other inspectors** : An inspector that can display the state of a loop or function would provide a greater amount of interesting information.
- **Settings** : Creating a window to change debugger settings would allow for more appropriate customization for different users. For example, changing the colours or display of certain components (such as loop iterations).
- **Optimization** : Currently, the Boids example takes 3–4 seconds to start because the different .json files are over 150,000 lines long. Better code optimization would drastically reduce this number.

5 Conclusion

Through this project, it was possible to change the entire frontend of *PrintWizard* in order to be able to navigate and debug in a simpler and more consistent way than with the old interface. The addition of new features such as a search with integrated commands, or being able to visualize the different states of an object make *PrintWizard* a powerful tool to help developers debug their code. An improvement of the setup would be appreciated in order to have a truly easy-to-use software. Finally, through this project, we have shown that it is possible to solve a bug easily in a complex code simply by inspecting the execution of the program.

6 Appendix

6.1 Original proposal

6.1.1 Introduction

Debugging is a critical aspect of software development, particularly when dealing with complex systems. Tools that facilitate the understanding of program execution are indispensable for developers. Unlike a classic debugger which stops the execution of the program at a given moment, a tracing debugger is an uncommon plugin that logs every event during the execution of a Java program, allowing users to explore them interactively after runtime. Originally developed as part of Erwan's master thesis, *PrintWizard* is a tracing debugger prototype enabling inspection of the trace through a web-based interface.

Currently, the prototype is operational but requires some modifications to make the user experience more pleasant. The goal of this project is to enhance the existing web interface, making the tool more intuitive, interactive, and practical for developers to use in debugging Java applications. By improving the user interface and overall experience, we aim to turn this plugin into a useful debugging tool.

6.1.2 Current State

In order to be able to analyse the trace in the web interface, *PrintWizard* successfully outputs 3 different .json files:

- **eventTrace.json** : Contains all the trace events that were logged during the execution of the program
- **objectData.json** : Contains all the states of the objects that were created during the execution of the program.
- **source_format.json** : Contains the source code of the program.

In order to create these 3 files, the prototype is unfortunately rather unintuitive and requires manual configuration for each different files. As a result, it is challenging to get started with the debugger. However, the data backend works very well with one exception: the debugger does not know when a method in an API (like the standard one) modifies an object. The biggest limitations of the prototype come from the web interface:

- **Limited interactivity** : While the plugin allows inspection of variables and objects, navigating through events is cumbersome. All statements are expandable by default, and it is not possible to unexpand a loop or a condition statement in order to have more visibility. It is also not possible to search for a predefined state.
- **Poor design** : Currently, it is difficult for users to understand information of the trace. There is a lot of repeated information and colours, which can be confusing for the user. Furthermore, it is not possible to know if something is clickable or not.

The primary objective now is to address these issues and build upon this foundational work to create a developer-friendly tool.

6.1.3 Proposed Solution

To improve the usability of the debugger, the proposed solution involves the following key enhancements:

- **Collapsible Event Blocks** : Allow users to collapse and expand events, especially for loops and conditionals, providing better control over the level of granularity displayed.
- **Visual Differentiation** : Introduce a consistent colour scheme and visual cues to help users distinguish between different event types, object states, and program statements.
- **Clickable Interactions** : Clearly highlight interactive elements like buttons or expandable sections to eliminate confusion about what parts of the interface are actionable.
- **Backend Improvements** : Implement a way to know when a method in an API modifies an object. This will improve the accuracy of the trace and provide more detailed information to the user.
- **Setup Process** : If time allows, simplifying setup process with clear instructions and automated configuration where possible. This will make it easier for users to install and use the tool. It should be easy to use the tool for first-year EPFL students.

By improving these aspects, the tool will become an integral part of the Java debugging workflow, saving developers time and effort in tracking down and resolving bugs.

6.1.4 Timeline

The development process will involve the following key tasks:

- **Week 1-2** :
 - Analyse and understand the existing codebase and identify areas for improvement.
 - Write the project proposal.
- **Week 3** :
 - Decide on a list of features that need to be implemented and prioritize them.
 - Think about how to use them to make the user experience optimal. Imagine several concepts in order to choose the best one.
- **Week 4-11** :
 - Implementations of the features decided in week 3 given their priority. This will include the following tasks:
 - * Implement a way to choose the level of granularity displayed.
 - * Implement a way to know what can be interacted with or not.
 - * Implement functionality to search for a predefined state.
 - * Improve the object inspection interface.
 - * Implement a consistent colour scheme and visual cues to help users distinguish between different event types, object states, and program statements.
 - * Other features that may be added depending on the progress of the project.

- **Week 12-13 :**

- Improve the backend to know when an object is modified in an external API.
- Write documentation for the tool.
- Minor modifications of the UI to make it more user-friendly.
- If time allows, simplify the setup process with clear instructions and automated configuration.

- **Week 14 :**

- Finalize the project, write the final report and prepare the presentation.

6.2 Boids Example

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.util.ArrayList;
import java.awt.event.WindowEvent;
import java.util.List;
import java.util.Random;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) throws InterruptedException {

        List<Boid> boids = Stream
            .generate(World::createRandomBoid)
            .limit(10)
            .toList();

        int x = 0;
        while(++x<12){
            boids = BoidLogic.tickWorld(boids, World.physics);
        }

    }

    static class Canvas extends Frame {
        static final int BOID_SIZE = 3;
        List<Boid> boids = Stream
            .generate(World::createRandomBoid)
            .limit(10)
            .toList();

        public Canvas()
        {
            setVisible(true);
            setSize(World.Physics.WIDTH, World.Physics.HEIGHT);
            addWindowListener(new WindowAdapter() {
                @Override
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            })
        }
    }
}
```

```

    });
}
public void paint(Graphics g)
{
    if(boids==null){
        return;
    }
    boids = BoidLogic.tickWorld(boids, World.physics);
    g.setColor(Color.RED);
    for(Boid b : boids){
        Geometry.Vector2 pos = b.position();
        Geometry.Vector2 direction = b.velocity().normalized().scale(BOID_SIZE);
        Geometry.Vector2 leftPoint = pos.add(direction.orthogonal());
        Geometry.Vector2 upperPoint = pos.add(direction.scale(3));
        Geometry.Vector2 rightPoint = pos.minus(direction.orthogonal());
        int[] xPoints = new int[]{(int) leftPoint.x(), (int) upperPoint.x(), (int)
            rightPoint.x()};
        int[] yPoints = new int[]{(int) leftPoint.y(), (int) upperPoint.y(), (int)
            rightPoint.y()};
        g.fillPolygon(xPoints, yPoints, 3);
    }
}

public record Boid(
    Geometry.Vector2 position,
    Geometry.Vector2 velocity) {

}

static public class BoidLogic {
    static final float EPSILON = 0.001F;

    static Stream<Boid> boidsWithinRadius(Boid thisBoid, Stream<Boid> boids, float
        radius) {
        Geometry.Vector2 pos = thisBoid.position();
        return boids
            .filter(b -> pos.distanceTo(b.position()) < radius &&
                !b.equals(thisBoid));
    }

    static Geometry.Vector2 avoidanceForce(Boid thisBoid, Stream<Boid>
        boidsWithinAvoidanceRadius) {
        Geometry.Vector2 pos = thisBoid.position();
        return boidsWithinAvoidanceRadius
            .map(b -> {
                Geometry.Vector2 bToThisBoid = pos.minus(b.position());
                if (bToThisBoid.norm() < EPSILON) {
                    return Geometry.Vector2.zero();
                } else {
                    return bToThisBoid.scale(1 / bToThisBoid.squaredNorm());
                }
            })
            .reduce(Geometry.Vector2.zero(), Geometry.Vector2::add);
    }
}

```

```

static Geometry.Vector2 cohesionForce(Boid thisBoid, List<Boid>
    boidsWithinPerceptionRadius) {
    Geometry.Vector2 sum = Geometry.Vector2.zero();
    for(int i=0; i<boidsWithinPerceptionRadius.size(); ++i){
        sum = sum.add(boidsWithinPerceptionRadius.get(i).position);
    }
    Geometry.Vector2 center = sum.scale(1 / (float)
        boidsWithinPerceptionRadius.size());
    return center.minus(thisBoid.position());
}

static Geometry.Vector2 alignmentForce(Boid thisBoid, List<Boid>
    boidsWithinPerceptionRadius) {
    if (boidsWithinPerceptionRadius.isEmpty()) {
        return Geometry.Vector2.zero();
    } else {
        Geometry.Vector2 meanVelocity = boidsWithinPerceptionRadius.stream()
            .map(Boid::velocity)
            .reduce(Geometry.Vector2.zero(), Geometry.Vector2::add).scale(1 /
                (float) boidsWithinPerceptionRadius.size());
        return meanVelocity.minus(thisBoid.velocity());
    }
}

static Geometry.Vector2 containmentForce(Boid thisBoid, List<Boid> allBoids, int
    width, int height) {
    Geometry.Vector2 pos = thisBoid.position();
    if (pos.x() < 0 && pos.y() < 0) {
        return new Geometry.Vector2(1, 1);
    } else if (pos.x() < 0 && pos.y() > height) {
        return new Geometry.Vector2(1, -1);
    } else if (pos.x() > width && pos.y() < 0) {
        return new Geometry.Vector2(-1, 1);
    } else if (pos.x() > width && pos.y() > height) {
        return new Geometry.Vector2(-1, -1);
    } else if (pos.x() < 0) {
        return new Geometry.Vector2(1, 0);
    } else if (pos.x() > width) {
        return new Geometry.Vector2(-1, 0);
    } else if (pos.y() < 0) {
        return new Geometry.Vector2(0, 1);
    } else if (pos.y() > height) {
        return new Geometry.Vector2(0, -1);
    } else {
        return Geometry.Vector2.zero();
    }
}

static Geometry.Vector2 totalForce(Boid thisBoid, List<Boid> allBoids,
    World.Physics physics) {
    List<Boid> withinPerceptionRadius = boidsWithinRadius(thisBoid,
        allBoids.stream(), physics.perceptionRadius()).toList();
    Geometry.Vector2 cohere = cohesionForce(thisBoid, withinPerceptionRadius);

```

```

        Geometry.Vector2 align = alignmentForce(thisBoid, withinPerceptionRadius);
        Stream<Boid> withinAvoidanceRadius = boidsWithinRadius(thisBoid,
            withinPerceptionRadius.stream(), physics.avoidanceRadius());
        Geometry.Vector2 avoid = avoidanceForce(thisBoid, withinAvoidanceRadius);
        Geometry.Vector2 contain = containmentForce(thisBoid, allBoids,
            World.Physics.WIDTH, World.Physics.HEIGHT);

        return avoid.scale(physics.avoidanceWeight())
            .add(cohere.scale(physics.cohesionWeight()))
            .add(align.scale(physics.alignmentWeight()))
            .add(contain.scale(physics.containmentWeight()));
    }

    static Boid tickBoid(Boid thisBoid, List<Boid> allBoids, World.Physics physics) {
        Geometry.Vector2 acceleration = totalForce(thisBoid, allBoids, physics);
        Geometry.Vector2 velocity = thisBoid.velocity().add(acceleration);
        if (velocity.norm() > physics.maximumSpeed()) {
            velocity = velocity.normalized().scale(physics.maximumSpeed());
        }

        if (velocity.norm() < physics.minimumSpeed()) {
            velocity = velocity.normalized().scale(physics.minimumSpeed());
        }

        return new Boid(thisBoid.position().add(thisBoid.velocity()), velocity);
    }

    public static List<Boid> tickWorld(List<Boid> allBoids, World.Physics physics) {
        List<Boid> newBoids = new ArrayList<>(10);
        for(int i=0; i<allBoids.size(); ++i){
            newBoids.add(tickBoid(allBoids.get(i), allBoids, physics));
        }
        return new ArrayList<>(newBoids);
    }
}

static public class World {
    static Random rand = new Random(4);
    static World.Physics physics = new World.Physics(
        2f,
        8f,
        80f,
        15f,
        1f,
        0.001f,
        0.027f,
        0.5f);

    static Boid createRandomBoid() {
        float x = randomFromRange((float) Physics.WIDTH /3, (float) (Physics.WIDTH *
            2) /3);
        float y = randomFromRange((float) Physics.HEIGHT /3, (float)
            Physics.HEIGHT*2/3);
    }
}

```

```

        float rotation = randomFromRange(0, (float) (2 * Math.PI));
        float initialSpeed = randomFromRange(physics.maximumSpeed(),
            physics.maximumSpeed());
        Geometry.Vector2 initialVelocity =
            Geometry.Vector2.UnitUp().rotate(rotation).scale(initialSpeed);
        return new Boid(new Geometry.Vector2(x, y), initialVelocity);
    }

    static float randomFromRange(float start, float end) {
        return rand.nextFloat() * (end - start) + start;
    }

    public record Physics(
        float minimumSpeed,
        float maximumSpeed,
        float perceptionRadius,
        float avoidanceRadius,
        float avoidanceWeight,
        float cohesionWeight,
        float alignmentWeight,
        float containmentWeight
    ) {
        public static final int WIDTH = 1000;
        public static final int HEIGHT = 700;
    }
}

static public class Geometry {

    public record Vector2(float x, float y){

        public static Geometry.Vector2 zero(){return new Geometry.Vector2(0, 0);}
        public static Geometry.Vector2 UnitUp(){
            return new Geometry.Vector2(0, -1);
        }

        public Geometry.Vector2 rotate(float radians) {
            return new Geometry.Vector2(
                (float) (Math.cos(radians)*x - Math.sin(radians)*y),
                (float) (Math.sin(radians) * x + Math.cos(radians) * y)
            );
        }

        public float squaredNorm(){
            return this.x*this.x+this.y*this.y;
        }

        public float norm(){
            return (float) Math.sqrt(squaredNorm());
        }

        public Geometry.Vector2 normalized(){
            if (norm()==0){
                return zero();
            }
        }
    }
}

```

```

        }else {
            return this.scale(1/norm());
        }
    }

    public Geometry.Vector2 orthogonal(){
        return new Geometry.Vector2(-this.y, this.x);
    }

    public float squaredDistanceTo(Geometry.Vector2 that){
        float dx = this.x - that.x;
        float dy = this.y - that.y;
        return dx*dx+dy*dy;
    }

    public Geometry.Vector2 minus(Geometry.Vector2 that){
        return new Geometry.Vector2(this.x-that.x, this.y-that.y);
    }

    public Geometry.Vector2 add(Geometry.Vector2 that){
        return new Geometry.Vector2(this.x+that.x, this.y+that.y);
    }

    public float distanceTo(Geometry.Vector2 that){
        return (float) Math.sqrt(squaredDistanceTo(that));
    }

    public Geometry.Vector2 scale(float scale){
        return new Geometry.Vector2(scale*x, scale*y);
    }
}
}
}

```

References

- [EPF24] EPFL. Software construction course. <https://edu.epfl.ch/studyplan/en/bachelor/computer-science/coursebook/software-construction-CS-214>, 2024. Accessed: 2024-12-20.
- [JLL23] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. Open tracing tools: Overview and critical comparison. *Journal of Systems and Software*, 204:111793, 2023.
- [Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. <https://www.red3d.com/cwr/papers/1987/boids.html>, 1987. Accessed: 2024-12-20.
- [Ros96] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, New York, 1996.

- [Ser24a] Erwan Serandour. Immediate tracing for smoother debugging and code exploration. <https://github.com/thurgarion2/PrintWizard/blob/main/report.pdf>, 2024. Accessed: 2024-12-20.
- [Ser24b] Erwan Serandour. Printwizard. <https://github.com/thurgarion2/PrintWizard>, 2024. Accessed: 2024-12-20.