IFS INSTITUTE FOR SOFTWARE

OST Ostschweizer Fachhochschule

Visual OO Debugger

Term Project

Department of Computer Science
OST – University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2021

Author(s):	Gino Cardillo, Alexandre Lagadec, Pascal Schürmann
Advisor:	Prof. Mirko Stocker
Project Partner:	Institute for Software

Abstract

Object-oriented programming can be a challenge for unexperienced or new developers. The relations between objects, variables, and the concept of call-by-reference in methods is difficult to comprehend for a lot of people, sometimes even for more experienced developers. Teaching object-oriented programming can be just as challenging as learning it. One of the best ways to teach this topic, is to visualize the relations between objects and variables. Creating such visualizations is time consuming and the result is often too abstract for learners because such self-made visualizations are usually static images.

The goal of this project is to create a tool for developers, with any level of experience, as well as teachers, to ease the process of learning and teaching the concepts of object-oriented programming. This tool uses debugger information at runtime to visualize objects and variables in a graph. The nodes of the graph represent variables and objects while the edges represent references between variables and objects, as well as references between two objects.

The result was an extension for the widely used and free IDE Visual Studio Code. The Visual OO Debugger, VOOD for short, uses the information the built-in debugger provides to create a graph of the variables and objects. For the visualization of the graph, we used the open-source visualization library vis.js. In case of an object, a node consists of the name of the class in parentheses and, if present, every instance field with a primitive data type including its value. References to other objects are displayed with edges/arrows to other nodes. In case of a variable referencing an object or null, a node simply contains the name of the variable. A variable with a primitive data type contains the type, name of the variable and the value. Newly added nodes and edges are colored in yellow. Multiple options to export the visualization were implemented, as a PNG, PlantUML or GraphViz of the current state or as a GIF of multiple steps. Using the two buttons in the upper left-hand corner of the debugger view, it is possible to load the previous/next state of the visualization, all the way back to the first visualization.

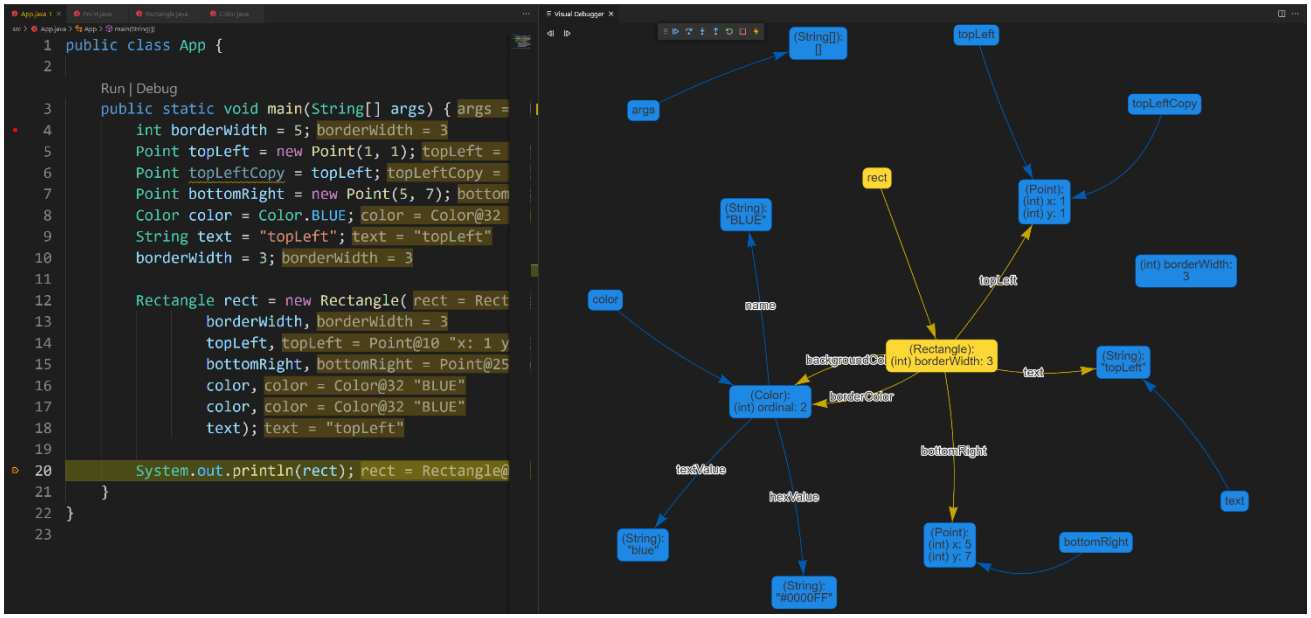


Figure 1 Visual OO Debugger in VS Code

Table of Contents

1 Introduction and Goals 4
1.1 Requirements Overview 4
1.2 Quality Goals 6
1.3 Stakeholders 7
1.4 Existing comparable products 10
1.5 Contextual inquiry 11
2 Constraints 12
3 System Scope and Context 13
3.1 Business Context 13
3.2 Technical Context 14
4 Solution Strategy 15
5 Building Block View 16
5.1 Whitebox Overall System 16
5.2 Level 2 18
5.3 Level 3 20
6 Runtime View 23
6.1 Use Case: vis.js rendering 23
6.2 Use Case: Export GIF 24
7 Deployment View 25
8 Cross-cutting Concepts 26
8.1 User Experience concepts (UX) 26
8.2 Development Concepts 32
9 Design Decisions 33
9.1 VOOD will be developed as a VS Code Extension 33
9.2 vis.js will be used for the visualization 34
9.3 JointJs will be implemented in the Bachelor Thesis 35
10 Quality Requirements 36
10.1 Quality Tree 36
10.2 Quality Scenarios 37
11 Risks and Technical Debts 40
11.1 Risk Assessment 41
11.2 Risk Matrix 42
12 Conclusion 43
12.1 Target Achievement 43

12.2 Outlook 45
13 Glossary 46
14 List of Figures 47
15 List of Tables 48
16 Bibliography 49

1 Introduction and Goals

This document describes the development process and the results of the project “Visual OO Debugger”, short “VOOD”.

The arc42[1]-template was used and extended for this document.

1.1 Requirements Overview

From the initial assignment we derived the following requirements (Figure 2)

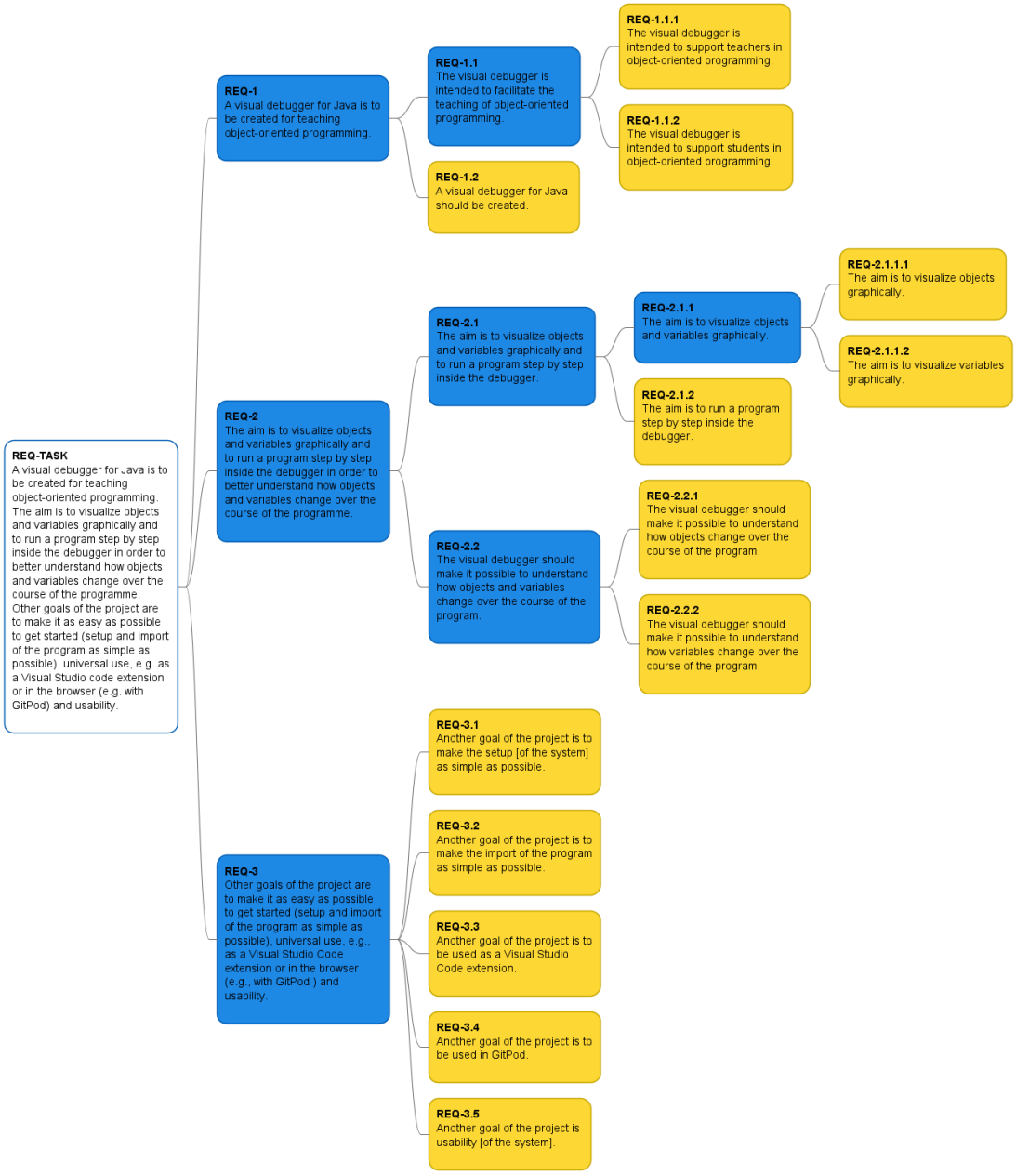


Figure 2 Mind map of the requirements

[1] (arc42, 2021)

ID	Description
REQ-1.1.1	The visual debugger is intended to support teachers in object-oriented programming.
REQ-1.1.2	The visual debugger is intended to support students in object-oriented programming.
REQ-1.2	A visual debugger for Java should be created.
REQ-2.1.1.1	The aim is to visualize objects graphically.
REQ-2.1.1.2	The aim is to visualize variables graphically.
REQ-2.1.2	The aim is to run a program step by step inside the debugger.
REQ-2.2.1	The visual debugger should make it possible to understand how objects change over the course of the program.
REQ-2.2.2	The visual debugger should make it possible to understand how variables change over the course of the program.
REQ-3.1	Other goals of the project are to make it as easy as possible to get started (setup and import of the program as simple as possible), universal use, e.g., as a Visual Studio Code[2] extension or in the browser (e.g., with GitPod[3]) and usability.
REQ-3.2	Another goal of the project is to make the import of the program as simple as possible.
REQ-3.3	Another goal of the project is to be used as a Visual Studio Code extension.
REQ-3.4	Another goal of the project is to be used in GitPod.
REQ-3.5	Another goal of the project is usability [of the system].

Table 1 Requirements

[2] (Visual Studio Code, 2021)
[3] (GitPod, 2021)

1.2 Quality Goals

Form the original requirements, we consider the following (Table 2 Quality goals from requirementsTable 2) as key quality goals.

ID	Quality Characteristic	Description
REQ-2.2.2	Functional Stability	The visual debugger should make it possible to understand how objects change over the course of the program.
REQ-3.2	Operability	Another goal of the project is to make the import of the program as simple as possible.
REQ-3.3	Compatibility	Another goal of the project is to be used as a Visual Studio code extension.
REQ-3.4	Compatibility	Another goal of the project is to be used in GitPod.
REQ-3.5	Operability	A goal of the project is usability [of the system].

Table 2 Quality goals from requirements

A further key quality goal was found during the stakeholder analysis (Table 3).

ID	Quality Characteristic	Description
REQ-4	Maintainability	Well documented and maintainable source code that is open to extensions.

Table 3 Quality goals from stakeholder analysis

1.3 Stakeholders

Group	Contact	Goals	Role(s)	Expectations
OO lecturers: Mirko Stocker	mirko.stocker@ost.ch	Successful completion of the project	Adviser Customer Product Owner Lecturer	MVP as a basis for further thesis and development. A tool for his students to study the runtime behavior of OO programs easily.
Initial developers: Gino Cardillo Pascal Schürmann Alexandre Lagadec	gino.cardillo@ost.ch pascal.schuermann@ost.ch alexandre.lagadec@ost.ch	Successful completion of the project	Developer Student	To gain experience To gain reputation
FOSS community		To gain experience To gain reputation	Developers	Well documented and maintainable source code that is open to extensions.
Lecturers in OO-related subjects: Thomas Letsch (AD) Silvan Gehrig (PF)	thomas.letsch@ost.ch silvan.gehrig@ost.ch	To demonstrate the runtime behavior of high-level OO concepts.	Lecturer	To have a tool with which one can demonstrate the runtime behavior of high-level OO concepts.
Students of OO and related subjects: Patrick Schürmann Alexandre Lagadec	patrick.schuermann@ost.ch alexandre.lagadec@ost.ch	To gain a deeper understanding of OO in general. To gain a deeper understanding of concepts that build on OO.	Student	To have a tool with which one can study the runtime behavior of OO programs in general. To have a tool with which one can study the runtime behavior of high-level OO concepts.

Table 4 Stakeholders

1.3.1 Stakeholder Analysis

Group	Cooperation	Influence	Motivation (s)
OO lecturers	Very Positive	Very High	Very High
Initial developers	Very Positive	Very High	Very High
FOSS community	Positive There is a vast amount of FOSS projects. Those who choose to contribute to our project are likely to be supportive.	Positive There is a vast amount of FOSS projects. Those who choose to contribute to our project are likely to be supportive.	Medium There is a vast amount of FOSS projects. Those who choose to contribute to our project are likely to be motivated to provide at least simple feature requests or bug reports.
Lecturers in OO-related subjects	Positive	Very High Lecturers in OO-related subjects can provide valuable insights on what makes learning high-level OO concepts and OO in general challenging	High Lecturers in OO-related subjects would probably like to demonstrate the runtime behavior of high-level OO concepts.
Students of OO and related concepts	Positive	High Students of OO and related subjects can provide raw feedback on what makes learning high-level OO concepts and OO in general challenging	High Students of OO and related subjects would probably like to study the runtime behavior of high-level OO concepts and OO in general.

Table 5 Stakeholder analysis

1.3.2 Relation map

	OO lecturers	Initial developers	FOSS community	Lecturers in OO-related subjects	Students of OO and related concepts
OO lecturers	-	Very Good Successful kick-off meeting Weekly meetings planned	Unknown	Good Silvan and Mirko even share some lectures	Default Patrick attends OO lecture; no active discussion yet
Initial developers	-	-	None	Default Alexandre attends AD and PF lectures; no active discussion yet	Good Patrick is Pascal's brother
FOSS community	-	-	-	Unknown	Unknown
Lecturers in OO-related subjects	-	-	-	-	Default Alexandre attends AD and PF lectures; no use case yet
Students of OO and related concepts	-	-	-	-	-

Table 6 Relation map of the stakeholders

1.4 Existing comparable products

Title	Authors	Organisation	Category	Features	References
Visual Tracing for the Eclipse Java Debugger	- Bilal Alsallakh - Peter Bodesinsky - Alexander Gruber - Silvia Miksch	TU Wien	Eclipse plug-in	- Tracking - Temporal scaling - Search for variables	Paper (TU Wien) Paper (IEEE) Paper (Research gate) Youtube
Mirur Visual Debugger	Brandon Borkholder	Brandon Borkholder	Eclipse plug-in	- Plots for numeric arrays	Eclipse Marketplace
JIVE	Support: - Demian Lessa - Lead JIVE Developer - Jeffrey K. Czyz - Eclipse/JIVE Developer - Paul V. Gestwicki - Stand-alone JIVE Developer - J. Swaminathan - JIVE Plug-in Developer	University at Buffalo	Eclipse plug-in	- 'Reverse stepping' - Based on UML model	University at Buffalo
OCL-based Runtime Monitoring of JVM hosted Applications	Lars Hamann (H-Man2), Martin Gogolla, Mirco Kuhlmann	Universität Bremen	Stand-alone?	- Based on UML model	Stackoverflow TU Berlin Sourceforge
Visual Debugger	Tim Kräuter	-	IntelliJ plug-in	- Uses native IntelliJ debugger as data source	Tim Kräuters Webseite JetBrains Marketplace Visual Debugger GitHub UI GitHub
Debug Visualizer	Henning Dieterichs	Microsoft (VS Code)	VS Code plug-in	- Dynamic rendering	Visual Studio Marketplace GitHub

Table 7 Existing comparable products

1.5 Contextual inquiry

To gather further information on what the requirements are from a student's perspective, a contextual inquiry was performed.

1.5.1 Method

The inquiry consisted of multiple observations that were performed during the exercises for OOP1. The focus of these observations was on how a student works with the IDE and how he solves problems that occurred during the exercises. After the observation a brief conversation was held which gave the student the chance to discuss what they anticipate for a visualization.

1.5.2 Results

ID	Description
CI-1	The current debugger is used as a last resort tool to solve a problem during the exercises.
CI-2	The modelling of relationships between classes is perceived as a difficult task. A visualization of the relationships would indeed be helpful.
CI-3	The student uses only a part of the screen for the IDE, often the screen will be shared with the pdf description of the exercise or lecture notes. The space that would be available for a visualization is therefore limited.
CI-4	During the exercise lessons, the students work on their private laptops, often without an external mouse. Some curser movements are therefore difficult to perform.

Table 8 Results of contextual inquiry

2 Constraints

ID	Constraint	Consequences
CO-1	The visual OO debugger should be implemented as a VS Code extension.	The constraints and guidelines for VS Code extensions apply for the entire project. VS Code must be used for testing.
CO-2	For the visualization, external libraries should be used.	The libraries have various requirements for the input data. These requirements need to be considered for the construction of the extensions.
CO-3	The project needs to be completed within the Autumn Term 2021.	The time of the project is limited to the duration of the semester.
CO-4	The documentation of the project should meet the formal requirements of a technical publication.	Established templates and solutions need to be evaluated and used to guarantee a certain degree of technical correctness.
CO-5	The VOOD will be published as an open-source project.	English must be used as the official project language.

Table 9 Constraints

3 System Scope and Context

This chapter describes the delimitations of the system from all its communication partners. The visualization is based on the System Context diagram of the C4 Model.[4]

3.1 Business Context

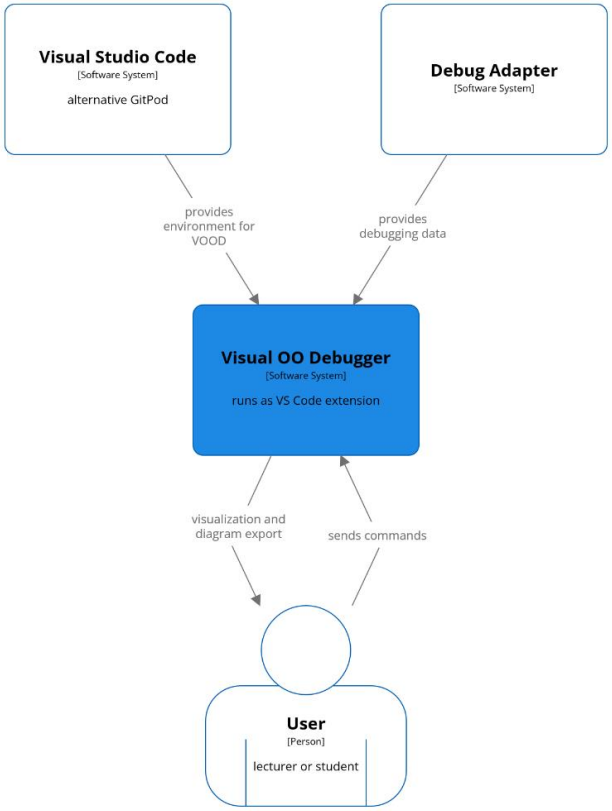


Figure 3 Context diagram

Partner	Communication
User	The user can influence the Visual OO Debugger, either by sending commands or by adjusting the visualization. The user can also export the current visualization.
Visual Studio Code	Besides being the environment for the Visual OO Debugger, Visual Studio Code provides the Visual OO Debugger with: - Redirected user commands - Settings set by the user - Debugger information
Debug Adapter	The debug adapter manages the communication with the debugger. The debug adapter provides the Visual OO Debugger with the debugging data.

Table 10 Description of the business context

[4] (C4 model, 2021)

3.2 Technical Context

The Visual OO Debugger requires the following API's and protocols.

API	Definition
(VS Code API, 2021)	https://code.visualstudio.com/api/references/vscode-api
(Debug Adapter Protocol, 2021)	https://microsoft.github.io/debug-adapter-protocol/specification

Table 11 Required API's and protocols

VS Code API

The VS Code API allows the Visual OO Debugger to access the functionality and data of VS Code. The following VS Code API features are relevant for the visual OO debugger:

Feature	Description
commands	For registering and listening to commands
debug	Provides functionalities for accessing the debugger
DebugSession	Provides access to the current debug session
ExtensionContext	Provides a collection of utilities private to the extension.
Uri	A universal resource identifier for representing a resource
ViewColumn	To specify a location of a window inside VS Code
Webview	To display html content inside VS Code
WebviewPanel	For handling a window containing a Webview
window	Namespace of the currently active window
workspace	Give access to the current workspace

Table 12 Used VS Code API features

Debug Adapter Protocol

Visual Studio Code communicates with the debugger through the debug adapter. The debug adapter is an intermediate component that normalizes the access to different debuggers. It is possible for the Visual OO Debugger to send requests to the debug adapter using the debug adapter protocol. These requests allow access to the following resources:

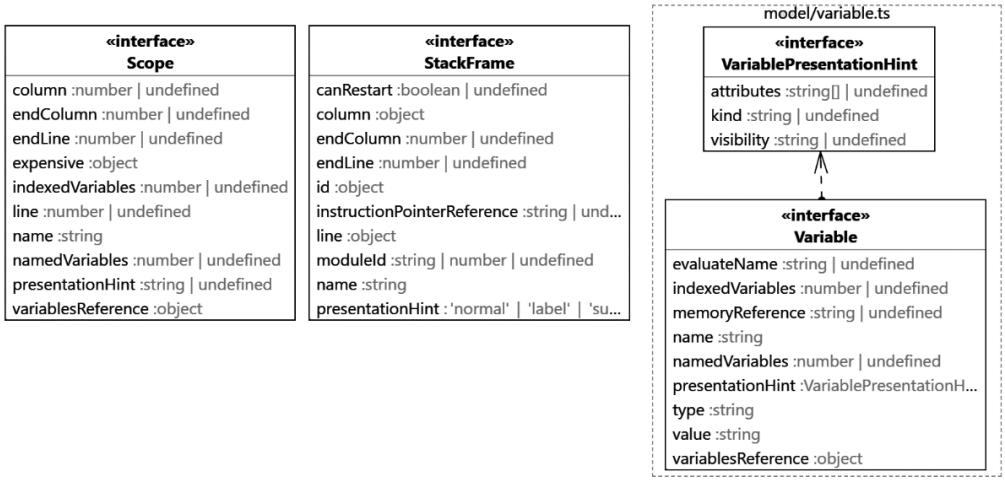


Figure 4 Class diagram of debug adapter protocol models

4 Solution Strategy

Goal/Requirements	Solution approach
REQ-1.1.1	The visualization can be exported as a PNG, animated GIF, PlantUML or GraphViz for further use in teaching.
REQ-2.1.1.1 & REQ-2.1.1.2	vis.js[5] will be used for the visualization.
REQ-3.1 & REQ-3.2	The finished extension will be made available over the VS Code extension marketplace.
CO-1	The extension will be written in Typescript, as it is the standard for VS Code extensions.
CO-2	A neutral visualization interface will be created which is then implemented by the designated visualization.

Table 13 Solution strategy

[5] (vis.js, 2021)

5 Building Block View

The building block view shows the static decomposition of the system into building blocks as well as their dependencies

5.1 Whitebox Overall System

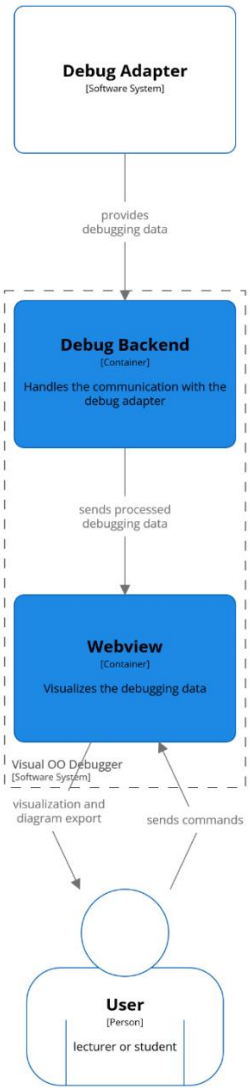


Figure 5 Class diagram of debug adapter protocol models

The Visual OO Debugger can be roughly divided in two parts, the debug backend and the webview. This split was made to separate the logic for visualizing the data and retrieving and processing the debugging data. Furthermore, this solution simplified the distribution of work inside the project team.

5.1.1 Debug Backend

Responsibility

The debug backend handles the communication with the debug adapter of VS Code. This goal can be split in three separated tasks:

- Handling debugging events
- Retrieving the data from the external debug adapter
- Process the received debugging data for the webview.

Interfaces

- The debug backend communicates with the debug adapter via the debugger adapter protocol.
- The debug backend is given an instance of a webview class which is called whenever a debugging event is triggered.

5.1.2 Webview

Responsibility

The webview is responsible for rendering the visualization of the debugging data. Besides the visualization, the webview handles user interactions. This includes:

- The user interactions using commands
- User interactions with the visualization, either by repositioning elements or by using the back-stepper function

Interfaces

- The user can send commands to the Visual OO Debugger to open the webview panel.
- The user can trigger an export by sending a command.
- The user can interact with the visualization on the webview panel
- An instance of a webview object is given to the debug backend. When the debug backend detects a debug event, an update function will be triggered.

5.2 Level 2

Level 2 specifies the inner structure of the building blocks in the overall system.

5.2.1 White Box Debug Backend

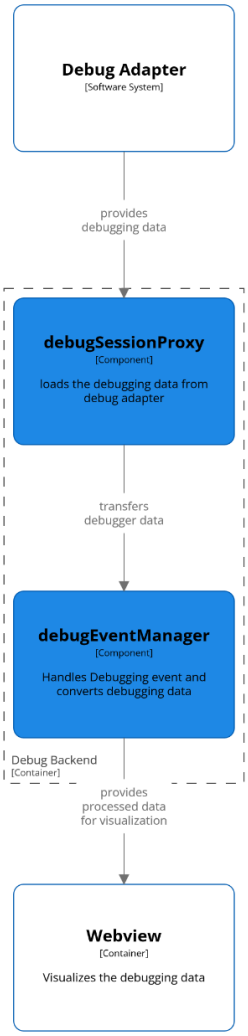


Figure 6 Component diagram of debug backend

The debug backend consists of two components, the debug session proxy, and the debug event manager.

The debugSessionProxy handles the communication with the debug adapter and receives debugging data.

The debugEventManager handles the debugging events. If the debugEventManager detects that the debugger has stopped, it will request the debugging data from the debugSessionProxy. The debugging data received by the debugSessionProxy will then be further processed before it is sent to the webview for the visualization.

5.2.2 White Box Webview

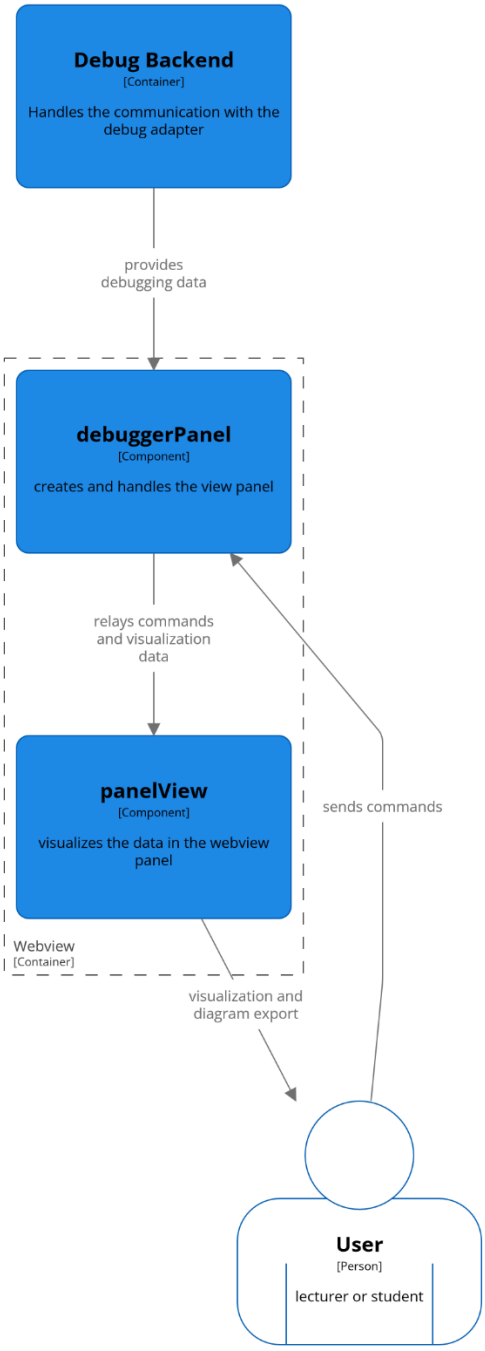


Figure 7 Component diagram of webview

The webview can be split in two basic components, the debuggerPanel and the panelView.

The debuggerPanel creates the view panel and handles incoming commands for the visualization.

The panelView renders the visualization, provides the possibility for the user to reposition elements and handles the export of a diagram.

5.3 Level 3

Level 3 specifies the inner structure of the building blocks in level 2.

5.3.1 White Box DebugEventManager

Figure 8 Class diagram of DebugEventManager

The DebugEventManager is a single class with a DebugSessionProxy instance to load the debug data.

The DebugEventManager creates the event handler for when the debugger stopped in his constructor. In this case, the event handler uses the debug session proxy to load all current variables. These variables are then processed to PanelViewInputs for the webview to render.

5.3.2 White Box PanelView

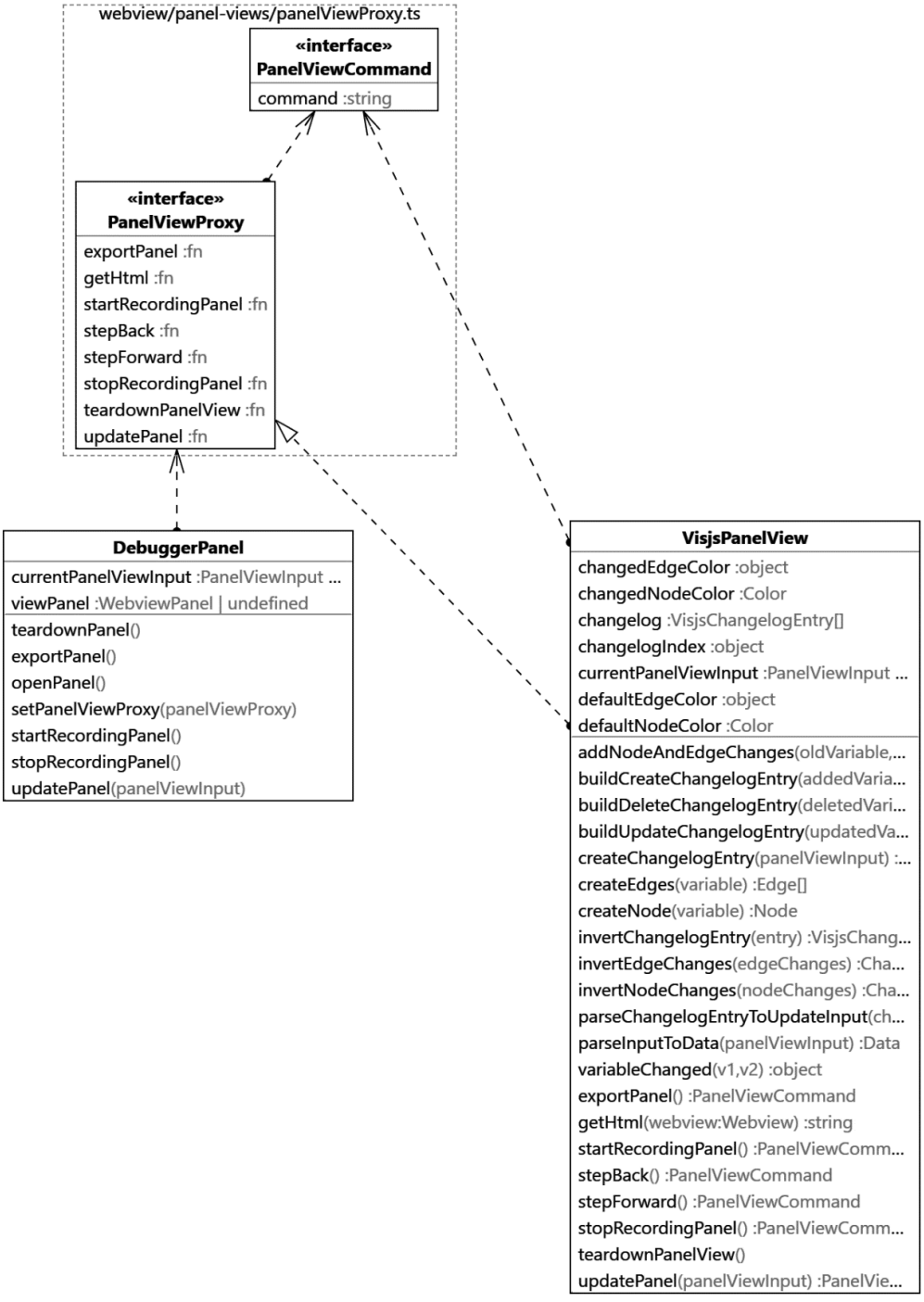


Figure 9 Class diagram of PanelView

The PanelView is implemented as the interface PanelViewProxy with concrete implementations. VisjsPanelView is one of these concrete implementations which renders the debug data as visjs diagram. The visjsPanelView itself calls an html file where the rendering of the diagram takes place. The communication between the Visual OO Debugger and the html page is handled by PanelViewCommands.

This html page is rendered in a webview panel, which the DebuggerPanel constructs and manages.

5.3.3 Panel View Variable

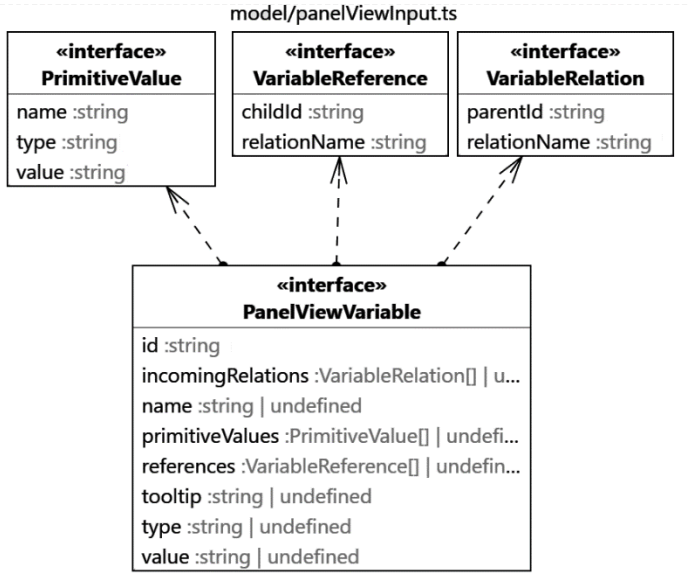


Figure 10 Class diagram of PanelView variables

The DebugEventManager prepares the variables for the webview as PanelView variables

6 Runtime View

The runtime view describes concrete behavior and interactions of the system's building blocks in form of scenarios.

6.1 Use Case: vis.js rendering

Brief use case

The user can open a new VOOD panel with the command "VOOD: Open debugger view". When the user starts the debugging process in VS Code, a visualization of the current debugging step will be rendered in the panel. This visualization uses the vis.js library. Between the debugging steps the user can reposition elements.

Sequence diagram

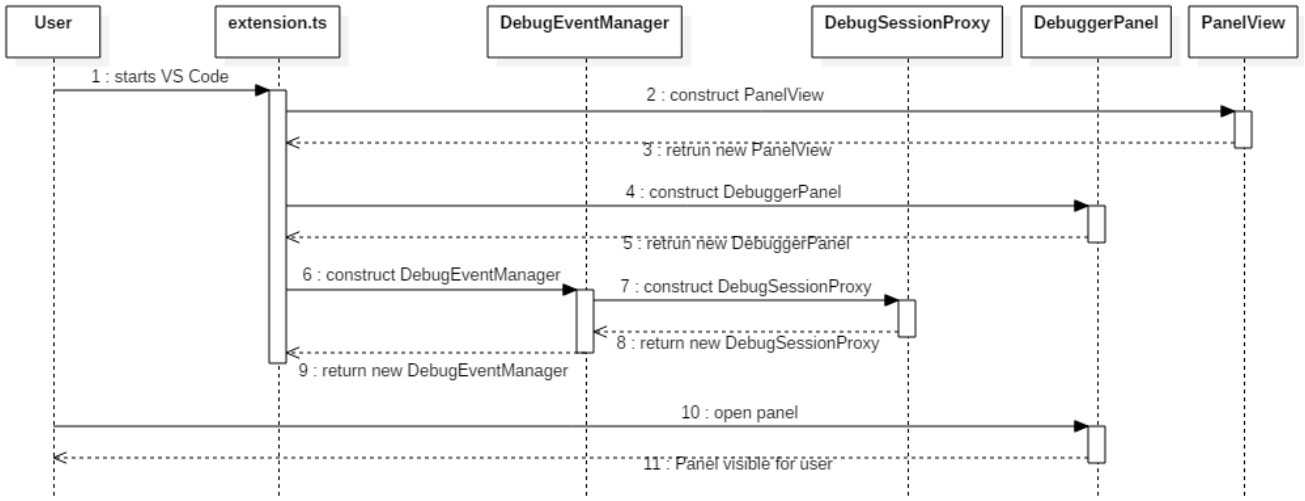


Figure 11 Sequence diagram of the extension's start

1. The user starts VS Code, if the VOOD extension is enabled, the extension.ts file of the Visual OO Debugger extension will start to set up the extension.
2. The first step is to create the PanelView. Based on the user settings, the extension.ts will use a concrete implementation of the PanelView interface.
4. The DebuggerPanel gets initialized with the newly created PanelView
5. After the DebuggerPanel is created, the extension.ts will register the user commands to interact with the DebuggerPanel.
6. When the DebugEventManager is created, the constructor of the DebugEventManager creates the DebugSessionProxy and sets up the event handler for the debugger.
9. After the DebugEventManager is set up, the extension is ready to start.
10. The user can use the command "VOOD: Open debugger view", which is processed by the DebuggerPanel and opens a new webview panel.

6.2 Use Case: Export GIF

The Visual OO Debugger has multiple export features. For demonstration purposes, this section will discuss the inner workings of the GIF export.

Brief use case

A user can start recording a GIF of what is currently visualized with the command “VOOD: Start recording a GIF”. With “VOOD: Stop recording a GIF”, the user can stop the recording and download the GIF.

Sequence diagram

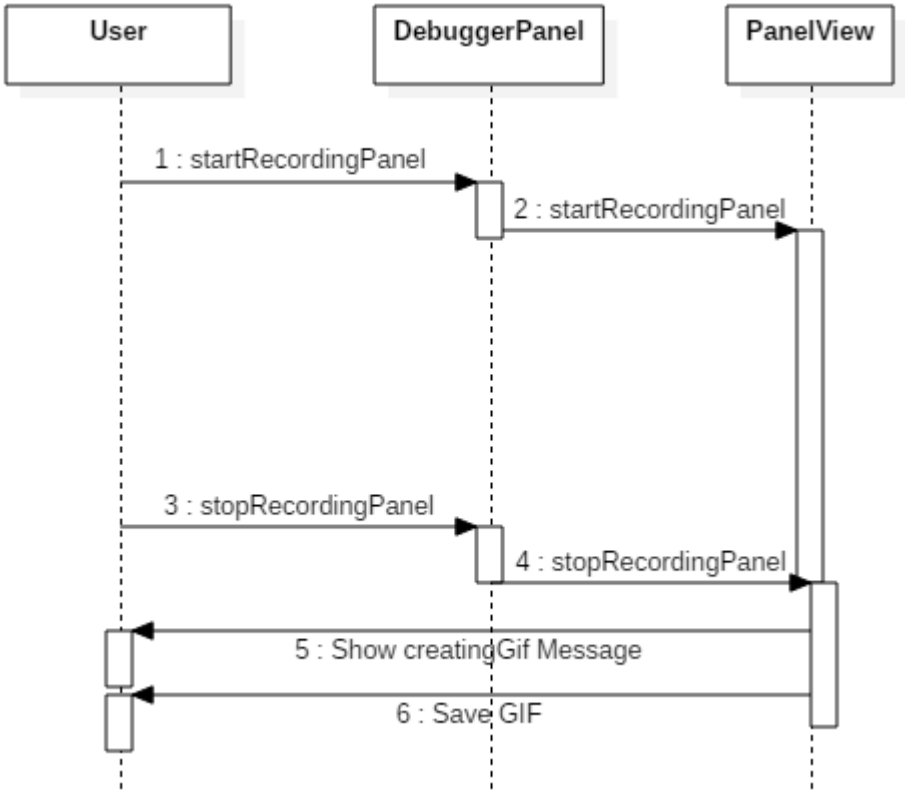


Figure 12 Sequence diagram of GIF export

1. The user runs the command “VOOD: Start recording GIF”, which is redirected to the DebuggerPanel as a startRecordingPanel function call.
2. The command to start the recording is redirected to the PanelView, where the recording of the GIF starts.
3. When the user has made his actions that he wishes to record he can stop the recording with “VOOD: Stop recording a GIF”, which calls the stopRecordingPanel function of the DebuggerPanel.
4. The PanelView processes the recording, creates a GIF, and saves it.

7 Deployment View

The project makes use of GitHub[6] Actions to enable Continuous Deployment. Three workflows are defined.

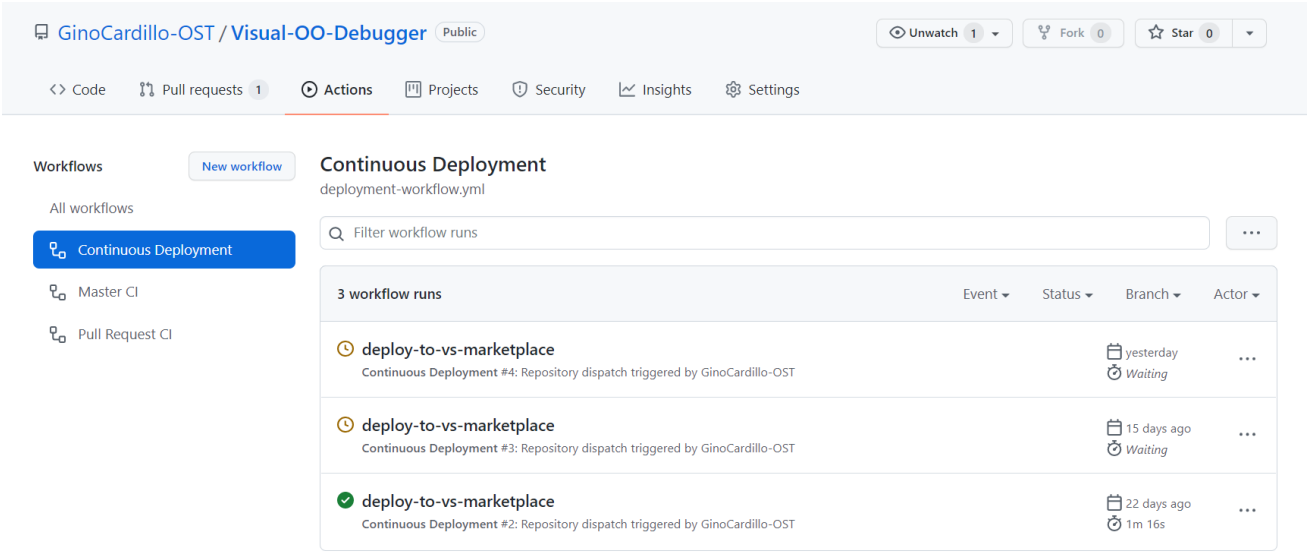


Figure 13 GitHub workflow overview

Pull Request CI

The Pull Request CI workflow is executed whenever a pull request is created or updated. This workflow does a checkout of the code and then runs linting checks, formatting checks, unit tests and integration tests. Only if all those tasks succeed, the pull request can be merged.

Master CI

The Master CI workflow is triggered on every commit to the master branch. First it does the same checks as the Pull Request CI workflow. If that succeeds, another job is started which uses a GitHub action to create a repository dispatch event. This dispatch event will then trigger the Continuous Deployment workflow.

Continuous Deployment

The Continuous Deployment workflow can only be triggered by a repository dispatch event. This workflow only has one job for the deployment, which is executed on the production environment. Any workflows that operate on the production environment must be reviewed before the jobs can start. If approved, the job does a checkout of the code on the master branch, builds it, and publishes it to the VS Marketplace.

[6] (GitHub, 2021)

8 Cross-cutting Concepts

This chapter describes overall, principal regulations and solution ideas that are relevant in multiple parts of your system.

8.1 User Experience concepts (UX)

The UI of the extension must be designed in a way that is intuitive for the user and gives enough options to freely visualize the current debugging step.

VS Code offers different UI elements for extension. But the usage of these elements is regulated by the extension guidelines of Visual Studio.

For the Visual OO Debugger it is sufficient to use the command palette for the basic commands. The commands will be described in the README.md file of the extension, which is shown on the extension marketplace site. Further UI elements for the control of the extension aren't needed. They would clutter the UI.

For rendering the visualization, a WebView will be used. This approach gives the most flexibility and allows the use of external visualization libraries. User input, which directly effects the visualization, for example the back-stepper function, is handled by the WebView directly.

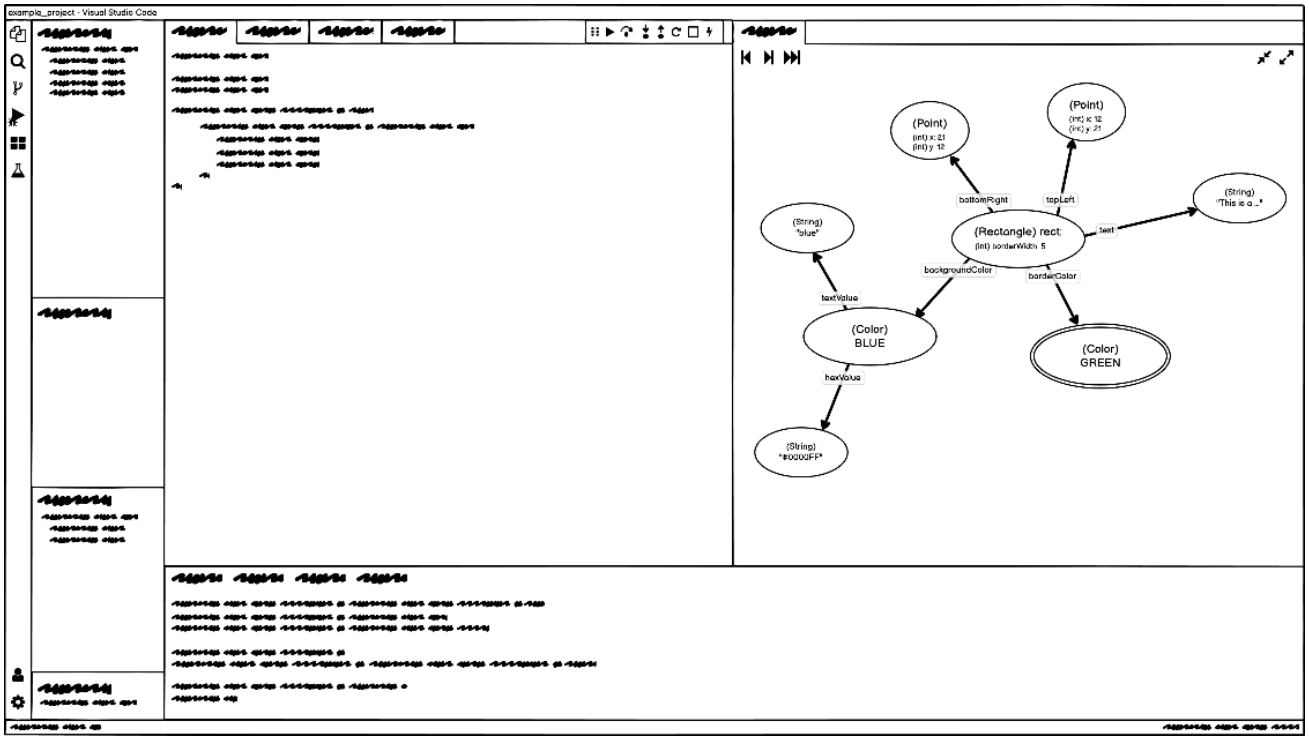


Figure 14 Wireframe of VS Code Integration

8.1.1 Usage of vis.js

The Visual OO Debugger can use different libraries to visualize the current debugging step. For the term project, the first library to be used is vis.js. The vis.js library can position the nodes by itself and therefore no calculation for positioning the elements is needed. Another benefit is that the user can easily manipulate the visualization with drag and drop if needed.

Low fidelity Wireframes

These low fidelity Wireframes depict how various data types are displayed as well as how the extension will be integrated inside VS Code.

The Wireframes were created using Balsamiq[7].

Primitive Types, Strings and null

Although Strings technically are objects, because they are so common, we decided to display them as if they were a primitive type instead of the underlying byte array because it is a commonly used object and to increase comprehensibility.

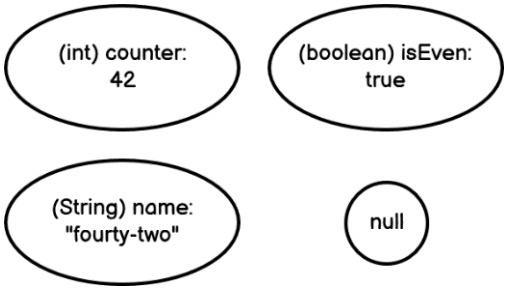


Figure 15 Wireframe of primitive types

Simple Object

A simple object only consists of primitive types and Strings. Three versions have been created to display a simple object.

Simple Object Version 1

In this version both primitive types and Strings are separate bubbles and connected with an arrow.

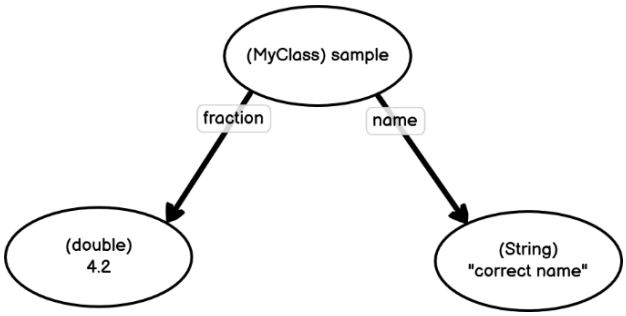


Figure 16 Wireframe of simple object version 1

Simple Object Version 2

In this version both primitive types and Strings are part of the object's bubble.

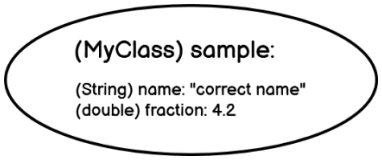


Figure 17 Wireframe of simple object version 2

[7] (Balsamiq, 2021)

Simple Object Version 3

In this version primitive types are part of the object's bubble and Strings are in separate bubbles.

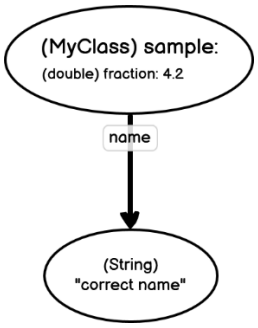


Figure 18 Wireframe of simple object version 3

Simple Object Decision

We decided to go with version 3 for displaying simple objects. This way, an arrow is defined as a reference and a String is correctly used as an object.

Potentially it could be configurable to select the more compact version 2.

Arrays of Primitive Types and Strings

Arrays of primitive types and Strings have a simplified display. An array that exceeds a certain length will be shortened and a tooltip will show the full content of the array.

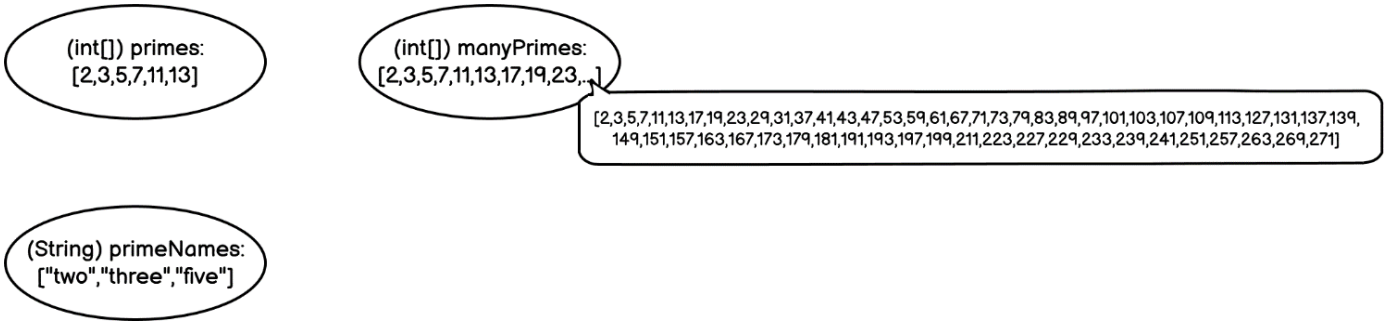


Figure 19 Wireframe of arrays of primitive types and strings

Array of Simple Objects

The indexes are the references between the array and its objects.

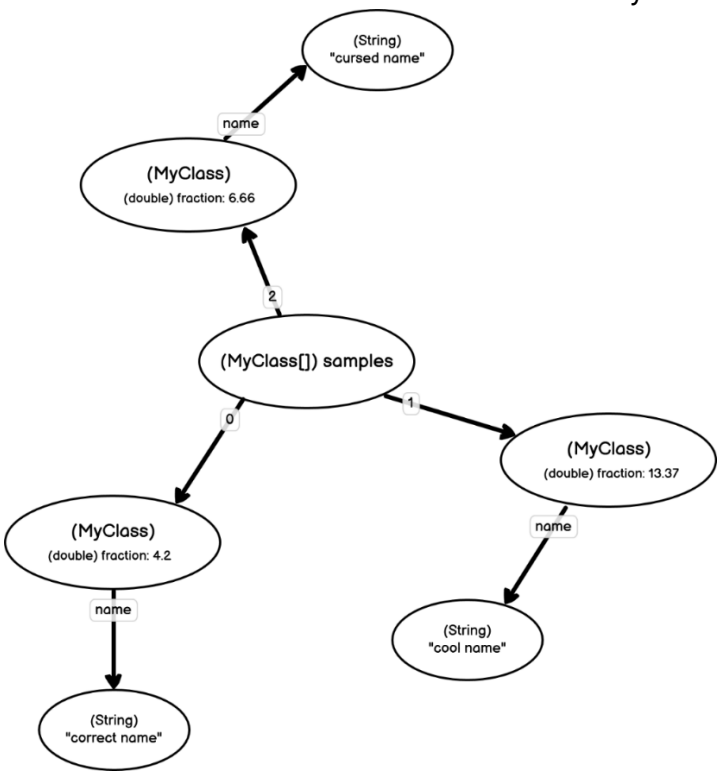


Figure 20 Wireframe of object array

Map of String to Object

A map is very similar to an array. All entries of a map are saved in some specialization of the Collection interface. Therefore, the nodes are indexed.

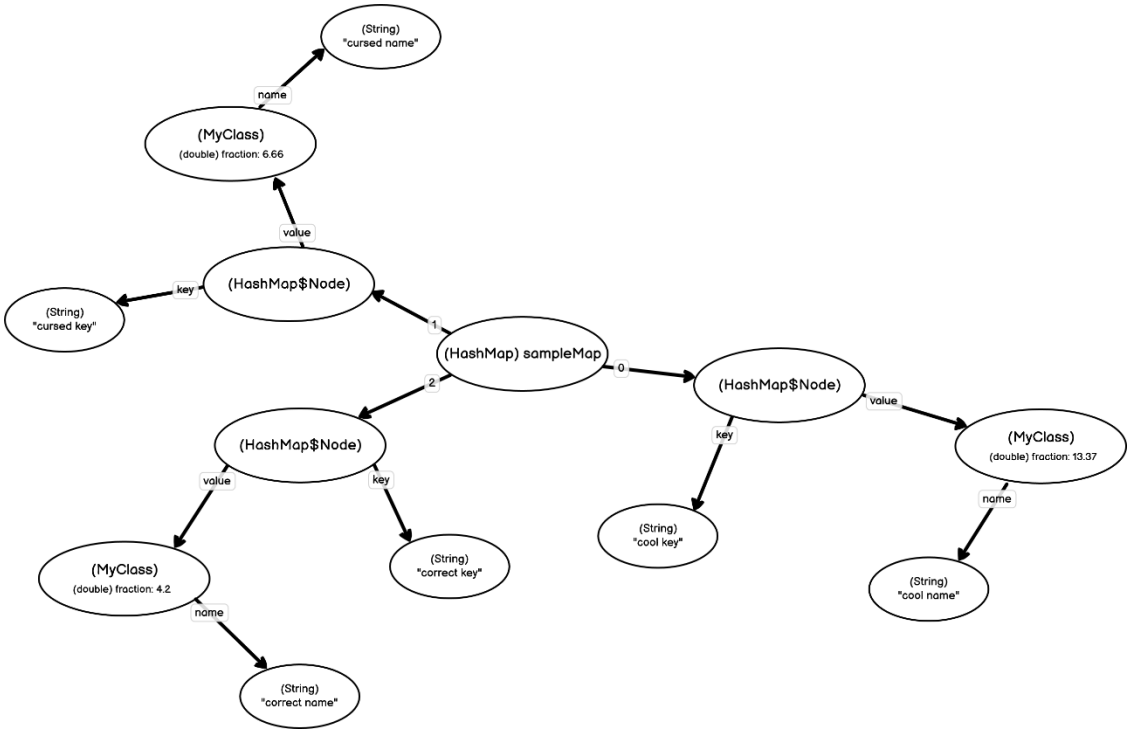


Figure 21 Wireframe of String to Object Map

Enum

An Enum is special because it has a value (e.g., “RED”) and can have references.

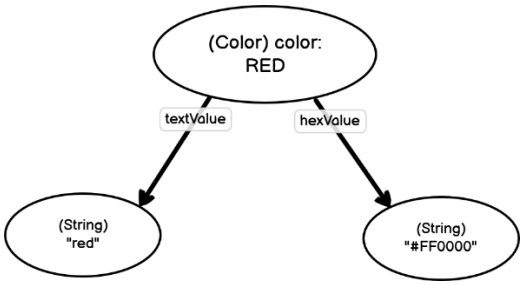


Figure 22 Wireframe of an Enum

Composite Object

A composite object has references to other objects which are connected by arrows.

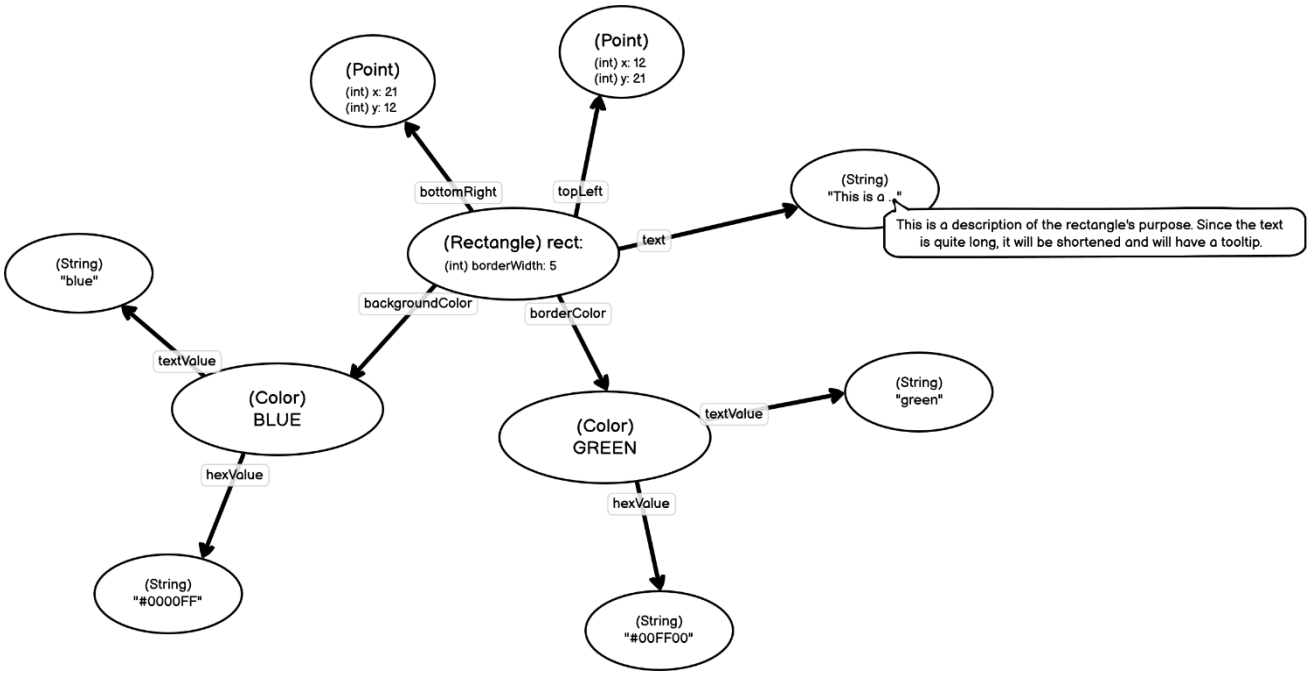


Figure 23 Wireframe of Composite Object

Collapsed Objects

If a graph becomes too big, the user can collapse objects or a variable.

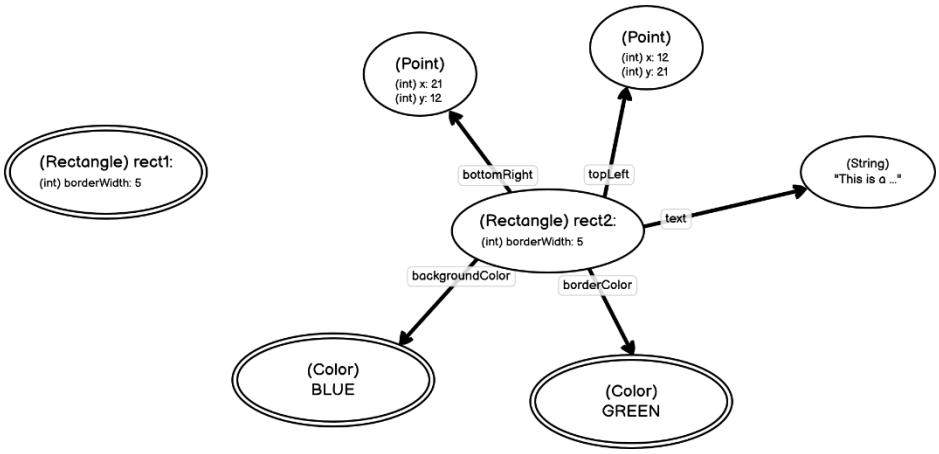


Figure 24 Wireframe of Collapsed Objects

This feature was ultimately not implemented.

8.2 Development Concepts

This chapter describes concepts that were used during development, with focus on code quality.

8.2.1 Code Review

For maintaining and developing the code, GitHub was used. New changes to the code will be developed in feature branches. To merge a feature branch into the master branch, a pull request will be created. Only if at least one other team members approves the changes, they will be merged into the master branch.

8.2.2 Code Guidelines

To ensure a coherent code style, prettier will be used.

To ensure a certain degree of code quality, ESLint[8] and SonarCloud[9] will be used.

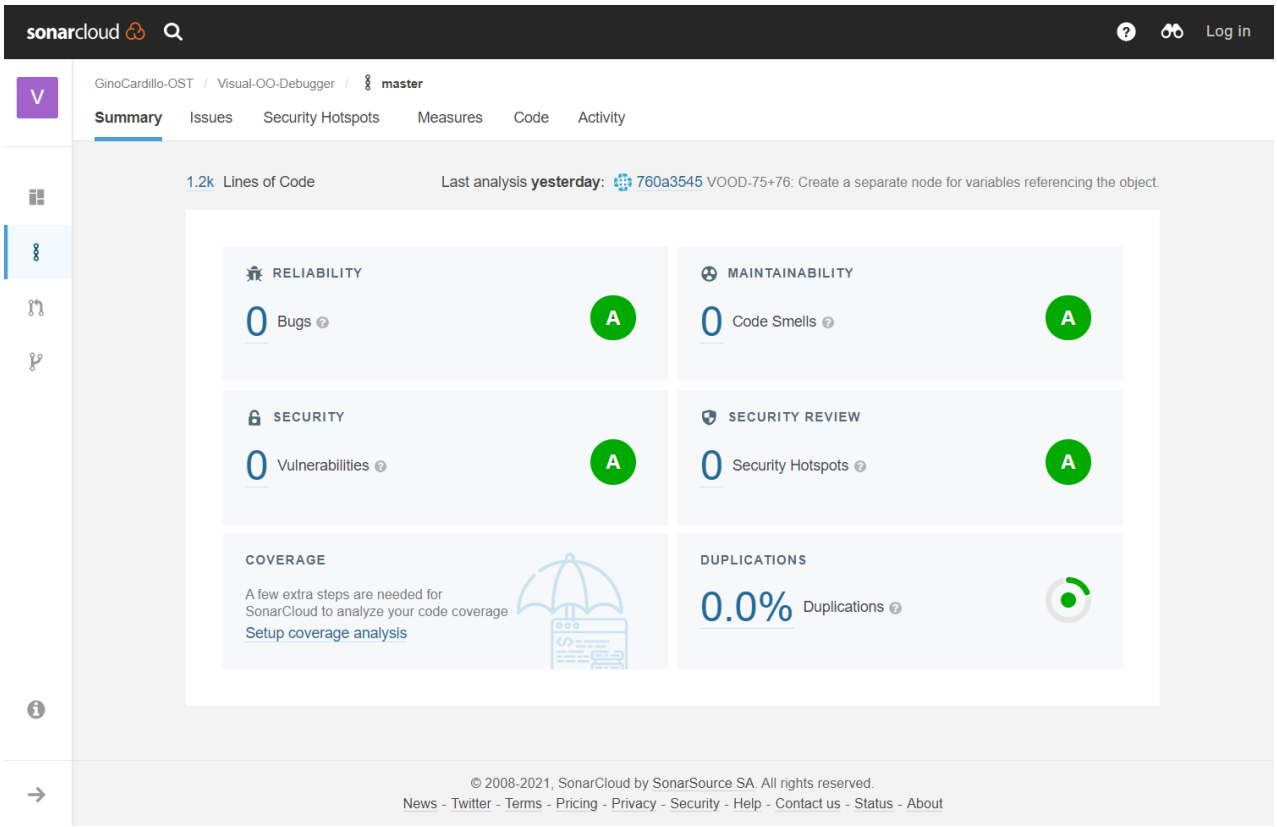


Figure 25 SonarCloud summary

[8] (ESLint, 2021)
[9] (SonarCloud, 2021)

9 Design Decisions

This chapter documents the most important design decisions for the development of the Visual OO Debugger.

9.1 VOOD will be developed as a VS Code Extension

History

Date	Step
23.09.2021	Initial research
23.09.2021	Decision was made to develop VOOD as a VS Code extension

Table 14 History of decision to develop VOOD as VS Code Extension

Context

The initial pitch of the project did not specify an IDE for which the visualization should be implemented. Therefore, different solutions were investigated to find a suitable platform for the solution.

The following options were discussed:

- Implementation as a Visual Studio Code extension
- Implementation as an IntelliJ Plugin

Decision

The decision was made to implement the VOOD as a VS Code extension. The reason for this decision is that VS Code is free to use, and VS Code extensions can also be used in GitPod, a web IDE.

Consequences

- The project depends on the VS Code extension API
- The project depends on the debug adapter protocol
- TypeScript will be used as programming language

9.2 vis.js will be used for the visualization

History

Date	Step
04.10.2021	Initial research
04.10.2021	Decision to use vis.js for visualization

Table 15 History of decision to use vis.js for visualization

Context

There were multiple possible approaches for how to render a visualization of the debugging steps.

The following Options were discussed:

- Usage of vis.js for visualization
- Stepwise rendering as a PlantUML diagram
- Developing an own visualization

Decision

The decision was made to use vis.js for the visualization, and to construct the extension to support different visualizations. vis.js offers a good-looking visualization of the debugging steps and can position the elements by itself.

Consequences

- Webview will be needed to render vis.js
- Dependency to vis.js
- The architecture must support multiple visualization options

9.3 JointJs will be implemented in the Bachelor Thesis

History

Date	Step
08.11.2021	Discussion implementation of JointJs[10] as an alternative visualization
15.11.2021	Proposition to implement JointJs in a bachelor thesis
22.11.2021	Team members agree to implement JointJs in the bachelor thesis

Table 16 History of decision to split JointJS off in a separate project

Context

After the implementation of the vis.js visualization, the discussion was held on what alterative visualization could be added. The vis.js solution was visually appealing but had its limitation.

The following options where considered:

- To use JointJs as an alternative visualization
- Implementation of a table-based visualization

Decision

After creating low fidelity wireframes for the table-based visualization, the decision was made to not proceed any further with this visualization. It didn't provide any new information to the user which is not already available in the default debugger.

A prototype was built with JointJs which looked promising. A major drawback is that JointJs doesn't position the elements by itself. Therefore, an implementation of a JointJs view needs to calculate the size and position of the elements by itself. It was decided that this would consume too much time for the term project and is better suited for a bachelor thesis.

The implementation of JointJs and some other features are therefore moved to a bachelor thesis where the project team will continue working on the Visual OO Debugger.

Consequences

No immediate consequences for the project.

[10] (JointJS, 2021)

10 Quality Requirements

This chapter contains all quality requirements as a quality tree with scenarios. The most important ones have already been described in section “1.2 Quality Goals”.

10.1 Quality Tree

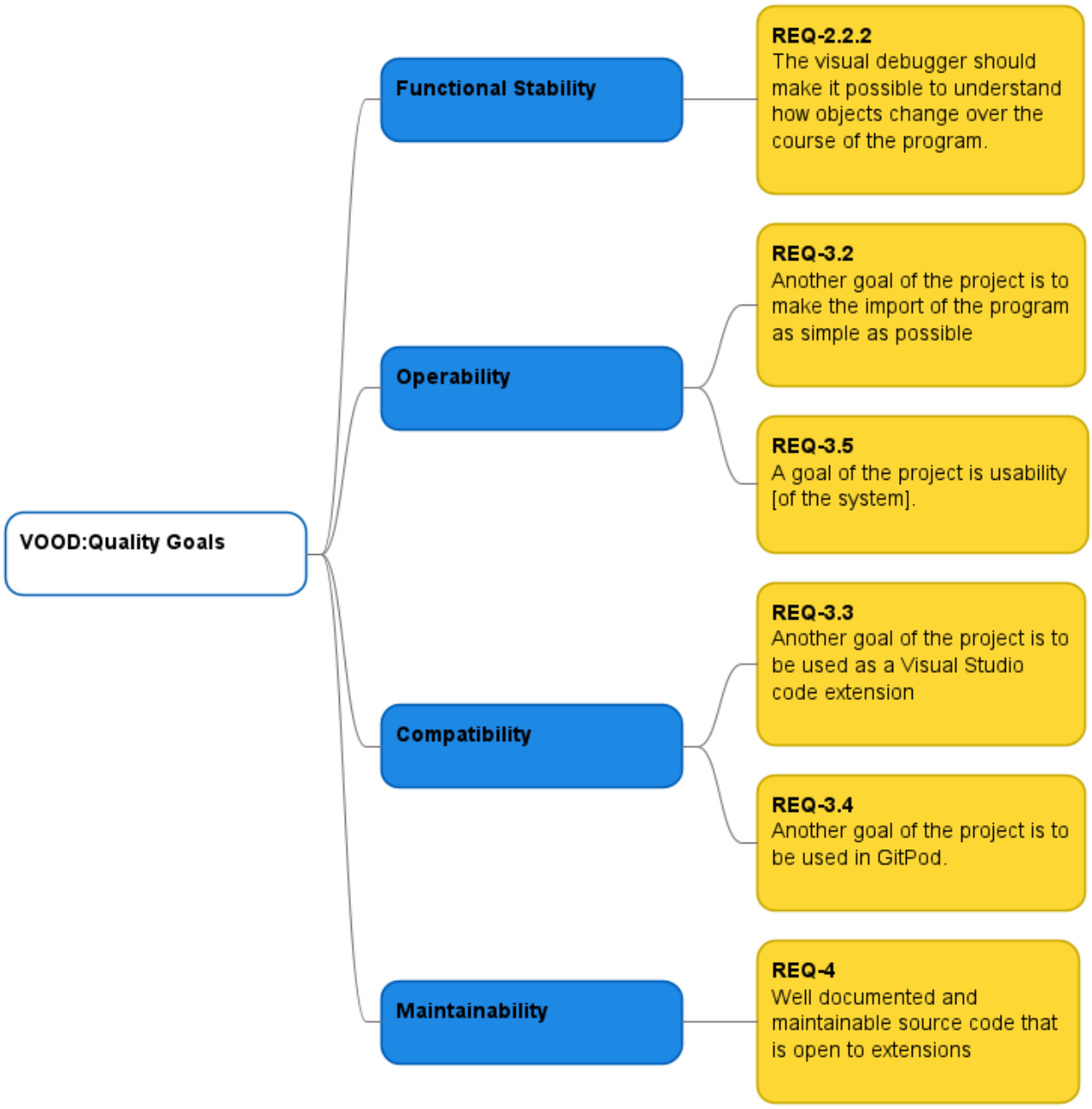


Figure 26 Quality tree

10.2 Quality Scenarios

This section concretizes quality requirements using scenarios.

10.2.1 REQ-2.2.2

Scenario	User uses VOOD during an OO exercise	
Business Goals	The VOOD should make it possible to understand how objects change over the course of the program.	
Relevant Quality attributes	Functional Stability	
Scenario Components	**Stimulus**	User runs extension during an OO exercise
	Stimulus Source	User
	Environment	VOOD is installed as extension in VS Code. The OO exercise is opened in VS Code and the debugger has started
	Artifact	PanelView
	Response	The system visualizes the debugging steps in a way that helps the user to understand what happens
	Response Measure	The user feels that the VOOD helped him to understand the OO exercise

Table 17 Quality scenario for REQ-2.2.2

10.2.2 REQ-3.2

Scenario	User installs the VOOD	
Business Goals	Another goal of the project is to make the import of the program as simple as possible.	
Relevant Quality attributes	Operability	
Scenario Components	**Stimulus**	The VOOD extension is installed and started for the first time in VS Code
	Stimulus Source	New user
	Environment	A published version of VOOD available on the VS marketplace
	Artifact	-
	Response	The VOOD can be installed from the VS marketplace. The introductory text on the VS marketplace page should instruct the user on how they can work with the extension
	Response Measure	The user can install and use the VOOD within reasonable time

Table 18 Quality scenario for REQ-3.2

10.2.3 REQ-3.3

Scenario	Installation in Visual Studio Code	
Business Goals	Another goal of the project is to be used as a Visual Studio Code extension.	
Relevant Quality attributes	Compatibility	
Scenario Components	**Stimulus**	The VOOD extension is installable and useable in VS Code
	Stimulus Source	User
	Environment	A published version of VOOD available on the VS marketplace
	Artifact	-
	Response	The VOOD can be installed from the marketplace. The extension should be usable in VS Code
	Response Measure	The User can install and run the VOOD in VS Code

Table 19 Quality scenario for REQ-3.3

10.2.4 REQ-3.4

Scenario	Installation in GitPod	
Business Goals	Another goal of the project is to be used in GitPod.	
Relevant Quality attributes	Compatibility	
Scenario Components	**Stimulus**	The VOOD extension is installable and useable in GitPod
	Stimulus Source	User
	Environment	A published version of VOOD available on the VS marketplace
	Artifact	-
	Response	The VOOD can be installed from the marketplace. The extension should be usable in GitPod
	Response Measure	The user can install and run the VOOD in GitPod

Table 20 Quality scenario for REQ-3.4

10.2.5 REQ-3.5

Scenario	User uses system to debug a simple solution	
Business Goals	Another goal of the project is usability [of the system]	
Relevant Quality attributes	Operability	
Scenario Components	**Stimulus**	User runs command “VOOD: Open debugger view” and starts debugging
	Stimulus Source	User
	Environment	VOOD is installed as extension in VS Code. The User has read the VS marketplace page
	Artifact	PanelView, marketplace page, registered commands
	Response	The system visualizes the debugging steps in a way that is understandable for the user
	Response Measure	The user can use the functions of the extension without a problem, after he read the VS marketplace page

Table 21 Quality scenario for REQ-3.5

10.2.6 REQ-4

Scenario	Implementation of new view for the VOOD	
Business Goals	Having a well-documented and maintainable source code that is open to extensions.	
Relevant Quality attributes	Maintainability	
Scenario Components	**Stimulus**	A new view for the VOOD is developed
	Stimulus Source	Future developer
	Environment	VOOD is in a stable condition
	Artifact	The extension.ts and the extension manifest
	Response	The view can be added to the system by adding a new templateView to the repository and configuring the extension.ts and the extension manifest
	Response Measure	The developer can implement a new view

Table 22 Quality scenario for REQ-4

11 Risks and Technical Debts

Nr.	Title	Description	Expected effects	Prevention	Behavior on entry
R0	PM :: Bad work package ordering	The order in which work packages are processed is causing the project to stall	Delays	Analyze inter-package dependencies during planning	Assign all available team members to blocking work packages
R1	PM :: Poor requirements analysis	Requirements are not properly understood, approved, and prioritized	Delays	Regular validation of the requirements internally and with stakeholders	Re-evaluate erroneous requirements with stakeholders
R2	CI/CD :: A build failure in the CI/CD pipeline blocks the project	A build failure in the CI/CD pipeline blocks the project	Delays	Develop guidelines: wait for CI/CD results each day before finishing work	Responsible developers fix build problems immediately and notify all blocked team members
R3	Dev :: Technology mastery :: Inefficient extension development	Functionalities and best practices related to the chosen extension framework are unknown, leading to unnecessary efforts and/or delays	Delays, unnecessary complexity	Study extension development tutorials	Perform design reviews on a regular basis and maintain a knowledge base for key insights

Table 23 Identified risks

11.1 Risk Assessment

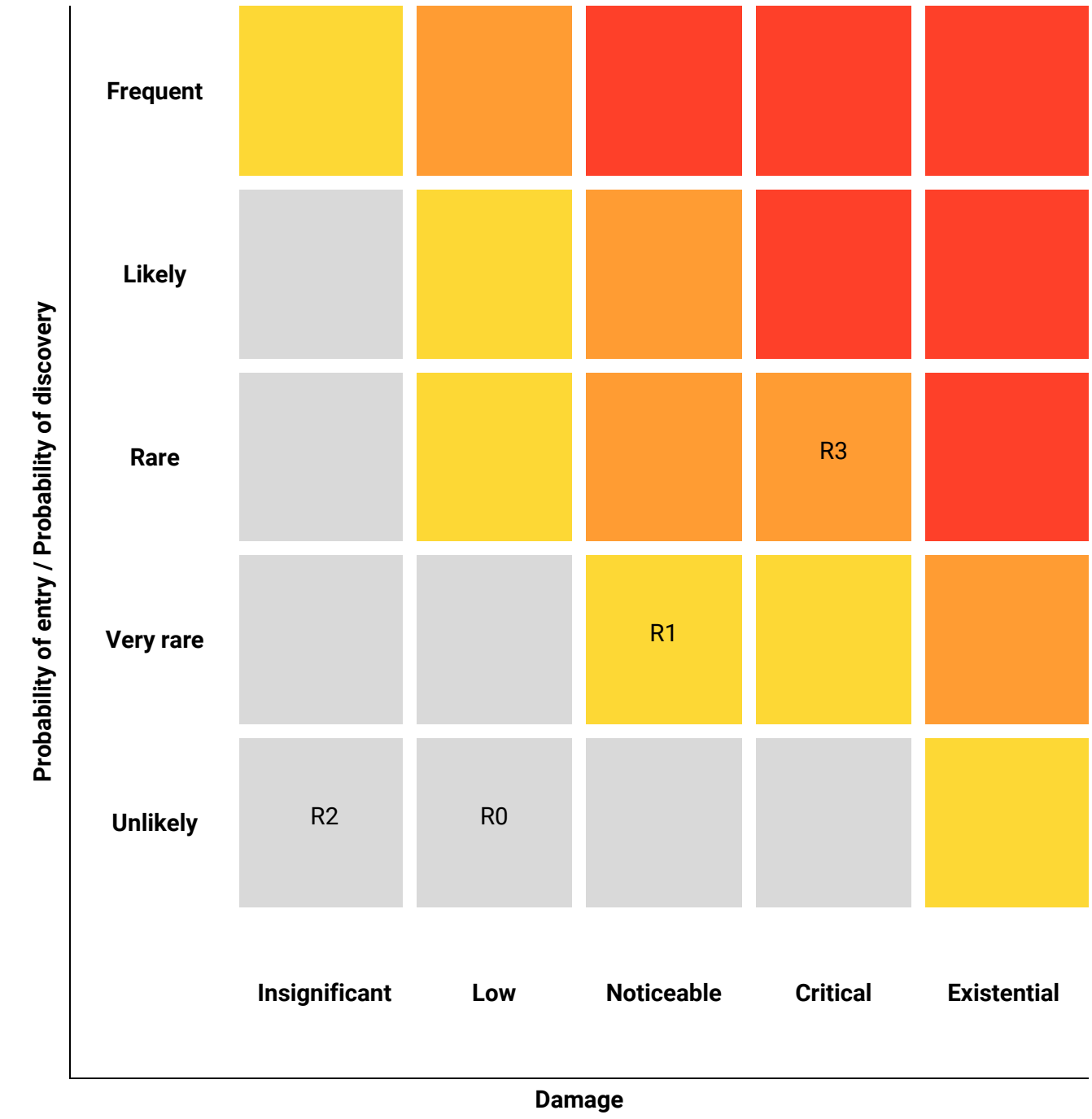
Nr.	Max. damage	Probability of occurrence	Probability of discovery	Probability of occurrence	Weighted damage
R0	60	5%	50%	10%	6
R1	120	10%	25%	40%	48
R2	60	5%	100%	5%	3
R3	120	20%	25%	80%	96
Sum	**360**				**153**

Table 24 Risk probabilities

After planning risk prevention and containment, we were able to reduce all identified risks to an acceptable level.

The weighted damage adds up to 153 hours, which is 21.25% of the time budget of 720 hours. In order to reserve time for dealing with the expected risks, we have increased our time estimates accordingly by around 20% during the sprint planning.

11.2 Risk Matrix



Low risk, no countermeasures required

Moderate risk, consider countermeasures

High risk, countermeasures required

Inacceptable risk, countermeasures urgently required

Figure 27 Risk matrix

12 Conclusion

This chapter contains an evaluation of the project as well as an outlook for further work on the Visual OO Debugger.

12.1 Target Achievement

The target achievement is evaluated by looking at each of the requirements defined in the section "1.1 Requirements Overview".

REQ-1.1.1

"The visual debugger is intended to support teachers in object-oriented programming."

Achieved: yes

The Visual OO Debugger generates visualizations of objects at runtime which supports teachers by saving time, since they don't have to create their own visualizations.

REQ-1.1.2

"The visual debugger is intended to support students in object-oriented programming."

Achieved: yes

The visualizations generated by the Visual OO Debugger help understanding the references between variable and objects as well as references between two objects.

REQ-1.2

"A visual debugger for Java should be created."

Achieved: yes

The Visual OO Debugger supports debugging of a Java application.

REQ-2.1.1.1

"The aim is to visualize objects graphically."

Achieved: yes

The Visual OO Debugger visualizes objects as nodes and references as edges.

REQ-2.1.1.2

"The aim is to visualize variables graphically."

Achieved: yes

The Visual OO Debugger visualizes variables as nodes. If the variable is a reference to an object, the reference is visualized as an edge. Otherwise, the node also contains the value of the primitive type.

REQ-2.1.2

"The aim is to run a program step by step inside the debugger."

Achieved: yes

VS Code's built-in debugger can be used to progress the code and the visualizations are updated immediately. In addition to that, the user has the option to use the stepper of VOOD to load previous states of the variables.

REQ-2.2.1

"The visual debugger should make it possible to understand how objects change over the course of the program."

Achieved: yes

The Visual OO Debugger updates the visualization on every step of the execution and colorizes the new or changed objects and references in a different colour.

REQ-2.2.2

"The visual debugger should make it possible to understand how variables change over the course of the program."

Achieved: yes

The Visual OO Debugger updates the visualization on every step of the execution and colorizes the new or changed variables and references in a different colour.

REQ-3.1

"Other goals of the project are to make it as easy as possible to get started (setup and import of the program as simple as possible), universal use, e.g., as a Visual Studio Code extension or in the browser (e.g., with GitPod) and usability."

Achieved: yes

The Visual OO Debugger is a VS Code extension which requires no further setup other than installing it. Since GitPod also uses VS Code extensions, it is also possible to use VOOD with GitPod.

REQ-3.2

"Another goal of the project is to make the import of the program as simple as possible."

Achieved: yes

The import of a program depends on the IDE in use. In the case of VS Code, the code needs to be on the file system and can be easily imported using VS Code. In the case of GitPod, a program can be imported by creating a connection to a git repository.

REQ-3.3

"Another goal of the project is to be used as a Visual Studio Code extension."

Achieved: yes

The Visual OO Debugger is a Visual Studio Code extension.

REQ-3.4

“Another goal of the project is to be used in GitPod.”

Achieved: yes

Since GitPod uses the same extensions as VS Code, the Visual OO Debugger can be use with GitPod as well.

REQ-3.5

“Another goal of the project is usability [of the system]”

Achieved: yes

The Visual OO Debugger is kept as simple as possible. The only action required by the user is running a command to open the debugger view. The rest is handled by the IDE.

12.2 Outlook

The development of the Visual OO Debugger will be continued in a bachelor thesis. With the current state of VOOD extension, there is only one type of visualization using vis.js. This library has its limitations, and the initial goal of the bachelor thesis will be to add another type of visualization using the library JointJS. This goal is described in more detail in section “9.3 JointJs will be implemented in the Bachelor Thesis”.

Another possible feature to implement is collapsing and expanding nodes. A wireframe was created for this feature in the section “Collapsed Objects”. The larger the object graph gets, the more beneficial would this feature be.

13 Glossary

Term	Definition
AD	Algorithms and Data structures
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Deployment
FOSS	Free and Open-Source Software
IDE	Integrated Development Environment
MVP	Minimum Viable Product
OO	Object-Oriented
OOP1	Object-Oriented Programming 1 (a course at OST)
OST	Ostschweizer Fachhochschule
PF	Patterns and Frameworks
VOOD	Visual OO Debugger
VS Code	Visual Studio Code

Table 25 Glossary

14 List of Figures

Figure 1 Visual OO Debugger in VS Code 1
Figure 2 Mind map of the requirements 4
Figure 3 Context diagram 13
Figure 4 Class diagram of debug adapter protocol models 14
Figure 5 Class diagram of debug adapter protocol models 16
Figure 6 Component diagram of debug backend 18
Figure 7 Component diagram of webview 19
Figure 8 Class diagram of DebugEventManager 20
Figure 9 Class diagram of PanelView 21
Figure 10 Class diagram of PanelView variables 22
Figure 11 Sequence diagram of the extension's start 23
Figure 12 Sequence diagram of GIF export 24
Figure 13 GitHub workflow overview 25
Figure 14 Wireframe of VS Code Integration 26
Figure 15 Wireframe of primitive types 27
Figure 16 Wireframe of simple object version 1 27
Figure 17 Wireframe of simple object version 2 27
Figure 18 Wireframe of simple object version 3 28
Figure 19 Wireframe of arrays of primitive types and strings 28
Figure 20 Wireframe of object array 29
Figure 21 Wireframe of String to Object Map 29
Figure 22 Wireframe of an Enum 30
Figure 23 Wireframe of Composite Object 30
Figure 24 Wireframe of Collapsed Objects 31
Figure 25 SonarCloud summary 32
Figure 26 Quality tree 36
Figure 27 Risk matrix 42

15List of Tables

Table 1 Requirements 5
Table 2 Quality goals from requirements 6
Table 3 Quality goals from stakeholder analysis 6
Table 4 Stakeholders 7
Table 5 Stakeholder analysis 8
Table 6 Relation map of the stakeholders 9
Table 7 Existing comparable products 10
Table 8 Results of contextual inquiry 11
Table 9 Constraints 12
Table 10 Description of the business context 13
Table 11 Required API's and protocols 14
Table 12 Used VS Code API features 14
Table 13 Solution strategy 15
Table 14 History of decision to develop VOOD as VS Code Extension 33
Table 15 History of decision to use vis.js for visualization 34
Table 16 History of decision to split JointJS off in a separate project 35
Table 17 Quality scenario for REQ-2.2.2 37
Table 18 Quality scenario for REQ-3.2 37
Table 19 Quality scenario for REQ-3.3 38
Table 20 Quality scenario for REQ-3.4 38
Table 21 Quality scenario for REQ-3.5 39
Table 22 Quality scenario for REQ-4 39
Table 23 Identified risks 40
Table 24 Risk probabilities 41
Table 25 Glossary 46

16 Bibliography

arc42. (2021). Retrieved from https://www.arc42.de/

Balsamiq. (2021). Retrieved from https://balsamiq.com/

C4 model. (2021). Retrieved from https://c4model.com/

Debug Adapter Protocol. (2021). Retrieved from https://microsoft.github.io/debug-adapter-protocol/specification

ESLint. (2021). Retrieved from https://eslint.org/

GitHub. (2021). Retrieved from https://github.com/

GitPod. (2021). Retrieved from https://www.gitpod.io/

JointJS. (2021). Retrieved from https://www.jointjs.com/

OST - Ostschweizer Fachhochschule. (2021). Retrieved from https://www.ost.ch/

SonarCloud. (2021). Retrieved from https://sonarcloud.io/

vis.js. (2021). Retrieved from https://almende.github.io/vis/

Visual Studio Code. (2021). Retrieved from https://code.visualstudio.com/

VS Code API. (2021). Retrieved from https://code.visualstudio.com/api/references/vscode-api