

FH Aachen

Fachbereich 5 Elektrotechnik und Informationstechnik

Bachelorarbeit

zur Erlangung des akademischen Grades
B. Sc. im Studiengang Informatik

**Entwicklung einer Lernplattform zur schrittweisen
Visualisierung benutzerprogrammierter Algorithmen**

Gerrit Daniel Weiermann

Matrikelnummer 3518063

Aachen, 07.08.2025

Erstprüfer: Prof. Dr. rer. nat. Andreas Claßen
Zweitprüfer: Michel Erbach (B. Sc.)

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, den

Inhaltsverzeichnis

1 Einleitung	6
2 Ziele der Arbeit	8
2.1 Ziele	8
2.2 Anforderungen	9
3 Grundlagen	11
3.1 Clang	11
3.2 LLDB	11
3.3 Monaco Editor	12
3.4 Docker	12
4 Architektur und technische Konzepte	14
4.1 Webanwendung	18
4.1.1 Grundlegender Aufbau der Webanwendung	18
4.1.2 Aufbau der Problemstellungsseite	21
4.1.3 Visualisierungsansicht	22
4.1.4 Testansicht	35
4.1.5 Codeeditor	37
4.2 Webserver	41
4.3 Analysetool	41
4.3.1 Überblick	42
4.3.2 Verwendung des Tools	43
4.3.2.1 Ausführung der Analyse	43
4.3.2.2 Ausführung von Testfällen	44
4.3.2.3 Fehlerbehandlung	45

Inhaltsverzeichnis

4.3.3	Ablauf der Analyse	45
4.3.4	Preset	50
4.3.5	Code Validierung	52
4.3.6	Collectoren	53
4.3.6.1	Konfiguration eines Collectors	54
4.3.6.2	Arten von Collectoren	54
4.3.6.3	Implementierung von Collectoren	56
4.3.6.4	Verfügbare Collectoren	58
4.3.7	Postprocessoren	64
4.3.7.1	SkipInterSwappingSteps	66
4.3.7.2	KeepTrackOfItems	68
4.3.8	Presets der bereitgestellten Aufgabenstellungen	69
4.4	Executor	69
4.5	Reverse Proxy	72
4.6	Preset-Datenbank	73
4.7	Containerisierung	75
5	Details der technischen Realisierung	76
5.1	Webanwendung / Webserver	76
5.1.1	Editor	76
5.1.2	Speicherung der Kursstruktur	82
5.1.3	Dockerfile	84
5.2	Analysetool	85
5.2.1	Template	85
5.2.2	Aufbau eines Presets	86
5.2.3	Beispiel eines Analyseergebnisses	89
5.2.4	Anbindung von LLDB	91
5.2.5	CLI-Parameter mit JSON	93
5.2.6	Validierung des übermittelten Codes	94
5.2.7	Kompilierung	95
5.2.8	Syntaxfehler	95
5.2.9	Collectoren	97
5.2.10	Postprocessoren	105

5.2.11 Testing	109
5.2.12 Dockerfile	111
5.3 Executor	113
5.3.1 Bereitstellung des HTTP-Endpoints mit Streamingverhalten . .	113
5.3.2 Starten des Analysetools	114
5.3.3 Containerrestriktionen	116
5.3.4 Anfragelogik der Webanwendung	116
5.3.5 Dockerfile	118
5.4 Reverse Proxy	119
5.4.1 Dockerfile	119
5.4.2 Konfiguration von NGINX	119
5.5 Preset-Datenbank	120
5.5.1 Auslieferung eines Presets	120
5.5.2 Auslieferung eines Templates	122
5.5.3 Dockerfile	123
5.6 Docker Compose File	124
5.6.1 Für Produktivumgebungen	124
5.6.2 Für Entwicklungsumgebungen	125
6 Fazit und Ausblick	128
6.1 Fazit	128
6.2 Ausblick	129
Quellenverzeichnis	131
Abbildungsverzeichnis	136

1 Einleitung

In der Lehre – insbesondere in theoretisch geprägten Fächern wie der Informatik – ist es entscheidend, Inhalte möglichst anschaulich und individuell zugeschnitten zu vermitteln. Häufig kommen dabei Visualisierungen in Form von Animationen zum Einsatz, um die Abläufe und Strukturen eines Algorithmus nachvollziehbar darzustellen.

Bei der eigenständigen Entwicklung solcher Algorithmen stehen Personen, die das Programmieren erlernen, vor der Herausforderung, Fehler im eigenen Programm zu identifizieren und zu beheben. Zwar bieten klassische Debugging-Tools die Möglichkeit, den Programmzustand zu einem beliebigen Zeitpunkt einzufrieren und Informationen wie Variablenwerte oder Stackframes anzuzeigen, doch fällt es vielen Lernenden schwer, den Überblick zu behalten und die Übergänge zwischen den einzelnen Debuggingschritten richtig einzuordnen. Eine visuelle Aufbereitung der Programmausführung und der Veränderungen an den zugrunde liegenden Datenstrukturen kann hier entscheidend zur Verständlichkeit und Fehlersuche beitragen.

Die vorliegende Arbeit entwickelt eine prototypische Webanwendung, die Programmieraufgaben aus verschiedenen Problemkategorien im Bereich Algorithmen und Datenstrukturen bereitstellt. Der von Lernenden eingegebene C++-Code wird dabei Schritt für Schritt ausgeführt und in seiner Wirkung visuell aufbereitet, sodass die Verbindung zwischen Code und Datenstrukturzustand nachvollziehbar wird. Ein weiteres Ziel der Implementierung ist eine flexible Architektur, die es erlaubt, mit minimalem Aufwand neue Aufgabenstellungen und Problemkategorien in das System zu integrieren.

Die vorliegende Bachelorarbeit gliedert sich wie folgt:

Kapitel 2 beschreibt die Zielsetzung sowie die funktionalen und nicht-funktionalen Anforderungen an das Gesamtsystem. Kapitel 3 beschreibt die zentralen Technologien, die

für die Umsetzung der Webanwendung zum Einsatz kommen. Dazu zählen Werkzeuge zur Analyse von Code im Backend ebenso wie Technologien zur Orchestrierung und Kapselung der Backend-Komponenten. Kapitel 4 widmet sich der Systemarchitektur. Es werden die einzelnen Komponenten – wie beispielsweise die Webanwendung und die verschiedenen Backend-Module – detailliert beschrieben, ihr jeweiliger Zweck erläutert und ihr Zusammenspiel im Gesamtsystem dargestellt. Kapitel 5 beleuchtet ausgewählte Aspekte der technischen Umsetzung im Detail. Anhand exemplarischer Codeausschnitte und Konfigurationen wird veranschaulicht, wie zentrale Komponenten realisiert wurden. Kapitel 6 zieht ein abschließendes Fazit über die erreichten Ergebnisse der Arbeit und gibt einen Ausblick auf potenzielle Weiterentwicklungen des Systems.

2 Ziele der Arbeit

Im Folgenden werden zunächst die Ziele beschrieben, die mit dieser Arbeit verfolgt werden. Daran anschließend werden die daraus abgeleiteten Anforderungen formuliert, die konkretisieren, auf welche Weise diese Ziele technisch umgesetzt werden sollen.

2.1 Ziele

Das System soll eine browserbasierte Oberfläche bereitstellen, auf der Lernende eine Aufgabenstellung wählen und diese direkt im integrierten Codeeditor bearbeiten können. Ziel ist es, das Verständnis für selbstgeschriebenen C++-Code zu fördern, indem die Ausführung des Programms schrittweise visualisiert wird. Dabei sollen insbesondere die zugrunde liegenden Datenstrukturen grafisch dargestellt und ihre Veränderungen im zeitlichen Ablauf nachvollziehbar gemacht werden. Über die browserbasierte Oberfläche sollen Lernende die Programmausführung interaktiv steuern können: Sie haben die Möglichkeit, einzelne Schritte vor- und zurückzusetzen, eine automatische Wiedergabe mit einstellbarer Geschwindigkeit zu starten, anzuhalten oder zum Anfangszustand zurückzukehren.

Eine Visualisierung besteht beispielsweise bei dem Sortieralgorithmus Bubblesort aus der grafischen Darstellung der zu sortierenden Liste als Auflistung aller Elemente. Beim Wechsel des ausgewählten visualisierten Ausführungsschritts werden Tausch- oder Ersetzungsvorgänge animiert, sodass die Veränderung der Liste im zeitlichen Ablauf nachvollziehbar wird.

Jeder Visualisierungsschritt soll mit denjenigen Codezeilen verknüpft werden, die für den jeweiligen Programmschritt verantwortlich sind. Diese Zeilen sollen im Editor farb-

lich hervorgehoben werden, sodass die Verbindung zwischen Programmverhalten und Quelltext unmittelbar ersichtlich wird. Parallel dazu sollen die aktuellen Werte aller lokalen UmgebungsvARIABLEN dargestellt werden, um Ursache-Wirkungs-Beziehungen klar nachvollziehbar zu machen. Das System soll außerdem die Möglichkeit bieten, den Code der Awendenden automatisch anhand vordefinierter Testfälle auf Korrektheit zu prüfen. Um die Einstiegshürde möglichst gering zu halten, soll weder eine Installation noch eine Registrierung erforderlich sein. Der notwendige Boilerplate-Code soll bereitgestellt werden, sodass sich die Awendenden auf das eigentliche algorithmische Problem konzentrieren können. Schließlich soll das System mehrere Algorithmus- und Problemkategorien unterstützen. Das System soll darauf ausgelegt sein, verschiedene Algorithmus- und Problemkategorien zu unterstützen. Für jede dieser Kategorien sollen spezifische Visualisierungen, Codegerüste und Testfälle zur Verfügung stehen, um unterschiedliche fachliche Kontexte didaktisch angemessen abzudecken.

2.2 Anforderungen

Das Backend führt den von Anwender:innen eingegebenen C++-Code in einer isolierten Umgebung aus, instrumentiert diese Ausführung und protokolliert sämtliche Zustandsänderungen von Datenstrukturen und Variablen. Alle Zwischenschritte werden persistiert, sodass sie sowohl vorwärts als auch rückwärts durchlaufen und jederzeit auf den Ausgangszustand zurückgesetzt werden können.

Jede Aufgabe ist in vier Bereiche gegliedert: „Introduction“ enthält die Aufgabenbeschreibung, „Visualization“ ermöglicht die interaktive Darstellung der Programmausführung, „Tests“ zeigt die Ergebnisse der automatisierten Testfälle und es gibt einen Codeeditor, in dem Awendende ihre Lösung eingeben können.

Die Webseite bietet Schaltflächen für Vor-, Zurück-, Play-/Pause-Funktionen sowie einen Geschwindigkeitsregler; Übergänge zwischen zwei Zuständen – einschließlich Datenstruktur, Editor-Markierungen und Variablenwerte – werden animiert dargestellt. Mehrere Anwender:innen können gleichzeitig mit der Weboberfläche arbeiten, wobei Code, Zustandsdaten und UI-Konfiguration pro Session getrennt gehalten werden.

2.2 Anforderungen

Im Editor wird die zuletzt ausgeführte Codezeile farblich hervorgehoben. Zu jedem Ausführungsschritt zeigt eine Variablenliste alle aktuell verfügbaren lokalen Variablen an. Bei Zeigern wird auf die Darstellung der tatsächlichen Adresswerte verzichtet, um die Lesbarkeit zu erhöhen. Referenzvariablen zeigen stattdessen direkt den Wert der referenzierten Variablen an.

Die Testfälle werden im Backend ebenfalls in einer isolierten Umgebung ausgeführt. Dabei wird der von Anwender:innen eingegebene Code getestet und das Ergebnis jedes Testfalls protokolliert, einschließlich der Information, ob er erfolgreich war oder fehlgeschlagen ist.

Die Anwendung ist vollständig webbasiert und erfordert von Seiten der Anwender:innen weder eine Installation noch eine Registrierung. Ein integrierter Editor stellt Syntaxhervorhebung, Auto-Indent und gängige Shortcuts bereit und erlaubt das Sperren nicht zu bearbeitender Codeblöcke, um die vorgegebene Struktur zu bewahren. Ein einfacher Klick auf eine entsprechende Schaltfläche kompiliert, analysiert und testet den Code; Compilerfehler werden dabei direkt im Editor angezeigt.

Aufgaben sind nach Problemkategorien organisiert, welche wiederum einem jeweiligen Kurs zugeordnet sind. Mit der Auswahl einer Programmieraufgabe wird ein passendes Codegerüst geladen, die Analysepipeline entsprechend konfiguriert und eine domänen-spezifische Visualisierung der relevanten Datenstrukturen aktiviert. Unterstützt werden die Problemkategorien „einfache Sortierverfahren“ mit Bubblesort, „rekursive Sortierverfahren“ mit Quicksort sowie „binäre Suchbäume“ mit einer Einfüge-Aufgabe.

3 Grundlagen

Im Folgenden werden die zentralen Technologien vorgestellt, die der in dieser Arbeit entwickelten Anwendung zugrunde liegen. Dabei wird jeweils kurz erläutert, welche Rolle sie im Gesamtsystem einnehmen und weshalb sie für die Umsetzung geeignet erscheinen.

3.1 Clang

Clang [1], [2] ist ein Compiler-Frontend für die Programmiersprachen C, C++ und weitere Sprachen des C-Sprachstandards. Es ist Teil des *LLVM*-Projekts [3], [4] und bietet eine modulare, plattformunabhängige und performante Alternative zu traditionellen Compilern wie *GCC* [5], [6]. Im Rahmen des zu entwickelnden Systems wird *Clang* verwendet, um den von den Anwender:innen eingereichten Code zu kompilieren.

In Kombination mit dem Debugger *LLDB* [7]–[9], der nahtlos mit *Clang* zusammenarbeitet, lässt sich der eingereichte C++-Code automatisiert debuggen. Dies ermöglicht eine programmgesteuerte Auswertung von Code ohne manuelle Eingriffe und bildet die Grundlage für eine automatisierte Verarbeitung innerhalb des Systems.

3.2 LLDB

LLDB ist der Debugger des *LLVM*-Projekts. Er unterstützt unter anderem die Sprachen C, C++ und Objective-C und zeichnet sich durch seine enge Integration mit *Clang* aus.

3.4 Docker

Ein wesentliches Merkmal von *LLDB* ist die verfügbare Programmierschnittstelle in *Python* [10], [11]. Über dieses Interface kann der Debugger vollständig gesteuert und in eigene Anwendungen integriert werden. Dies ermöglicht eine automatisierte Steuerung von Breakpoints, das Auslesen von Speicherinhalten sowie die Ausführung von Programmcode, ohne dass eine direkte Interaktion durch Anwender:innen erforderlich ist. Diese Automatisierung bildet eine zentrale Grundlage für die in dieser Arbeit vorgestellte Analyse des C++-Codes.

3.3 Monaco Editor

Bei dem in der Webanwendung verwendeten Codeeditor handelt es sich um den *Monaco Editor* [12], [13], der auch in Visual Studio Code [14], [15] integriert ist. Dort werden die Anwender:innen ihre Lösung in Form von C++-Code eingeben. Für die Programmiersprache C++ bietet er integriertes Syntax-Highlighting und erlaubt darüber hinaus das gezielte Hervorheben einzelner Codezeilen, was es später für die Visualisierung von Bedeutung sein wird.

3.4 Docker

Docker [16] ist eine Plattform zur Containerisierung, die es ermöglicht, Anwendungen mitsamt ihren Abhängigkeiten isoliert und reproduzierbar auszuführen. Im Gegensatz zu virtuellen Maschinen teilen sich *Docker*-Container den Kernel des Hostsystems, was sie besonders ressourcenschonend macht. Die Anwendungen laufen in abgeschotteten Umgebungen, was eine sichere Trennung zwischen einzelnen Prozessen und Systemkomponenten gewährleistet.

Ein zentraler Vorteil von *Docker* liegt in der Modularisierung: Jede Komponente eines Systems kann in einem eigenen Container betrieben werden. Durch dieses Sandboxing-Konzept kann der von Anwender:innen eingereichte C++-Code kontrolliert und sicher ausgeführt werden, ohne das Hostsystem zu gefährden. Gleichzeitig ermöglicht die

einheitliche Umgebung eine konsistente Ausführung unabhängig von der zugrunde liegenden Infrastruktur.

4 Architektur und technische Konzepte

Im Folgenden werden sämtliche Komponenten des Frontends und Backends vorgestellt. Dabei wird jeweils erläutert, welche Rolle sie im Gesamtsystem einnehmen, wie sie miteinander interagieren und wie ihre Funktionsweise im Detail ausgestaltet ist.

Das System besteht aus einem zentralen Backend, sowie einer Webanwendung, die jeweils im Browser der Anwendenden ausgeführt wird. Das Backend setzt sich dabei zusammen aus den Teilsystemen Webserver, Analysetool, Executor, Reverse Proxy und Preset-Datenbank. *Abbildung 4.1* zeigt einen Überblick über den Aufbau des Systems.

Die Webanwendung als Frontend bietet eine Auswahl an Programmieraufgaben zur Implementierung verschiedener Algorithmen. Anwender:innen wählen zunächst einen Kurs, anschließend eine Problemkategorie und schließlich eine konkrete Aufgabe aus. Jede Aufgabenstellung kann direkt in einem eingebetteten Codeeditor bearbeitet werden.

Nachdem Anwender:innen ihren C++-Code in den Editor eingegeben haben, können sie die Analyse und das Testing starten. Die Webanwendung überträgt den Code an das Backend, wo er kompiliert, ausgeführt und während der Ausführung überwacht, sowie mit Testfällen getestet wird. Während der Ausführung werden gezielt die für die später erfolgende Visualisierung relevanten Programmschritte aufgezeichnet. Dazu gehören insbesondere Zustandsänderungen der zu überwachenden Datenstruktur sowie zentrale Abläufe des Algorithmus, wie z. B. der Funktionsaufrufe. Eine genauere Erläuterung der Auswahlkriterien, welche Schritte genau aufgezeichnet werden, erfolgt in Abschnitt 4.3.6.

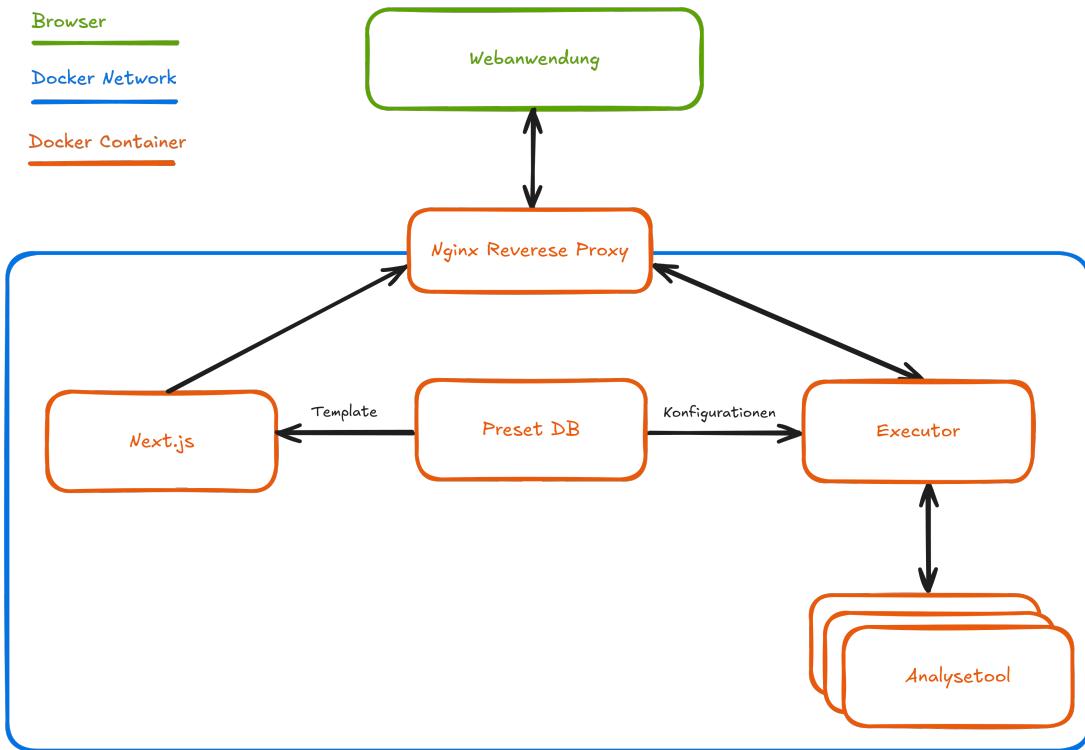


Abbildung 4.1: Gesamtüberblick über den Aufbau des Systems.

Die Analyseergebnisse werden anschließend an die Webanwendung zurückgesendet und in einer speziellen Visualisierungsansicht dargestellt. Diese zeigt die schrittweise Ausführung des eingereichten Codes auf Basis der zuvor ermittelten Schritte. Anwender:innen können die Visualisierung interaktiv steuern und einzelne Schritte des Algorithmus vor- und zurückspringen. Zustandsänderungen und relevante Abläufe werden durch Animationen anschaulich dargestellt. Auch die Testergebnisse werden an die Webanwendung übermittelt, die in einer Testansicht betrachtet werden können. Eine detaillierte Beschreibung der Webanwendung folgt in Unterkapitel 4.1.

Das Backend übernimmt sowohl das Hosting der Webanwendung als auch die im vorherigen Abschnitt beschriebene Analyse des eingereichten C++-Codes. Es besteht aus mehreren Teilsystemen: Der Webserver stellt die grafische Oberfläche der Webanwendung bereit. Das Analysetool führt die Analyse des C++-Codes durch, indem es relevante Ausführungsschritte und Zustandsänderungen erfasst. Das Analysetool besitzt zudem die Fähigkeit, den C++-Code mithilfe von Testfällen zu testen. Der Executor

verwaltet Analyseaufträge, indem er Anfragen der Webanwendung entgegennimmt, an das Analysetool weiterleitet und dessen Statusmeldungen, sowie Ergebnisse zurück an die Webanwendung sendet. Ein vorgeschalteter Reverse Proxy verbindet sowohl den Webserver als auch den Executor mit dem externen Netzwerk. Ergänzt wird das Backend durch eine Preset-Datenbank, die das Codegerüst, die Analyseparameter und die Testcases von Aufgabenstellungen in Form eines sogenannten Presets bereitstellt.

Der Webserver stellt die Webanwendung über den vorgeschalteten Reverse Proxy den Anwender:innen zur Verfügung. Dabei kommt das JavaScript-Framework Next.js [17], [18] zum Einsatz, das während der Entwicklung mit dem integrierten Entwicklungsserver und im Produktivbetrieb mit dem optimierten Produktionsserver betrieben wird. Das Framework läuft auf der *Node.js*-Runtime [19], [20]. Unterkapitel 4.2 wird genauer auf den Webserver eingehen.

Das Analysetool ist eine Kommandozeilenanwendung, die den von Anwender:innen eingegebenen C++-Code kompiliert und das daraus erzeugte Executable kontrolliert ausführt, um Informationen über den Ablauf des Programms zu gewinnen. Dabei werden insbesondere Änderungen von Variablenwerten und Zustände der verwendeten Datenstrukturen erfasst. Technisch erfolgt die Analyse über ein kontrolliertes Debugging des Executables. Während dieses Prozesses sammelt das Tool Informationen zu Laufzeitwerten, Datenstrukturzuständen und relevanten Abläufen innerhalb des Algorithmus. Über ein parametrisierbares Preset lässt sich festlegen, welche Daten im Rahmen der Analyse für einen bestimmten Algorithmus erfasst werden sollen. Die gewonnenen Analyseinformationen werden von dem Tool gesammelt und nach der Ausführung in Form von JSON-Nachrichten auf der Standardausgabe ausgegeben, ergänzt durch Statusmeldungen zum Fortschritt der Analyse, während die Analyse im Gange ist. Während der Ausführung ergänzen zusätzliche Statusmeldungen in Form von JSON-Nachrichten die Ausgabe und geben dem Client fortlaufend Rückmeldung über den aktuellen Stand des Analyseprozesses. Das Analysetool kann neben der Durchführung von Analysen auch zur Ausführung vordefinierter Testfälle verwendet werden. In diesem Modus wird der übergebene C++-Code zunächst wie gewohnt kompiliert und anschließend mit den definierten Testfällen überprüft. Die Ergebnisse dieser Tests werden am Ende in strukturierter Form als JSON-Nachrichten auf der Standardausgabe ausgegeben. Auch hier

melden Statusmeldungen den aktuellen Fortschritt des laufenden Prozesses. Eine detaillierte Beschreibung des Analysetools folgt in Unterkapitel 4.3.

Der Executor dient als Schnittstelle zwischen der Webanwendung und dem Analysetool zur Auswertung des im Editor von Anwendenden geschriebenen C++-Codes. Über einen bereitgestellten HTTP-Endpunkt kann die Webanwendung Analyseaufträge an den Executor übermitteln. Für jeden Auftrag startet der Executor eine separate, containerisierte Instanz des Analysetools und übernimmt deren Verwaltung. Er ist in der Lage, mehrere Analyseprozesse parallel zu handhaben, übermittelt dem Client während der Analyse fortlaufend Statusmeldungen und sendet nach Abschluss das Analyseergebnis zurück. Eine detaillierte Beschreibung des Executors erfolgt in Unterkapitel 4.4.

Der Reverse Proxy ist die einzige Komponente des Systems, die direkt mit dem externen Netzwerk verbunden ist. Er fungiert als zentrales Gateway für sowohl den Webserver als auch den Executor. Implementiert wird der Reverse Proxy mit *NGINX* [21], [22]. Ziel dieser Architektur ist es, den gesamten externen Zugriff über eine einzige Schnittstelle zu bündeln, wodurch sich der Verwaltungsaufwand reduziert: Es wird nur ein Netzwerkport benötigt, und durch die Nutzung eines gemeinsamen Hosts lassen sich typische Probleme mit Cross-Origin Resource Sharing (CORS) vermeiden. Zudem ermöglicht dieser Aufbau perspektivisch eine einfache Erweiterung um Load-Balancing-Funktionalität bzw. den Austausch des Reverse Proxys durch einen dedizierten Load Balancer, ohne dass Änderungen an den übrigen Teilsystemen erforderlich sind.

Mit einer Erweiterung der *NGINX* Konfiguration der Reverse Proxy Komponente oder eines weiteren vorangeschalteten Reverse Proxys wäre es möglich, die Webanwendung auch über HTTPS ausliefern zu lassen, um die Kommunikation zwischen dem Client und dem Server zu verschlüsseln. Auf eine konkrete Implementierung der HTTPS-Verschlüsselung wird im Folgenden jedoch nicht weiter eingegangen. Unterkapitel 4.5 wird genauer auf den Reverse Proxy eingehen.

Um die sogenannten Presets – also das Codegerüst sowie die Konfigurationen für Analyse und Testfälle zu jeder Aufgabenstellung – zentral und unabhängig von anderen Komponenten zu verwalten, kommt eine eigene Preset-Datenbank zum Einsatz. Diese hält die Presets in Form einer Dateistruktur vor und macht sie über einen HTTP-Endpunkt

4.1 Webanwendung

für den Webserver und den Executor zugänglich. Der Endpunkt ist ausschließlich für interne Zwecke vorgesehen und nicht aus dem externen Netzwerk erreichbar. Unterkapitel 4.6 wird genauer auf die Preset-Datenbank eingehen.

Die einzelnen Teilsysteme des Backends sind in Docker Containern implementiert, um eine einfache, automatisierte Installation und eine konsistente Ausführungsumgebung zu gewährleisten. Die Containerisierung ermöglicht es zudem, das Analysetool in einer isolierten Sandbox-Umgebung auszuführen und gezielt Restriktionen zu definieren, die der Erhöhung der Systemsicherheit dienen. Mithilfe von Docker Compose erfolgt die Orchestrierung der Container. Unterkapitel 4.7 wird genauer auf die Containerisierung eingehen.

4.1 Webanwendung

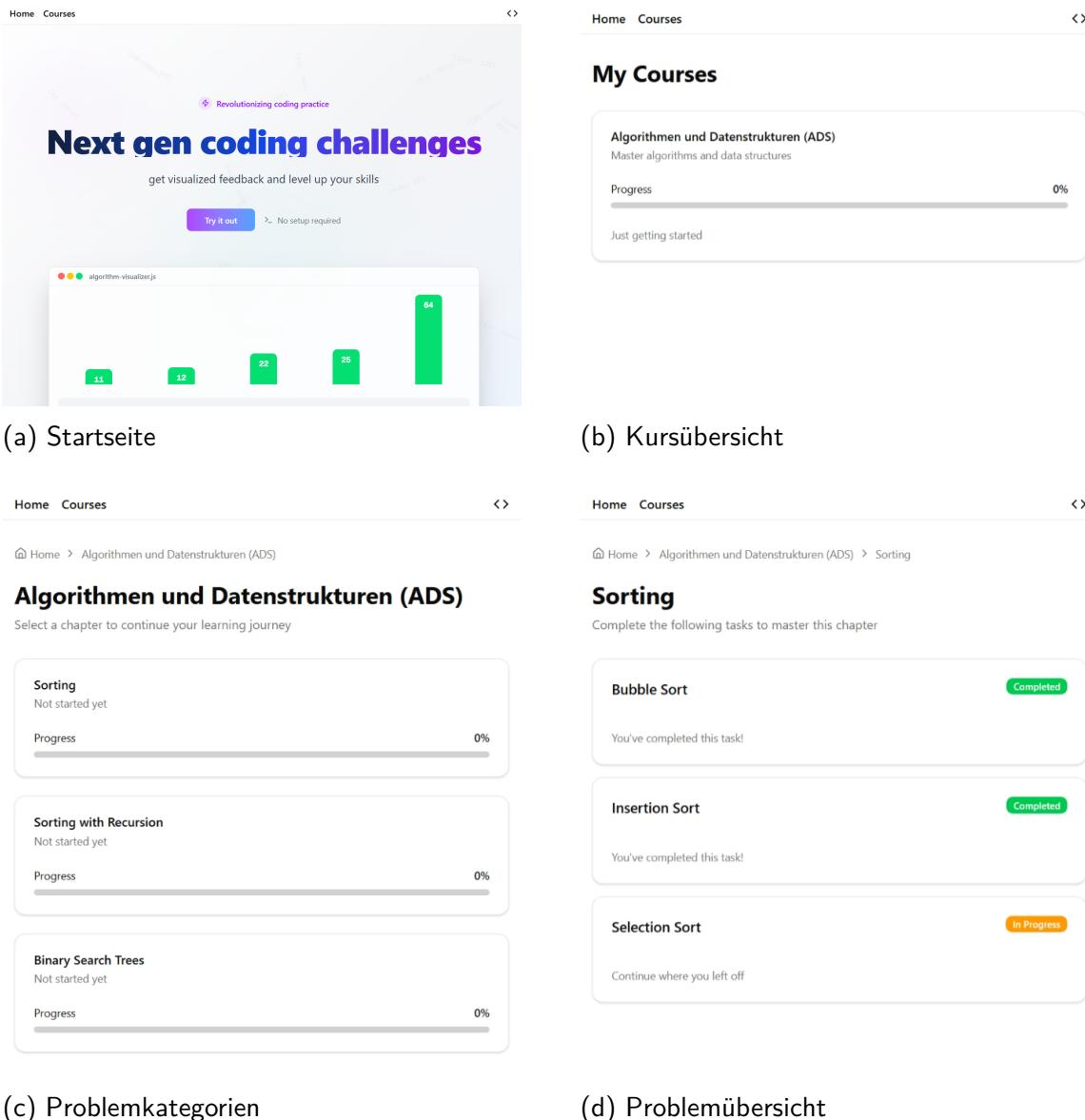
Die Webanwendung wurde mit dem Framework *Next.js* [17], [18] umgesetzt, das auf *React.js* [23], [24] basiert und dieses um Funktionen wie serverseitiges Rendering, statische Seitengenerierung und ein integriertes Routingsystem erweitert. Für die Gestaltung der grafischen Oberfläche kamen die Component Library *shadcn/ui* [25], [26] sowie das Framework *Tailwind CSS* [27], [28] zum Einsatz. Diese Kombination ermöglicht eine konsistente, modulare und zugleich flexibel anpassbare Umsetzung des Designs. Darüber hinaus profitieren die Komponenten von der Wiederverwendbarkeit und der klaren Trennung von Logik und Darstellung.

Mehrere Screens der Webanwendung wurden rein gestalterisch mit dem KI-Tool *v0* [29] entworfen. Dabei entstanden verschiedene UI-Komponenten, wie die Homepage in *Abbildung 4.2a* und die Testing-Ansicht in *Abbildung 4.13*.

4.1.1 Grundlegender Aufbau der Webanwendung

Die Screenshots der *Abbildung 4.2* verdeutlichen das Konzept der Kursplattform. Die Startseite (siehe *Abbildung 4.2a*) dient als Einstiegspunkt für Anwender:innen und soll durch eine übersichtliche Darstellung zum Entdecken und Nutzen der Lerninhalte

motivieren. Auf der Kursübersichtsseite (siehe Abbildung 4.2b) werden alle verfügbaren Kurse aufgelistet, aus denen Anwender:innen einen Kurs auswählen können. Im Rahmen dieser Arbeit wurde der Kurs „Algorithmen und Datenstrukturen (ADS)“ exemplarisch umgesetzt.



The figure consists of four screenshots labeled (a) through (d), illustrating a web-based learning platform for algorithms and data structures (ADS).

- (a) Startseite:** Shows the homepage with a banner for "Next gen coding challenges" and a "Try it out" button. Below is a section titled "algorithm-visualizer.js" showing a visual representation of an algorithm with numbered boxes (11, 12, 22, 25, 64).
- (b) Kursübersicht:** Shows the "My Courses" page for "Algorithmen und Datenstrukturen (ADS)". It displays a progress bar at 0% and the status "Just getting started".
- (c) Problemkategorien:** Shows the "Algorithmen und Datenstrukturen (ADS)" course page. It lists three categories: "Sorting" (Not started yet, 0% progress), "Sorting with Recursion" (Not started yet, 0% progress), and "Binary Search Trees" (Not started yet, 0% progress).
- (d) Problemübersicht:** Shows the "Sorting" chapter page under the ADS course. It lists three sorting algorithms: "Bubble Sort" (Completed, You've completed this task!), "Insertion Sort" (Completed, You've completed this task!), and "Selection Sort" (In Progress, Continue where you left off).

Abbildung 4.2: Screenshots der grafischen Oberfläche

4.1 Webanwendung

Nach Auswahl eines Kurses gelangen Anwender:innen auf die Problemkategorieseite (siehe *Abbildung 4.2c*), auf der sie eine Themenkategorie innerhalb des Kurses auswählen können. Für den ADS-Kurs wurden im Prototyp die drei Kategorien „Sorting“, „Sorting with recursion“ sowie „Binary Search Trees“ implementiert.

Die jeweils zugehörigen Aufgabenstellungen können anschließend auf der Problemübersichtsseite (siehe *Abbildung 4.2d*) ausgewählt werden. In der umgesetzten Version sind dies konkret: *Bubblesort* für die Kategorie „Sorting“, *Quicksort* für „Sorting with recursion“ sowie *Einfügen in einen Binary Search Tree* für „Binary Search Trees“.

Die Informationen zu den Kursen, den Kategorien sowie die Namen und Aufgabenbeschreibungen der Problemstellungen sind in der Webanwendung hartkodiert hinterlegt. Das zugehörige Codegerüst hingegen wird vom Webserver dynamisch aus der Preset-Datenbank geladen und mithilfe von Server Side Rendering beim Aufruf der Seite ausgeliefert.

Die Fortschrittsanzeigen „Progress“ in den Abbildungen 4.2b, 4.2c und 4.2d sowie die Badges „Not Started“ in *Abbildung 4.2d* dienen derzeit lediglich als Platzhalter. In einer zukünftigen Version der Plattform könnten diese Elemente mit funktionaler Logik hinterlegt werden. Ziel solcher Fortschrittsanzeigen ist es, den Anwender:innen dabei zu helfen, ihren Bearbeitungsstand im Kursverlauf nachzuvollziehen und den Überblick über bereits begonnene oder abgeschlossene Aufgaben zu behalten.

4.1.2 Aufbau der Problemstellungsseite

The screenshot shows a web-based application for solving algorithmic problems. On the left, there's a navigation bar with 'Home' and 'Courses', and a breadcrumb trail: Home > Algorithmen und Datenstrukturen (ADS) > Sorting > Bubble Sort. Below this is a section titled 'Bubble Sort' with three tabs: 'Instructions' (selected), 'Visualization', and 'Tests'. The 'Instructions' tab contains text explaining the Bubble Sort algorithm and a numbered list of steps. The 'Tests' tab is currently empty. On the right is a code editor with a dark theme. It displays a C++ code template:

```

1 > #pragma region prototypes ...
6
7 void sort(int* arr, int n) {
8     // todo: add your code here
9 }
10
11 > #pragma region custom_functions ...
21
22

```

A 'Try it out!' button is located at the bottom right of the code editor.

Abbildung 4.3: Initiale Seite einer Problemstellung

Nach der Auswahl einer Problemstellung gelangen Anwender:innen auf eine Seite, auf der die Aufgabe bearbeitet werden kann (siehe Abbildung 4.3). Auf der linken Seite befindet sich die Aufgabenstellung, die den Anwender:innen erklärt, wie der Algorithmus funktioniert und wie er implementiert werden soll.

Auf der rechten Seite der grafischen Oberfläche befindet sich der Codeeditor, in dem die Anwender:innen ihren C++-Code eingeben können. Der Editor enthält ein vorgegebenes Grundgerüst, das die grundlegende Struktur des Programms vorgibt und Anwender:innen dazu anleitet, ihre Lösung innerhalb eines definierten Coderahmens zu implementieren. Eine detaillierte Beschreibung der Funktionsweise und Realisierung des Editors findet sich in Unterkapitel 4.1.5.

Neben der Aufgabenstellung im Reiter „Instructions“ stehen die Reiter „Visualization“ und „Tests“ zur Verfügung. Im Reiter „Visualization“ findet die schrittweise Visualisierung des eingegebenen Codes statt, während im Reiter „Tests“ die Ergebnisse von Testfällen aufgelistet werden, die auf den eingegebenen Code ausgeführt wurden.

4.1 Webanwendung

Nachdem Anwender:innen eine Code-Lösung eingegeben haben, können sie über den Button „Try it out!“ eine Analyse oder einen Testlauf starten. Sowohl für die Analyse als auch für das Testing wird jeweils eine HTTP-Anfrage an den Executor im Backend gesendet. Diese Anfrage enthält eine eindeutige Preset-ID sowie den eingegebenen C++-Code. Das durch die Preset-ID referenzierte Preset umfasst vorkonfigurierte Einstellungen für den Analyseprozess sowie definierte Testfälle, die spezifisch auf die jeweilige Aufgabenstellung zugeschnitten sind. Anhand dieser Konfiguration erkennt das Analysetool, welche Daten während der Programmausführung gesammelt und für die Visualisierung aufbereitet werden sollen beziehungsweise welche Testfälle zur Überprüfung der Programmlogik ausgeführt werden sollen.

Die Reiter „Visualization“ und „Tests“ zeigen daraufhin einen Ladeindikator neben ihrem Namen. Wird einer der beiden Reiter geöffnet, erscheint in der jeweiligen Ansicht ein zentraler Ladespinner mit ergänzenden Statusmeldungen, die den Fortschritt der Analyse bzw. des Testings rückmeldet. Sobald die Analyse bzw. der Testlauf abgeschlossen ist, werden die vom Backend empfangenen Daten verarbeitet und die entsprechende Ansicht dargestellt. Während des gesamten Analyseprozesses bleibt die Netzwerkverbindung bestehen. Die in den Ansichten angezeigten Statusmeldungen werden über diese bestehende HTTP-Verbindung in Echtzeit vom Backend übermittelt. Dabei kommt das sogenannte „chunked transfer encoding“ zum Einsatz, das eine unidirektionale und semi-echtzeitfähige Datenübertragung ermöglicht. Nach dem Erhalt des Analyse- bzw. Testergebnisses ist seitens der Webanwendung keine weitere Kommunikation mit dem Backend erforderlich, da sämtliche Analyseschritte bereits vollständig in den Analyseergebnissen enthalten sind und lokal visualisiert werden können, weshalb die Verbindung zum Backend nach Erhalt geschlossen wird.

4.1.3 Visualisierungsansicht

Die Visualisierungsansicht stellt eine Oberfläche zur Verfügung, mit der das vom Backend bereitgestellte Analyseergebnis interaktiv erkundet werden kann. Das Analyseergebnis besteht aus einer Liste von Analyseschritten, wobei jeder Schritt entweder einen neuen Zustand der beobachteten Datenstruktur beschreibt oder einen Programmablauf

wie beispielsweise einen Funktionsaufruf repräsentiert. Zusätzlich enthält jeder Analyseschritt Informationen über den aktuellen Zustand der lokalen Variablen sowie die zum jeweiligen Schritt gehörende Codezeile. Mithilfe eines sogenannten Postprocessors ist es möglich, dass mehrere Codezeilen zu einem Analyseschritt zusammengefasst werden können (siehe 4.3.7.1). Jeder dieser Schritte wird in der Visualisierung als eigenständiger Zustand dargestellt, durch den mittels entsprechender Bedienelemente navigiert werden kann. Ein Beispiel eines Analyseergebnisses ist in Abschnitt 5.2.3 zu finden.

Für jeden Typ von Aufgabenstellung wurde eine eigenständige Visualisierung entwickelt und fest in die Webanwendung integriert. Diese Visualisierungen sind jeweils auf die spezifischen Anforderungen und das Verhalten des zugrunde liegenden Algorithmus zugeschnitten. Der in der Visualisierung dargestellte Programmablauf basiert auf dem vom Analysetool empfangenen Analyseergebnis. Basierend auf der ausgewählten Aufgabenstellung lädt die Problemstellungsseite die passende Visualisierung und zeigt sie im Reiter „Visualization“ an. So können beispielsweise Sortieralgorithmen wie Bubblesort, Insertionsort und Selectionsort mit demselben Visualisierungstyp dargestellt werden, da sie ähnliche Ablaufmuster aufweisen. Im Gegensatz dazu eignet sich die Visualisierung des Quicksorts nur eingeschränkt für andere rekursive Sortierverfahren wie Merge Sort, da deren Funktionsweise zu stark abweicht und die Implementierung der Visualisierungen nicht flexibel genug ist, mehrere Verfahren gleichzeitig visualisieren zu können. Um dem Vorgehen des Mergesorts gerecht zu werden, müsste eine eigenständige Visualisierung entwickelt werden.

Für jeden visualisierten Schritt werden die beteiligten Codezeilen im Codeeditor rechts hervorgehoben. Dabei ist zu beachten, dass – anders als beim klassischen Debugging – die im Analyseschritt angegebenen Codezeilen nicht unmittelbar vor ihrer Ausführung stehen, sondern bereits ausgeführt wurden. Dies dient der klareren Veranschaulichung, an welcher Stelle im Quellcode eine bestimmte Zustandsänderung tatsächlich ausgelöst wurde. Würde stattdessen – wie beim klassischen Debugging üblich – stets die noch auszuführende Zeile markiert, ließe sich der Zusammenhang zwischen Codestelle und ausgelöstem Effekt weniger eindeutig nachvollziehen.

4.1 Webanwendung

Bestandteile der Visualisierungsansicht

Im Folgenden werden die einzelnen Komponenten der Visualisierungsansicht näher erläutert. Dazu gehören die Variablenliste, die dargestellte und animierte Datenstruktur sowie das zugehörige Bedienfeld. Zudem wird dargestellt, wie das Verhalten der Visualisierung aus dem Analyseergebnis abgeleitet wird und in welcher Weise eine Interaktion mit dem Codeeditor erfolgt.

The screenshot shows a web-based visualization interface for the 'Bubble Sort' algorithm. At the top, there are navigation links: 'Home', 'Courses', 'Algorithmen und Datenstrukturen (ADS)', 'Sorting', and 'Bubble Sort'. Below this is a header with tabs: 'Instructions' (selected), 'Visualization', and 'Tests'. The 'Instructions' tab displays the C code for bubble sort:1 > #pragma region prototypes...
6
7 void sort(int* arr, int n) {
8 for (int i = 0; i < n - 1; ++i) {
9 for (int j = 0; j < n - i - 1; ++j) {
10 if (arr[j] > arr[j + 1]) {
11 int temp = arr[j];
12 arr[j] = arr[j + 1];
13 arr[j + 1] = temp;
14 }
15 }
16 }
17 }
18
19 > #pragma region custom_functions...
29
30Above the code, there is a table titled 'Variable' with two rows: 'Variable' and 'Value'. The 'Value' row contains seven boxes representing the array elements: 64, 34, 25, 12, 22, 11, and 90. To the right of the code editor is a 'Try it out!' button. At the bottom, there is a navigation bar with arrows and the text 'Step 1/15'.

Abbildung 4.4: Initiale Visualisierungsansicht

Die Visualisierungsansicht ist in *Abbildung 4.4* dargestellt und wie folgt aufgebaut: Eine zweizeilige Tabelle zeigt die lokalen Variablen an, die dem Debugger während der jeweiligen Analyseschritte zur Verfügung standen. Die erste Zeile der Tabelle zeigt die Namen der Variablen, die zweite Zeile ihre jeweiligen Werte. Zeigervariablen werden mit einem „-“ gekennzeichnet, da der konkrete Adresswert in den meisten Fällen keinen didaktischen Mehrwert für Anwender:innen bietet. Diese Darstellung trägt zu einer kompakteren und besser lesbaren Tabelle bei. Die Informationen zur Befüllung der Tabelle kommen von jeweiligen Analyseschritten des Analyseergebnisses. Es stellt eine Key-Value-Struktur bereit, in der jede lokale Variable durch ihren Namen als Schlüssel identifiziert wird. Der zugehörige Wert umfasst den aktuellen Wert der Variable als

String – im Fall einer Referenz den Wert der referenzierten Variable –, den ausgeschriebenen Datentyp als String sowie zusätzliche Flags, die kennzeichnen, ob es sich bei der Variable um einen Pointer oder eine Referenz handelt.

Unterhalb der Tabelle befindet sich die eigentliche Visualisierung des jeweiligen Algorithmus. Im Fall des Bubblesort-Algorithmus zeigt sie eine Reihe von Kästchen, die die Elemente des zu sortierenden Arrays repräsentieren. Beim Durchlaufen der einzelnen Animationsschritte werden die Kästchen entsprechend dem Verhalten des im Backend ausgeführten Codes neu angeordnet, wobei sowohl Vertauschungen als auch Ersetzungen von Elementen animiert dargestellt werden. Die Visualisierung bezieht die stattgefundenen Ereignisse der Arrayveränderungen aus den jeweiligen Analyseschritten der zuvor beschriebenen Analyseergebnis-Datenstruktur, die vom Backend übermittelt wurde. Das im Analyseschritt mitgegebene Ereignis gibt an, ob im aktuellen Schritt eine Vertauschung zweier Elemente oder eine Ersetzung eines einzelnen Elements stattgefunden hat. Darüber hinaus wird beschrieben, welche Positionen von der Vertauschung betroffen waren bzw. welcher Wert an welcher Position durch welchen neuen Wert ersetzt wurde.

Unterhalb der Visualisierung befinden sich Steuerelemente, die es den Anwender:innen ermöglichen, gezielt zwischen den einzelnen Visualisierungsschritten zu navigieren. Mit dem Symbol „>“ kann zum nächsten Schritt gesprungen werden, mit dem Symbol „<“ zum vorherigen. Die Webanwendung hält die empfangene Ausführungsschritt-Datenstruktur im Arbeitsspeicher vor und nutzt sie als Grundlage zur Darstellung der einzelnen Visualisierungsschritte. Dabei wird jedem Visualisierungsschritt der jeweilige Analyseschritt aus dem Analyseergebnis in aufsteigender Reihenfolge zugeordnet. Dabei entspricht jeder Analyseschritt exakt einem Animationsschritt. Die aktuelle Schrittposition ergibt sich aus dem Index des gerade angezeigten Eintrags in dieser Liste, wobei intern ein Offset von 1 berücksichtigt wird, um eine nutzerfreundliche Darstellung (beginnend bei Schritt 1) zu ermöglichen. Beim Vor- oder Zurückspringen verändert sich lediglich der Schrittindex, woraufhin der Visualisierungszustand basierend auf dem entsprechenden Analyseschritt neu gerendert wird. Für die animierte Darstellung der Übergänge zwischen den einzelnen Schritten wird die Bibliothek *Motion* [30] verwendet. Diese ermöglicht es, DOM-Änderungen automatisch visuell ansprechend zu animieren,

4.1 Webanwendung

wodurch ein flüssiges und intuitives Nutzererlebnis geschaffen wird.

Zusätzlich stehen weitere Steuerungsmöglichkeiten zur Verfügung: Mit dem Play-Symbol kann eine automatische Wiedergabe der Schritte gestartet werden, bei der die Visualisierung Schritt für Schritt durchlaufen wird. Das Pause-Symbol stoppt diese automatische Abfolge. Über das Dropdown-Menü „1x“ können Anwender:innen die Geschwindigkeit der automatischen Wiedergabe anpassen und so das Tempo der Animation regulieren.

Der in *Abbildung 4.4* gezeigte Zustand entspricht dem initialen Zustand der Datenstruktur, der vom Backend als erster Analyseschritt übermittelt wurde.

The screenshot shows a web-based programming environment for visualizing algorithms. At the top, there's a navigation bar with 'Home' and 'Courses'. Below it, a breadcrumb trail shows 'Home > Algorithmen und Datenstrukturen (ADS) > Sorting > Bubble Sort'. The main area is titled 'Bubble Sort' and contains three tabs: 'Instructions', 'Visualization', and 'Tests'. The 'Instructions' tab is active, showing a table with variables and their values:

Variable	arr	n	i	j	temp
Value	-	7	1	2	34

Below the table is a visualization area showing seven boxes representing array elements: 25, 12, 22, 34, 11, 64, and 90. The boxes 22 and 34 are highlighted with dashed outlines, indicating they are currently being compared or swapped. To the right of the visualization is a code editor window displaying the C code for the bubble sort algorithm:

```
1 > #pragma region prototypes...
6
7 void sort(int* arr, int n) {
8     for (int i = 0; i < n - 1; ++i) {
9         for (int j = 0; j < n - i - 1; ++j) {
10            if (arr[j] > arr[j + 1]) {
11                int temp = arr[j];
12                arr[j] = arr[j + 1];
13                arr[j + 1] = temp;
14            }
15        }
16    }
17 }
18
19 > #pragma region custom_functions...
29
30
```

At the bottom of the visualization area, there are navigation buttons for 'Step 9/15' and a play/pause button labeled '1x'. A 'Try it out!' button is located in the bottom right corner of the code editor.

Abbildung 4.5: Initiale Seite einer Problemstellung

In *Abbildung 4.5* ist ein Programmzustand nach mehreren Ausführungsschritten dargestellt. Im gezeigten Beispiel wurden im aktuellen Schritt die Elemente 22 und 34 miteinander vertauscht. Das bedeutet: Sobald der entsprechende Schritt über das Bedienfeld ausgewählt oder im Rahmen der automatischen Wiedergabe erreicht wurde, wird eine kurzweilige Animation abgespielt, die diesen Tausch visuell nachvollziehbar macht.

Blau eingefärbt sind jene Elemente der Datenstruktur, die im aktuell ausgewählten Schritt verändert wurden. Um diesen Effekt zu erzielen, werden die Analyseschritte von der Webanwendung vorverarbeitet. Dabei durchläuft das System jeden einzelnen Schritt und annotiert ihn mit zusätzlichen CSS-Klassen, die später in der Visualisierung verwendet werden. Wird in einem Schritt eine Vertauschung oder Ersetzung erkannt, werden die entsprechenden Elemente mit den vorhergesehenen CSS-Klassen annotiert, die eine visuelle Hervorhebung bewirken: Bei einer Vertauschung erscheint das jeweilige Element blau, bei einer Ersetzung orange.

Boxen mit gestricheltem Rahmen rotieren leicht nach rechts und links, um anzudeuten, dass sie sich im nächsten Schritt verändern werden. Diese Darstellung wird durch einen Vergleich zweier benachbarter Schritte erzeugt: Enthält der Folgeschritt eine Veränderung, erhalten die entsprechenden Elemente im aktuellen Schritt zusätzlich eine animierende CSS-Klasse, die ein „Wackeln“ nach links und rechts auslöst. Diese visuelle Markierung ermöglicht es den Anwender:innen, bereits im Vorfeld zu erkennen, welche Änderungen im nächsten Ausführungsschritt zu erwarten sind.

Abbildung 4.6 zeigt die Darstellung eines Ersetzungsfalls anhand eines Algorithmus, der alle Elemente schrittweise auf den Wert 0 setzt.

Die orangefarbene Markierung der Boxen signalisiert als Warnfarbe, dass ein Element im aktuellen Schritt ersetzt und nicht getauscht wurde, was bei korrekter Implementierung des Algorithmus nicht vorkommen dürfte.

Im Codeeditor auf der rechten Seite wird in den Zeilen 11 bis 13 ein Dreiecktausch durchgeführt. Da das Analysetool jedoch ausschließlich Zustandsänderungen des Arrays erkennt, werden lediglich die Zeilen 12 und 13 hervorgehoben – denn nur in diesen erfolgt tatsächlich ein Schreibzugriff auf das Array.

Ursprünglich wurde die Ausführung der Zeilen 12 und 13 als separater Analyseschritt behandelt. Durch eine Nachverarbeitung im Backend des Analysetools werden diese beiden Schritte jedoch zu einem einzigen zusammengeführt: Der erste Schritt wird aus dem Analyseergebnis entfernt und seine Informationen in den zweiten Schritt eingebettet. Nur so kann ein einzelner Analyseschritt einen vollständigen Tausch abbilden.

4.1 Webanwendung

Die Hervorhebung der ersten Zeilennummer (hier Zeile 12) erfolgt anhand dieser eingebetteten Zusatzinformation im zweiten Schritt. Die genaue Funktionsweise dieser Nachverarbeitung wird in Kapitel 4.3.7.1 beschrieben.

Da der dargestellte Analyseschritt der Ausführung von Zeile 13 entspricht, zeigt die Variablen-Tabelle jene lokalen Variablen an, die zu diesem Zeitpunkt im Gültigkeitsbereich waren.

The screenshot shows a web-based programming environment for learning algorithms and data structures. The title is "Bubble Sort". The interface includes a navigation bar with "Home" and "Courses", and a breadcrumb trail: Home > Algorithmen und Datenstrukturen (ADS) > Sorting > Bubble Sort. Below the title are three tabs: "Instructions", "Visualization", and "Tests". The "Tests" tab is currently active, showing a table with variables and their values:

Variable	arr	n	i
Value	-	7	2

Below the table is a visualization area showing seven boxes representing array elements: 0, 0, 0, 12, 22, 11, 90. The fourth box (containing 12) is highlighted with a dashed border, indicating it is the current element being compared or processed. To the right of the visualization is a code editor window displaying C-like pseudocode:

```
1 > #pragma region prototypes...
6
7 void sort(int* arr, int n) {
8     for (int i = 0; i < n - 1; ++i) {
9         arr[i] = 0;
10    }
11 }
12
13 > #pragma region custom_functions...
23
24
```

Line 9 is highlighted with a teal background, corresponding to the highlighted element in the visualization. At the bottom of the code editor is a "Try it out!" button.

Abbildung 4.6: Fehlerhafte Implementierung eines Bubblesorts

4.1 Webanwendung

The screenshot shows a web-based Quicksort visualizer. At the top, there are navigation links: Home, Courses, and a breadcrumb trail: Home > Algorithmen und Datenstrukturen (ADS) > Sorting with Recursion > Quick Sort. Below this is a title "Quick Sort" with tabs for Instructions, Visualization (selected), and Tests.

The visualization area displays an array of 8 elements: 5, 2, 7, 1, 8, 3, 6, 4. To the left, there are "Variable" and "Value" sections. A "Try it out!" button is located at the bottom right of the visualization panel.

On the right side of the interface, the C code for the Quicksort algorithm is shown:

```
1 > #pragma region prototypes ...
10
11
12 void quickSort(int* arr, int n) {
13     quickSortRecursive(arr, 0, n - 1); // already implemented for
14 }
15
16 void quickSortRecursive(int* arr, int low, int high) {
17     if (low < high) {
18         int pi = partition(arr, low, high);
19         quickSortRecursive(arr, low, pi - 1);
20         quickSortRecursive(arr, pi + 1, high);
21     }
22 }
23
24 int partition(int* arr, int low, int high) {
25     int* pivot = &arr[high];
26     int i = low - 1;
27
28     for (int j = low; j < high; j++) {
29         if (arr[j] <= *pivot) {
30             i++;
31             swap(arr[i], arr[j]);
32         }
33     }
34     swap(arr[i + 1], arr[high]);
35
36     return i + 1;
37 }
```

Abbildung 4.7: Initialer Zustand der Quicksort Visualisierung

Abbildung 4.7 zeigt den initialen Zustand der Quicksort-Visualisierung. Der Quicksort-Algorithmus dient hier als konkret umgesetztes Beispiel für einen rekursiven Sortieralgorithmus. Ziel der Visualisierung ist es, die zentralen Charakteristika rekursiver Verfahren verständlich darzustellen – insbesondere das gleichzeitige Bestehen mehrerer aktiver Funktionsaufrufe auf unterschiedlichen Rekursionsebenen.

Auf den ersten Blick wirkt die Darstellung ähnlich wie bei der Visualisierung des Bubblesort-Algorithmus. Solange im Rahmen der Analyseschritte lediglich zwei Elemente getauscht oder ein Element durch einen neuen Wert ersetzt wird, übernimmt die Visualisierung exakt das gleiche Verhalten wie beim Bubblesort-Algorithmus. Eine wesentliche Neuerung stellt jedoch das Funktionsaufruf-Label „sort(arr, 0, 7)“ dar: Es steht für den initialen Aufruf der Funktion quickSortRecursive, die das Array arr im Bereich von Index 0 bis 7 sortieren soll. Da Quicksort-Algorithmus rekursiv arbeitet, erscheinen nachfolgende Funktionsaufrufe als Einträge in weiteren Zeilen unterhalb – eine Darstellung, die in einem späteren Abschnitt noch genauer erläutert wird.

4.1 Webanwendung

The screenshot shows a web-based application for visualizing the Quicksort algorithm. At the top, there are navigation links: Home, Courses, and a breadcrumb trail: Home > Algorithmen und Datenstrukturen (ADS) > Sorting with Recursion > Quick Sort. Below this is a title "Quick Sort" and a navigation bar with tabs: Instructions, Visualization (selected), and Tests.

The main area displays a table with two rows:

Variable	arr	low	high	pi
Value	-	0	7	0

Below the table is a visualization of an array of integers: [5, 2, 7, 1, 8, 3, 6, 4]. The element at index 7 (value 4) is highlighted in orange, indicating it is the current pivot element. To the left of the array, the function call `sort(arr, 0, 7)` is shown.

On the right side of the interface, there is a code editor window showing the C code for the Quicksort algorithm. The code is annotated with line numbers and some parts are highlighted in blue and orange. A specific line of code is highlighted in orange: `int pi = partition(arr, low, high);`. This corresponds to the highlighted pivot element in the visualization.

At the bottom of the visualization area, there are navigation buttons: a left arrow, "Step 3/88", and a right arrow. To the right of the code editor, there is a "Try it out!" button.

Abbildung 4.8: Markierung des Pivot-Elements in der Quicksort-Visualisierung

Die Aufgabenstellung zur Bearbeitung des Quicksort-Algorithmus legt fest, dass das Pivot-Element stets das letzte Element des aktuell zu sortierenden Bereichs ist. In Abbildung 4.8 ist dieses ausgewählte Pivot-Element farblich hervorgehoben – es erscheint in Orange. Zur Bestimmung des Pivot-Elements nutzt die Visualisierung das `high`-Argument der Funktion `quickSortRecursive`, da dieses genau das letzte Element des jeweiligen Bereichs bezeichnet. Die Markierung erfolgt bereits in einer Vorverarbeitung der Analyseschritte auf Seiten der Webanwendung, indem ein Flag gesetzt wird, das angibt, ob ein Element das Pivot-Element ist.

4.1 Webanwendung

The screenshot shows a web-based application for visualizing Quicksort. At the top, there are navigation links: Home, Courses, and a search bar. Below that, a breadcrumb trail: Home > Algorithmen und Datenstrukturen (ADS) > Sorting with Recursion > Quick Sort. The main area is titled "Quick Sort" and contains three tabs: Instructions, Visualization (selected), and Tests.

Instructions:

Variable	arr	low	high	i	j
Value	-	0	7	-1	0

Visualization: Shows an array of 8 boxes containing the values 5, 2, 7, 1, 8, 3, 6, 4. The value at index 0 is 5, and the value at index 7 is 4. A pink arrow points from the value 5 to the first box. A cyan arrow points from the value 4 to the last box. A comparison bar below the array shows the expression `arr[j] <= arr[high]`, where `arr[j]` is highlighted in purple and `arr[high]` is highlighted in green.

Tests:

```

1 > #pragma region prototypes ...
10
11
12 void quickSort(int* arr, int n) {
13     quickSortRecursive(arr, 0, n - 1); // already implemented for
14 }
15
16 void quickSortRecursive(int* arr, int low, int high) {
17     if (low < high) {
18         int pi = partition(arr, low, high);
19         quickSortRecursive(arr, low, pi - 1);
20         quickSortRecursive(arr, pi + 1, high);
21     }
22 }
23
24 int partition(int* arr, int low, int high) {
25     int i = low - 1;
26
27     for (int j = low; j < high; j++) {
28         if (arr[j] <= arr[high]) {
29             i++;
30             swap(arr[i], arr[j]);
31         }
32     }
33     swap(arr[i + 1], arr[high]);
34     return i + 1;
35 }

```

A "Try out!" button is located at the bottom right of the code editor.

Abbildung 4.9: Vergleichsoperation in der Quicksort-Visualisierung

In Abbildung 4.9 ist im Code eine Vergleichsoperation zu sehen, bei der mindestens einer der Operanden eine Referenz auf das zu sortierende Array darstellt. Dies führt zu einem zusätzlichen Animationsschritt in der Visualisierung. Die beteiligten Operanden werden im Code farblich hervorgehoben, wobei jeder Operand eine eigene Farbe erhält.

Für jede Vergleichsoperation wird in der Visualisierung ein Balken mit drei Kästchen unterhalb der dargestellten Datenstruktur angezeigt: Das linke Kästchen zeigt den linken Operanden, das mittlere den Operator und das rechte den rechten Operanden. Die beiden Operanden-Kästchen sind jeweils in der gleichen Farbe eingefärbt wie die entsprechenden Operanden im Codeeditor, um eine visuelle Zuordnung zwischen Code und Visualisierung zu ermöglichen. Handelt es sich bei einem Operanden um eine Referenz auf das zu sortierende Array, so wird dieser über einen Pfeil mit dem entsprechenden Arrayelement in der Visualisierung verbunden, wobei der Pfeil in derselben Farbe eingefärbt wird, wie auch der Operand selbst eingefärbt wurde. Besteht eine solche Referenz nicht, wird stattdessen der ermittelte Wert des Ausdrucks des Operanden im Kästchen angezeigt. Enthält ein Analyseschritt mehrere Vergleichsoperationen, so werden die zugehörigen Visualisierungsbalken untereinander dargestellt. Die Operan-

4.1 Webanwendung

den werden jeweils in unterschiedlichen Farben dargestellt, um die einzelnen Operanden visuell voneinander zu trennen. Insgesamt stehen sechs Farben zur Verfügung, die zyklisch wiederverwendet werden, sobald mehr als drei Vergleichsoperationen in einer Codezeile auftreten.

The screenshot shows a web-based tool for visualizing the Quicksort algorithm. At the top, there are tabs for Home, Courses, and a breadcrumb navigation: Home > Algorithmen und Datenstrukturen (ADS) > Sorting with Recursion > Quick Sort. Below this is a section titled "Quick Sort" with three tabs: Instructions, Visualization (which is selected), and Tests. The "Instructions" tab shows variable assignments: arr = [-], low = 0, high = 7, i = 2, and j = 2. The "Tests" tab has a "Try it out!" button. On the right, the C code for the quicksort algorithm is displayed:

```
12 void quickSort(int* arr, int low, int high) {
13 }
14 }
15
16 void quickSortRecursive(int* arr, int low, int high) {
17     if (low < high) {
18         int pi = partition(arr, low, high);
19         quickSortRecursive(arr, low, pi - 1);
20         quickSortRecursive(arr, pi + 1, high);
21     }
22 }
23
24 int partition(int* arr, int low, int high) {
25     int i = low - 1;
26
27     for (int j = low; j < high; j++) {
28         if (arr[j] <= arr[high]) {
29             i++;
30             swap(arr[i], arr[j]);
31         }
32     }
33     swap(arr[i + 1], arr[high]);
34     return i + 1;
35 }
36
37
38 > #pragma region custom_functions...
```

Line 34 is highlighted in light blue. The "Visualization" tab shows an array of eight boxes: 2, 1, 3, 4 (highlighted in green), 8, 7, 6, 5. The "Tests" tab has a "Try it out!" button.

Abbildung 4.10: Rückgabewert der partition-Funktion

Abbildung 4.10 zeigt einen weiteren Animationsschritt, in dem der Rückgabewert der Funktion `partition` als Index interpretiert und das entsprechende Arrayelement in der Visualisierung grün hervorgehoben wird. Grundlage dieser Darstellung ist die Eigenschaft des Quicksort-Algorithmus, dass das von `partition` zurückgegebene Element im korrekt implementierten Algorithmus dauerhaft an seiner finalen Position im sortierten Array verbleibt. Auf dieser Logik basiert eine schrittweise grüne Einfärbung der Arrayelemente: Im Verlauf der Visualisierung werden – bei korrekter Implementierung – nach und nach alle Elemente grün markiert, bis das gesamte Array vollständig sortiert ist. Der zugrunde liegende Analyseschritt wurde erzeugt, weil nicht nur Funktionsaufrufe, sondern auch Rücksprünge aus Funktionen als eigenständige Analyseschritte betrachtet werden. In diesem Fall enthält der Schritt Informationen darüber, aus welcher Funktion zurückgesprungen wurde, in welche Funktion die Ausführung zurückkehrt und welcher Rückgabewert dabei übergeben wurde.

The screenshot shows a web application interface for a Quicksort visualizer. At the top, there are navigation links: Home, Courses, and a search bar. Below that, a breadcrumb trail shows the path: Home > Algorithmen und Datenstrukturen (ADS) > Sorting with Recursion > Quick Sort.

The main area is titled "Quick Sort". It has three tabs: "Instructions", "Visualization", and "Tests". The "Instructions" tab is active, showing a table with variables and their values:

Variable	arr	low	high	pi
Value	-	0	7	3

Below the table, there are two rows of boxes representing the array elements. The first row contains boxes labeled 4, 8, 7, 6, and 5. The second row contains boxes labeled 2, 1, and 3. The box containing 3 is highlighted in yellow, indicating it is the pivot element.

On the right side of the screen, there is a code editor window displaying C code for the Quicksort algorithm. The code is annotated with line numbers from 12 to 48. A specific line, line 19, is highlighted in blue, showing the recursive call to quickSortRecursive with parameters arr, low, and pi - 1. The code also includes a partition function and a pragma region for custom functions.

Abbildung 4.11: Rekursionsschritt in der Quicksort-Visualisierung

Abbildung 4.11 zeigt einen weiteren Animationsschritt, in dem die Funktion quickSortRecursive erstmals rekursiv aufgerufen wird. Die Visualisierung reagiert darauf, indem sie eine neue Zeile hinzufügt. Diese enthält das Funktionsaufruf-Label „sort(arr, 0, 2)“, das die beim Aufruf übergebenen Parameter darstellt. Zur Verbesserung der Lesbarkeit werden alle darüberliegenden Zeilen verkleinert. Überlagerte Elemente in diesen Zeilen werden ausgegraut, und darin enthaltene Zahlen ausgeblendet. Die Visualisierung der neu hinzugefügten Zeile verhält sich identisch zur Darstellung des initialen Aufrufs, wie zuvor beschrieben.

Jeder weitere rekursive Aufruf der Funktion würde wiederum erneut eine neue Zeile erzeugen, in der die Parameter des jeweiligen Aufrufs angezeigt werden. Sobald ein Funktionsaufruf beendet wird, wird die zugehörige Zeile entfernt. Die darüberliegende Zeile wird wieder vergrößert, und sowohl der Ausgraeuffekt als auch das Ausblenden der Zahlen werden rückgängig gemacht.

Grundlage dieses für die Erkennung der Rekursion ist die Tatsache, dass für jeden Funktionsaufruf und -rücksprung ein eigener Analyseschritt erzeugt wird. Dieser enthält Informationen darüber, welche Funktion den Aufruf ausgelöst hat, welche Funktion aufgerufen wurde sowie die übergebenen Parameter mitsamt Namen und Werten

4.1 Webanwendung

bzw. aus welcher Funktion in welche Funktion zurückgesprungen wird. Auf diese Weise kann eindeutig erkannt werden, ob es sich um einen Aufruf der Funktion quickSortRecursive handelt. Ist dies der Fall, wird eine neue Zeile mit dem entsprechenden Funktionsaufruf-Label erzeugt.

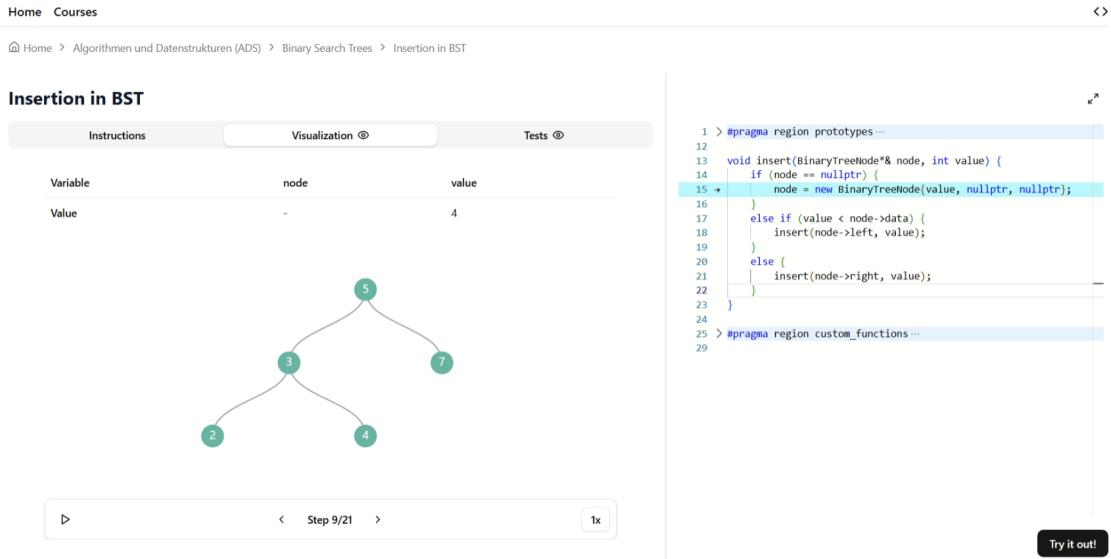


Abbildung 4.12: Visualisierung eines Einfüge-Algorithmus in einen Binärbaum

In Abbildung 4.12 ist die Visualisierung zur Aufgabenstellung eines Einfüge-Algorithmus für einen binären Suchbaum dargestellt. Die Struktur des Baums wird dabei dynamisch über die Pointer-Verbindungen durch das Analysetool ermittelt. Visualisierungsschritte ergeben sich aus rekursiven Funktionsaufrufen sowie strukturellen Veränderungen innerhalb des Baums, die beispielsweise bei einer Einfügung auftreten.

Die Baumstruktur wird mithilfe der D3-Bibliothek [31], [32] erstellt. Dabei wird die zugrunde liegende Pointerstruktur des Baums in ein hierarchisches Datenformat überführt, das von der Bibliothek anschließend zur Darstellung als Baumdiagramm verwendet wird.

Angesichts der erhöhten Komplexität bei der Darstellung dynamischer Baumstrukturen wurde im Rahmen der prototypischen Umsetzung bewusst auf Übergangsanimationen zwischen den einzelnen Schritten verzichtet.

Für zukünftige Versionen wäre es denkbar, dass das Analysetool sämtliche Knoten, die während der Ausführung mindestens einmal erkannt wurden, speichert. Falls ein gemerkter Knoten aus Versehen durch fehlerhafte Pointerzuweisungen aus dem Baum herausfällt, könnte er in einem separaten Teilbaum neben der Hauptstruktur visualisiert werden. Dadurch ließen sich derartige Fehlerfälle leichter nachvollziehen und besser debuggen.

4.1.4 Testansicht

Im Folgenden wird erläutert, wie das Testergebnis des Backends in der Testansicht visualisiert wird. Das Testergebnis entsteht durch die Ausführung vordefinierter Testfälle, bei denen das eingegebene Programm mit bestimmten Eingabewerten ausgeführt und die Ausgabe anschließend mit den erwarteten Ergebnissen verglichen wird.

Sobald die Testergebnisse vorliegen, wird prominent angezeigt, ob alle Tests erfolgreich durchlaufen wurden – „All tests passed!“ (siehe *Abbildung 4.13*) – oder ob es zu Fehlern kam – „Some tests have failed“ (siehe *Abbildung 4.14*). Das bedeutet konkret, dass das eingegebene Programm entweder für sämtliche Eingaben das erwartete Ergebnis liefern konnte oder bei mindestens einem Testfall davon abgewichen ist.

In der Darstellung der Testergebnisse wird zwischen öffentlichen und privaten Testfällen unterschieden. Auf der linken Seite befinden sich die öffentlichen Testfälle, bei denen sowohl die Eingabewerte als auch die erwarteten Ausgaben den Anwender:innen zur Verfügung gestellt werden. Durch einen Klick auf „Show all test cases“ lässt sich ein Modalfenster öffnen, in dem alle öffentlichen Testfälle gesammelt einsehbar sind. Zu jedem fehlgeschlagenen öffentlichen Test gibt es einen Button „Visualize“, der aktuell noch keine Funktion bietet, jedoch andeutet, dass perspektivisch eine Visualisierung dieses konkreten Falles ausgeführt werden könnte, um die Fehlersuche gezielt zu unterstützen.

Auf der rechten Seite sind hingegen die privaten Testfälle dargestellt, bei denen weder Eingabe- noch Ausgabewerte sichtbar sind. Diese Einschränkung dient dazu, zu

4.1 Webanwendung

verhindern, dass Anwender:innen ihre Lösungen gezielt auf die bekannten Testfälle zuschneiden – etwa durch das Hartkodieren von Rückgabewerten –, statt ein funktional korrektes und verallgemeinerbares Programm zu entwickeln.

Da sich die Entwicklung der Anwendung derzeit noch im Prototyp-Stadium befindet, wurden bislang lediglich Testfälle für Sortieralgorithmen erstellt. Für die Binärbaum-Aufgabenstellung ist die Testfunktionalität gegenwärtig noch nicht implementiert.

The screenshot shows a web-based programming environment for a 'Quick Sort' algorithm. At the top, there are tabs for 'Home', 'Courses', 'Instructions', 'Visualization', and 'Tests'. The 'Tests' tab is active, indicated by a highlighted border. Below the tabs, a green checkmark icon and the text 'All tests passed!' are displayed. On the left, there are two sections: 'Public Tests' (3/3 Passed) and 'Private Tests' (3/3 Passed). Both sections show a green checkmark icon and the text 'All public tests are passing!' and '3/3 Tests Passing' respectively. A note below the private tests states: 'Private test details are hidden to prevent hardcoding solutions'. On the right, the source code for the quicksort algorithm is shown in C-like syntax:

```
1 > #pragma region prototypes...
10
11 void quickSort(int* arr, int n) {
12     quickSortRecursive(arr, 0, n - 1); // already implemented for you ;
13 }
14
15
16 void quickSortRecursive(int* arr, int low, int high) {
17     if (low < high) {
18         int pi = partition(arr, low, high);
19         quickSortRecursive(arr, low, pi - 1);
20         quickSortRecursive(arr, pi + 1, high);
21     }
22 }
23
24 int partition(int* arr, int low, int high) {
25     int i = low - 1;
26
27     for (int j = low; j < high; j++) {
28         if (arr[j] <= arr[high]) {
29             i++;
30             swap(arr[i], arr[j]);
31         }
32     }
33     swap(arr[i + 1], arr[high]);
34     return i + 1;
35 }
```

A 'Try it out!' button is located at the bottom right of the code editor.

Abbildung 4.13: Erfolgreiche Testansicht

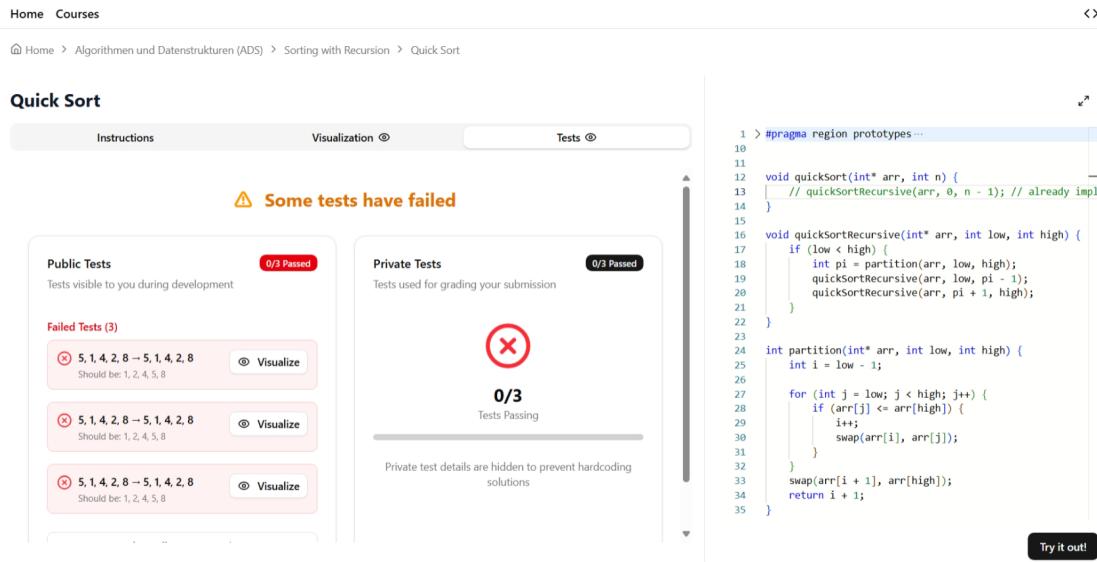


Abbildung 4.14: Fehlgeschlagene Testansicht

4.1.5 Codeeditor

Der Codeeditor wird mit dem *Monaco Editor* realisiert. Die zu markierenden Zeilennummern werden als Zeilennummernarray von einem globalen Store verwaltet, der mithilfe der *zustand*-Bibliothek [33] implementiert ist. Dieser globale Store wird von der Webanwendung pro Schritt aktualisiert. Die Editorkomponente greift auf dieses globale Zeilennummernarray zu, reagiert auf dessen Änderungen und aktualisiert entsprechend die Markierungen, indem alte Zeilenhervorhebungen entfernt und neue hinzugefügt werden. Der *Monaco Editor* stellt hierfür eine Schnittstelle bereit, mit der sich spezifische Textbereiche über CSS-Klassen stylen lassen (siehe Abschnitt 5.1.1). Die Webanwendung nutzt diese Möglichkeiten, um sowohl die Zeilennummer selbst als auch die entsprechende Codezeile mit einem leicht bläulichen Hintergrund zu versehen. Das gleiche Prinzip wird auch verwendet, um die einzelnen Operanden einer Vergleichsoperation im Codeeditor hervorzuheben (siehe Abbildung 4.9).

Da der *Monaco Editor* ausschließlich das Zuweisen von CSS-Klassen erlaubt und keine dynamischen Styles unterstützt, wurde die Anzahl der verfügbaren Farben auf sechs

4.1 Webanwendung

begrenzt. Werden mehr als sechs Farben benötigt, weil es mehr als drei Vergleichsoperationen in einer Zeile gibt, so werden die Farben zyklisch wiederverwendet.

Das Codegerüst – in der Preset-Datenbank als Template bezeichnet – wird über einen HTTP-Endpoint der Preset-Datenbank an den Webserver als Template-Payload ausgeliefert. Der Webserver übergibt das Template per Server-Side Rendering und überträgt es darüber gemeinsam mit der Webanwendung an den Client.

Die Template-Payload enthält zum einen das vollständige Codegerüst, das für Anwender:innen schreibgeschützt sein soll. Ergänzend enthält das Template eine Liste von Codebereichen, die einzelne Abschnitte des schreibgeschützten Codes explizit als editierbar markieren. Innerhalb dieser Abschnitte sollen die Anwender:innen ihre eigene Implementierung ergänzen. Falls diese Aufgabenstellung noch nicht bearbeitet wurde, liefert die Preset-Datenbank jeweils einen initialen Füllcode mit – typischerweise in Form eines Kommentars wie `// todo: implement this function`. Eine detaillierte Beschreibung der Template-Struktur und -Konfiguration folgt in Unterkapitel 4.6.

Die Webanwendung nutzt zur Umsetzung des Mechanismus der eingeschränkt editierbaren Bereiche die Bibliothek *constrained-editor-plugin* [34]. Diese erweitert den *Monaco Editor* um eine Schnittstelle, mit der sich der Editor insgesamt in einen schreibgeschützten Zustand versetzen lässt, während nachträglich konfigurierte Codebereiche gezielt freigegeben werden. Die Webanwendung übernimmt dabei die Bereiche, die von der Preset-Datenbank übermittelt wurden, und markiert sie zusätzlich mit eindeutigen Labels. Mit den Labels wird es möglich, diese Bereiche gezielt mit Inhalt zu befüllen – entweder mit dem von der Preset-Datenbank gelieferten Initialcode oder mit personenspezifischem Code aus dem Local Storage des Browsers.

Zu diesem Zweck speichert die Webanwendung bei jeder Änderung im Editor den Inhalt der editierbaren Bereiche als Liste im Local Storage. Der Speicherkey folgt dabei dem Schema `functionBodies:<course-id>/<category-id>/<task-id>`, wobei `<course-id>`, `<category-id>` und `<task-id>` durch die jeweiligen IDs des Kurses, der Kategorie und der Aufgabe ersetzt werden. Die Inhalte werden dort als JSON-Liste abgelegt.

In Abbildung 4.16 sind die Zeilen 3–4, 8 sowie 12–19 als beschreibbare Codebereiche markiert. Der darin enthaltene Code wurde initial von der Webanwendung eingefügt, kann jedoch von den Anwender:innen beliebig angepasst werden. Die beschreibbaren Bereiche passen sich dynamisch an den eingegebenen Code an – sie wachsen oder schrumpfen entsprechend der jeweiligen Eingabe.

In den Aufgabenstellungen folgen die bereitgestellten Codegerüste einem festen Schema: Zu den bearbeitbaren Codebereichen gehört zunächst der obere Abschnitt, in dem eigene Funktionsprototypen oder Definitionen ergänzt werden können. Am unteren Ende des Codegerüsts befindet sich ein weiterer beschreibbarer Bereich, der zur Implementierung selbst definierter Funktionen vorgesehen ist. Zusätzlich sind die Funktionsrümpfe der vorgegebenen Funktionen – typischerweise mittig im Code platziert – editierbar, sodass die eigentliche Bearbeitung der Aufgabe innerhalb dieser vorgesehenen Strukturen erfolgen kann.

Für die Darstellung von Syntaxfehlern stellt der *Monaco Editor* eine Schnittstelle bereit, mit der sich fehlerhafte Codebereiche visuell durch eine Unterschlängelung hervorheben und mit einer entsprechenden Fehlermeldung versehen lassen (vgl. 5.1.1).

Das Analysetool im Backend liefert der Webanwendung hierfür die relevanten Informationen: Es gibt die betroffenen Codebereiche unter Angabe von Start- und Endposition (jeweils bestehend aus Zeilen- und Spaltennummer) sowie eine passende Fehlermeldung zurück. Die zugrunde liegende Fehlernachricht wird vom Analysetool als Fehler vom Typ `compilation-error` anstelle eines regulären Analyseergebnisses zurückgegeben.

Aktuell ist die Funktionalität allerdings noch eingeschränkt: Aufgrund technischer Limitationen im Backend wird derzeit nur das erste Zeichen des fehlerhaften Codebereichs übermittelt. Die Webanwendung ist jedoch bereits in der Lage, größere Bereiche korrekt zu markieren. Eine Erweiterung des Analysetools würde daher ausreichen, um die volle Funktionalität zu realisieren.

In den vordefinierten `#pragma`-Regionen befinden sich Funktionsprototypen, Implementierungen von Funktionen sowie C++-Strukturen. Zur besseren Übersicht sind beim

4.1 Webanwendung

A screenshot of a code editor interface. The code is as follows:

```
1 > #pra use of undeclared identifier 'foo'
12
13 void No quick fixes available value) {
14     foo = "bar";
15 }
16
17 > #pragma region custom_functions...
21
```

The word 'foo' is underlined with a red wavy line, indicating a spelling or undeclared identifier error. A tooltip above the cursor says 'use of undeclared identifier 'foo''. A 'No quick fixes available' message is shown in a dropdown menu. The code editor has a light blue background with dark grey horizontal lines.

Abbildung 4.15: Syntaxfehler, während Computermaus über dem „f“ von „foo“ schwebt

A screenshot of a code editor interface showing the initial state of a Bubblesort algorithm. The code is as follows:

```
1 ∵ #pragma region prototypes
2 // insert custom functions here (or prototypes)
3 void swap(int* a, int* b);
4
5 #pragma endregion prototypes
6
7 ∵ void sort(int* arr, int n) {
8     // todo: add your code here
9 }
10
11 ∵ #pragma region custom_functions
12 // insert custom functions here
13
14 ∵ void swap(int* a, int* b) {
15     int temp = *a;
16     *a = *b;
17     *b = temp;
18 }
19
20 #pragma endregion custom_functions
21
22
```

The code editor shows several collapsed regions indicated by a triangle icon before the line number. The 'swap' function and its definition are expanded, showing the variable declarations and assignment logic. The code editor has a light blue background with dark grey horizontal lines.

Abbildung 4.16: Initialer Codeeditor des Bubblesort-Algorithmus mit ausgeklappten #pragma regions

Laden der Seite alle `#pragma region`-Blöcke zunächst eingeklappt – einschließlich jener, die von den Anwender:innen selbst erstellt wurden. Durch einen Klick auf den Pfeil links neben der jeweiligen Zeilennummer können die Bereiche jederzeit manuell aufgeklappt werden. *Abbildung 4.3* zeigt den Zustand mit eingeklappten Regionen, während in *Abbildung 4.16* die entsprechenden Bereiche ausgeklappt dargestellt sind.

4.2 Webserver

Für die Webanwendung wird das Framework Next.js eingesetzt. Durch Server-Side Rendering wird der HTML-Code der Seite bereits auf dem Server mithilfe von JavaScript vorgerendert, um die Ladezeiten zu verkürzen und die wahrgenommene Performance beim Endnutzer zu verbessern. Im Anschluss wird die Seite im Browser durch den mitgelieferten JavaScript-Code „hydriert“, also interaktiv gemacht.

Der Webserver selbst wird über einen Reverse Proxy an das externe Netzwerk angebunden und so den Anwender:innen bereitgestellt. Um das entsprechende Codegerüst für die jeweilige Aufgabenstellung auszuliefern, bindet der Webserver den HTTP-Endpoint der Preset-Datenbank ein. Das resultierende Template wird serverseitig zusammen mit der restlichen Webanwendung generiert und an die Anwender:innen ausgeliefert.

4.3 Analysetool

Das Analysetool ist ein Kommandozeilenprogramm, das den von Anwender:innen eingebrachten C++-Code analysiert und das Analyseergebnis in Form von JSON-Nachrichten auf der Standardkonsole ausliefert. Ebenso ist das Analysetool fähig, den C++-Code mit vorgegebenen Testfällen zu testen. Das Analysetool ist vollständig von der Webanwendung entkoppelt. Auf diese Weise können auch andere Werkzeuge oder Visualisierungssysteme auf die Funktionalität des Analysetools zurückgreifen. Die Architektur folgt dem Prinzip einer klaren Trennung von Zuständigkeiten zwischen den einzelnen Teilsystemen.

Im Folgenden wird erläutert, wie das Analysetool ausgeführt und konfiguriert wird, welche Ausgaben es erzeugt und wie es technisch umgesetzt ist. Dabei wird insbesondere auf die Funktionsweise der internen Komponenten, den Aufbau der Konfiguration sowie das Zusammenspiel mit dem restlichen System eingegangen.

4.3.1 Überblick

Abbildung 4.17 zeigt die zentralen Bestandteile, die für den Analyseprozess erforderlich sind. Als Eingabe erhält das Pythonprogramm — der zentrale Steuermechanismus des Analyseablaufs — das ausgewählte Preset, das den Analyse- oder Testingprozess konfiguriert, sowie den eingereichten Code. Das Pythonprogramm führt den eingereichten Code aus und überwacht dessen Ablauf mithilfe des Debuggers. Nach Abschluss der Analyse wird das Ergebnis in strukturierter Form über die Standardausgabe an die Konsole ausgegeben.

Das Analysetool wird als Docker-Image bereitgestellt und enthält ein Kommandozeilenprogramm, mit dem die Kompilierung, Ausführung, Analyse und Testing des von Anwender:innen eingereichten C++-Codes durchgeführt wird. Für die Kompilierung kommt der C++-Compiler *Clang* zum Einsatz, der ebenso wie der Debugger *LLDB* Bestandteil des Docker-Images ist.

Die Logik des Analysetools ist in Python implementiert, weshalb das Docker-Image zusätzlich eine Python-Laufzeitumgebung sowie die entsprechenden Python API-Bindings für *LLDB* und *Clang* enthält. Die Python-Schnittstelle zu *LLDB* erlaubt eine automatisierte Steuerung des Debuggers, um Informationen über den aktuellen Programmzustand zu erhalten – beispielsweise die aktuell ausgeführte Codezeile, Werte lokaler Variablen oder den Zustand einer Datenstruktur. Weitere Details zur Analyse mittels *LLDB* finden sich im Unterkapitel 4.3.3.

Die Python-API zu *clang* ermöglicht es, den eingereichten Code auf Syntaxebene zu analysieren. Dies dient einerseits der Überprüfung, ob das vorgegebene Codegerüst korrekt eingehalten wurde, und andererseits der Identifikation von Vergleichsoperationen, die später im Analyseprozess logisch und visuell berücksichtigt werden.

Im Folgenden ist mehrfach von sogenannten Collectoren die Rede. Dabei handelt es sich um modulare Bausteine, die gezielt für einen Analyseprozess konfiguriert werden. Jeder Collector ist dafür zuständig, einen bestimmten Aspekt des zu analysierenden Codes zu überwachen – etwa den Zustand von Variablen oder Datenstrukturen, das Auftreten von Funktionsaufrufen oder das Ausführen von Vergleichsoperationen. Eine ausführlichere Erläuterung findet sich in Abschnitt 4.3.6.

Analyse

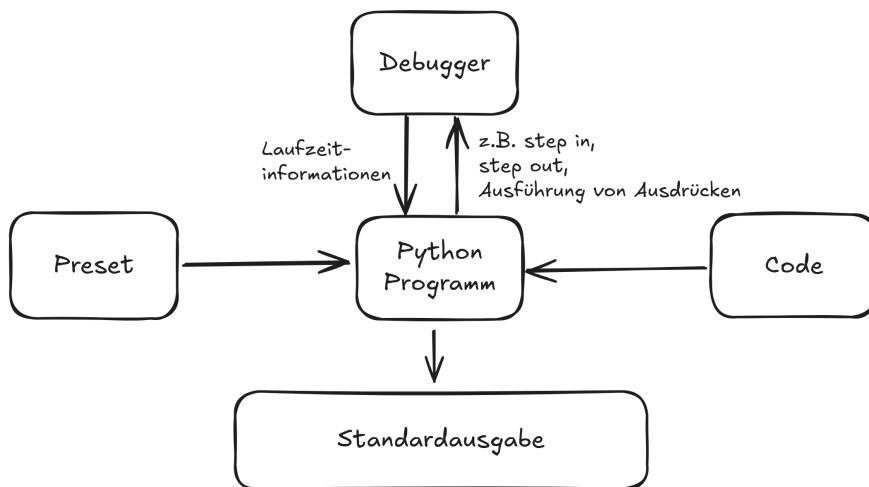


Abbildung 4.17: Komponenten des Analysetools

4.3.2 Verwendung des Tools

Im Folgenden wird jeweils für den Analyse- und den Testing-Modus kurz erläutert, mit welchen Parametern das Kommandozeilenprogramm gestartet wird und wie sich das Programm anschließend verhält.

4.3.2.1 Ausführung der Analyse

Für die Analyse wird das Analysetool im Modus `analyze` aufgerufen. Dabei werden zwei Parameter übergeben: Zum einen ein im JSON-Format kodiertes Preset, das

4.3 Analysetool

das zugrunde liegende Codegerüst konfiguriert und definiert, welche Zustände und Abläufe während der Analyse erhoben werden sollen. Zum anderen der vollständig eingereichte C++-Code als JSON-kodierter String. Dieser Code wurde zuvor durch die Webanwendung an den Executor übermittelt, der ihn beim Start des Analysetools an dieses weiterleitet. Der genaue Aufbau eines Presets wird im Unterkapitel 4.3.4 erläutert.

Während der Analyse gibt das Tool statusmarkierte Nachrichten im JSON-Format auf der Konsole aus, um den aktuellen Fortschritt darzustellen. Derzeit werden folgende Statusmeldungen ausgegeben: `Validating...`, `Compiling...`, `Launching Analysis...` und `Analyzing....`

Nach Abschluss der Analyse gibt das Tool eine finale, als Ergebnis gekennzeichnete JSON-Nachricht zurück. Diese enthält das vollständige Analyseergebnis. Das Analyseergebnis wird im JSON-Format bereitgestellt und besteht aus einer Liste einzelner Analyseschritte. Jeder Analyseschritt ist als Key-Value-Struktur aufgebaut, wobei jeder Schlüssel einem Collector entspricht und der zugehörige Wert die vom Collector ermittelten Daten in Form eines JSON-Objekts enthält. Es ist dabei wichtig zu betonen, dass während der Analyse keine Zwischenstände oder Teilergebnisse an die Webanwendung gesendet werden. Auch wenn die grafische Oberfläche eine schrittweise Ausführung des C++-Codes suggeriert, basiert diese vollständig auf dem zuvor berechneten Gesamtergebnis der vollständigen Programmanalyse. Die Collectoren werden im Abschnitt 4.3.6 erläutert. Ein beispielhaftes Analyseergebnis wird in Abschnitt 5.2.3 gezeigt.

4.3.2.2 Ausführung von Testfällen

Im Test-Modus wird das Analysetool mit denselben Parametern aufgerufen wie im Analysemodus, jedoch wird der erste Parameter auf `test` gesetzt. Zusätzlich zur Analysekonfiguration umfasst das übergebene Preset eine definierte Testumgebung sowie eine Reihe von Testfällen, die im Test-Modus automatisch vom Analysetool ausgeführt werden.

Auch im Test-Modus gibt das Tool während der Ausführung statusmarkierte Nachrichten im JSON-Format aus, um den Fortschritt zu dokumentieren. Aktuell werden

folgende Statusmeldungen erzeugt: Validating..., Compiling... und Running tests....

Nach Abschluss wird – analog zum Analysemodus – eine finale, als Ergebnis gekennzeichnete JSON-Nachricht zurückgegeben, die das Testergebnis enthält.

4.3.2.3 Fehlerbehandlung

Tritt während des Analyse- oder Testprozesses ein Fehler auf, wird anstelle eines Ergebnisses eine entsprechend gekennzeichnete Fehlermeldung zurückgegeben. Dabei wird differenziert, ob es sich um einen vom Compiler gemeldeten Syntaxfehler handelt – inklusive Angabe der fehlerhaften Codestellen und zugehöriger Fehlermeldung (vgl. Abschnitt 4.1.5) – oder ob der Fehler innerhalb des Analysetools selbst entstanden ist.

4.3.3 Ablauf der Analyse

In *Abbildung 4.18* ist der vollständige Ablauf des Analyseprozesses in Form eines Ablaufdiagramms dargestellt. Es handelt sich dabei um eine Detailansicht der Komponente „Pythonprogramm“, wie sie in *Abbildung 4.17* dargestellt ist.

4.3 Analysetool

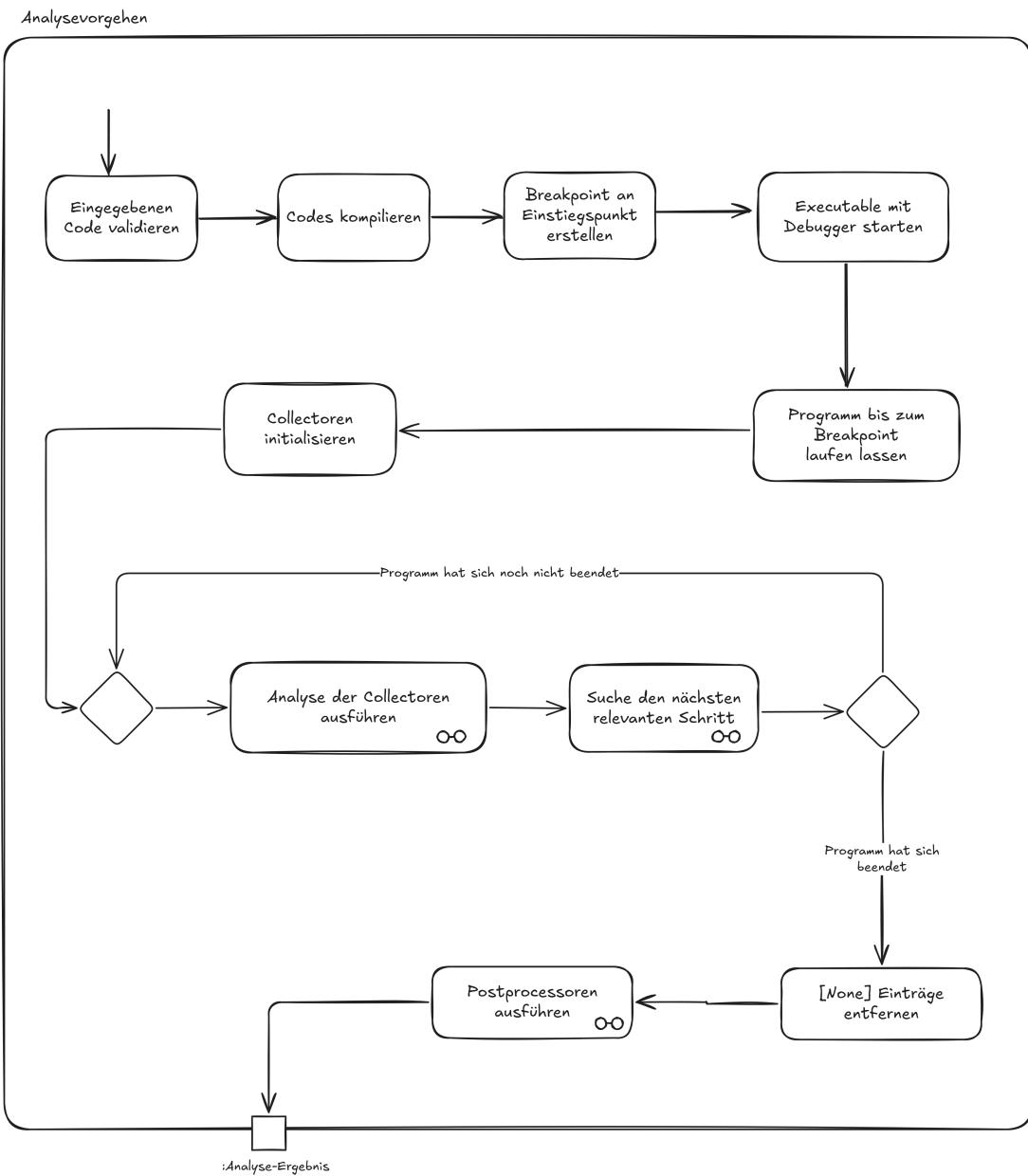


Abbildung 4.18: Implementierung der Analysephase

Zu Beginn überprüft das Analysetool im Schritt „Eingegebenen Code validieren“ den von Anwender:innen eingereichten Code auf seine strukturelle Integrität, um sicherzustellen, dass er mit dem vorgegebenen Template übereinstimmt und nicht manipuliert wurde. Dies dient insbesondere dem Schutz vor einer missbräuchlichen Verwendung

des Analyseprozesses. Abschnitt 4.3.5 beschreibt die dabei eingesetzten Strategien im Detail.

Im darauffolgenden Schritt „Codes kompilieren“ wird der im Preset enthaltene Boilerplate-Code, der unter anderem eine `int main()`-Funktion bereitstellt, gemeinsam mit dem eingereichten C++-Code kompiliert. Dazu werden aus beiden Quelltexten temporäre Dateien erzeugt, die mithilfe der Clang-CLI kompiliert und gelinkt werden. Das Ergebnis ist eine debugfähige ausführbare Datei. Die dafür genutzten Befehle werden im Abschnitt 5.2.7 gezeigt.

Nach erfolgreicher Kompilierung wird in den Schritten „Breakpoint an Einstiegspunkt erstellen“ und „Executable mit Debugger starten und bis zum Breakpoint laufen lassen“ mithilfe der LLDB-API ein Debugger initialisiert, der die erzeugte ausführbare Datei lädt und an der ersten Zeile der Einstiegsfunktion im eingereichten Code einen Breakpoint setzt. Das Program wird dann im Debug-Modus gestartet und bis zur Erreichung des Breakpoints laufen gelassen. Anschließend erfolgt mit dem Schritt „Collectoren initialisieren“ die Initialisierung aller konfigurierten Collectoren.

Mit Abschluss der Initialisierung beginnt die schrittweise Ausführung des Programms. Zunächst werden die zuvor vorbereiteten Collectoren in dem Schritt „Analyse der Collectoren ausführen“ angewiesen, den aktuellen Zustand des Programms zu analysieren. Anschließend werden im Schritt „Suche den nächsten relevanten Schritt“ mithilfe des Debuggers ein oder mehrere Einzelschritte durchgeführt. Befindet sich der Debugger dabei außerhalb des von Anwender:innen geschriebenen Codes, wird er so lange fortgeschreitend durch den Code geführt, bis erneut eine Stelle innerhalb des Codes der Anwender:innen erreicht ist. *Abbildung 4.19* veranschaulicht dieses Verfahren im Detail: Im Schritt „Debugger: Einzelschritt“ wird zunächst ein StepInto [35] ausgeführt. Sollte es sich um einen Funktionsaufruf handeln, springt der Debugger bei einem Step-Into in den aufgerufenen Funktionsrumpf. Danach wird geprüft, ob sich der Debugger weiterhin im Code der Anwender:innen befindet. Ist dies der Fall, wird der Schritt nach außen gegeben, sodass die Collectoren diesen analysieren können. Befindet sich der Debugger hingegen in einer anderen Datei, wird ein StepOut [36] ausgeführt. Dieser Schritt tritt typischerweise dann ein, wenn eine externe Funktion betreten wurde, deren Implementierung außerhalb des eingereichten Codes liegt, oder wenn die Ausführung in

4.3 Analysetool

die `int main()`-Funktion des Boilerplates zurückkehrt, etwa weil die Funktion des eingereichten Codes abgeschlossen ist. In letzterem Fall führt ein `StepOut` üblicherweise zum Ende der Programmausführung.

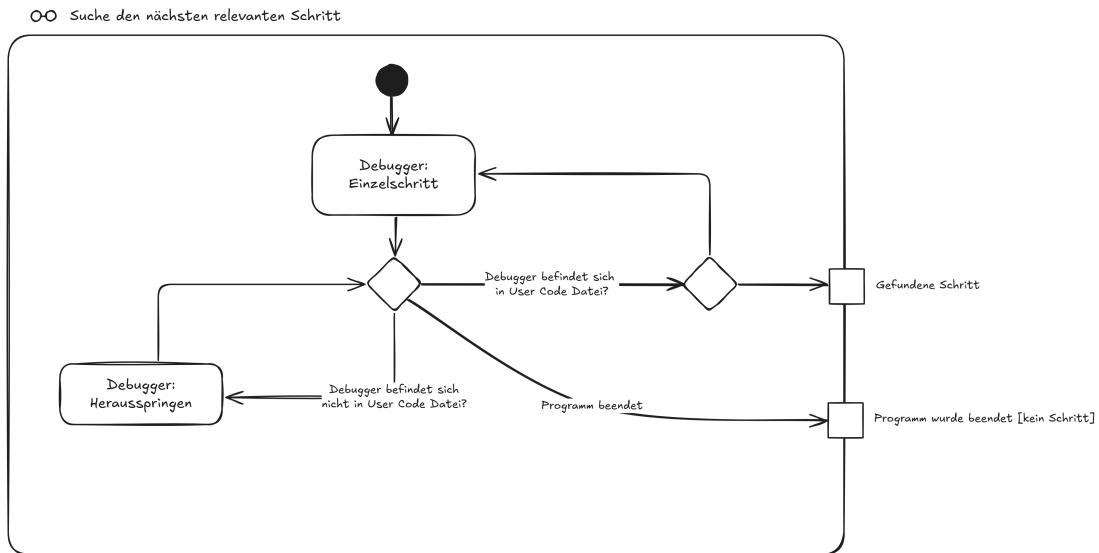


Abbildung 4.19: Auswahl der Debuggingschritte zur Analyse

Sofern sich das Programm nicht durch den Schritt „Suche den nächsten relevanten Schritt“ beendet hat, wird der Ablauf erneut durchlaufen: Zunächst analysieren die Collectoren den aktuellen Programmschritt. Anschließend wird der nächste relevante Ausführungsschritt ermittelt, und der gesamte Vorgang beginnt von vorn.

Die Analyse der Collectoren erfolgt dabei wie folgt: Erkennt mindestens ein Collector ein relevantes Ereignis, wird ein neuer Analyseschritt erzeugt. Die Ergebnisse aller beteiligten Collectoren werden in diesem Schritt zusammengeführt und als Eintrag in eine Liste von Analyseschritten aufgenommen. Die genaue Funktionsweise der Collectoren sowie eine Übersicht über die verfügbaren Collector-Klassen bietet Abschnitt 4.3.6.

Abbildung 4.20 veranschaulicht den Ablauf in einer vereinfachten Form. Diese Darstellung zeigt den Prozess bei drei Collectoren: Für jeden Debuggingschritt der Programmausführung werden sämtliche Collectoren angewiesen, diesen Schritt zu analysieren. Die dabei gewonnenen Erkenntnisse werden in einem gemeinsamen Analyseschritt zusammengeführt und dem Analyseergebnis hinzugefügt. Es sei jedoch angemerkt, dass

nicht bei jedem Programmschritt ein Analyseschritt erzeugt wird, was der Abbildung nicht zu entnehmen ist. Diese Annahme trifft dann zu, wenn von keinem der Collectoren eine signifikante Zustandsänderungen festgestellt wird.

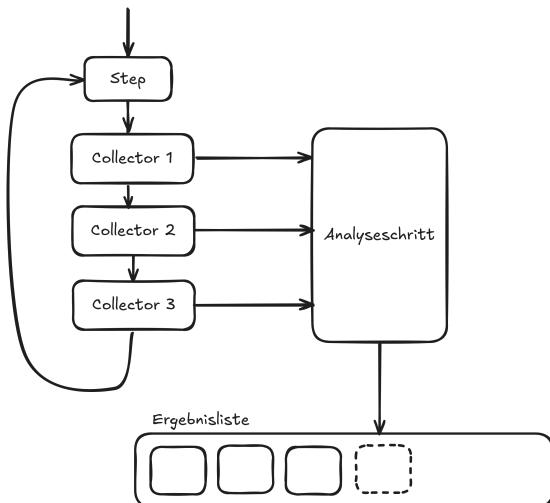


Abbildung 4.20: Erzeugung eines Analyseschritts

Nach Beendigung der Programmausführung wird die vollständige Liste der gesammelten Analyseschritte durch die im Preset konfigurierten Postprocessoren im Schritt „Postprocessoren ausführen“ weiterverarbeitet. Dieses Vorgehen ist in *Abbildung 4.21* dargestellt: Die durch die Collectoren erzeugte „Vorläufige Ergebnisliste“ wird sequentiell durch die Postprocessoren verarbeitet, wobei jeder Postprocessor Operationen auf der Liste durchführt und sein Ergebnis an den jeweils nächsten weitergibt. Das Resultat des letzten Postprocessors bildet schließlich das strukturierte Analyseergebnis, das an die Standardausgabe übergeben wird. Ein Postprocessor der Klasse `KeepTrackOf-Items` wird zum Beispiel dazu verwendet, um aus den Analyseschritten abzuleiten, ob und welche Array-Elemente eines Arrays getauscht wurden.

Der Datenfluss erfolgt, sofern nicht anders spezifiziert, ausschließlich im Arbeitsspeicher und nicht über Dateioperationen.

Um zu verhindern, dass eine ineffiziente Implementierung des eingereichten Codes oder eine Endlosschleife das Analysetool übermäßig lange blockiert, wird der Analyseprozess

4.3 Analysetool

nach zehn Sekunden automatisch abgebrochen. In diesem Fall wird ein entsprechender Fehler generiert und an den Client übermittelt.

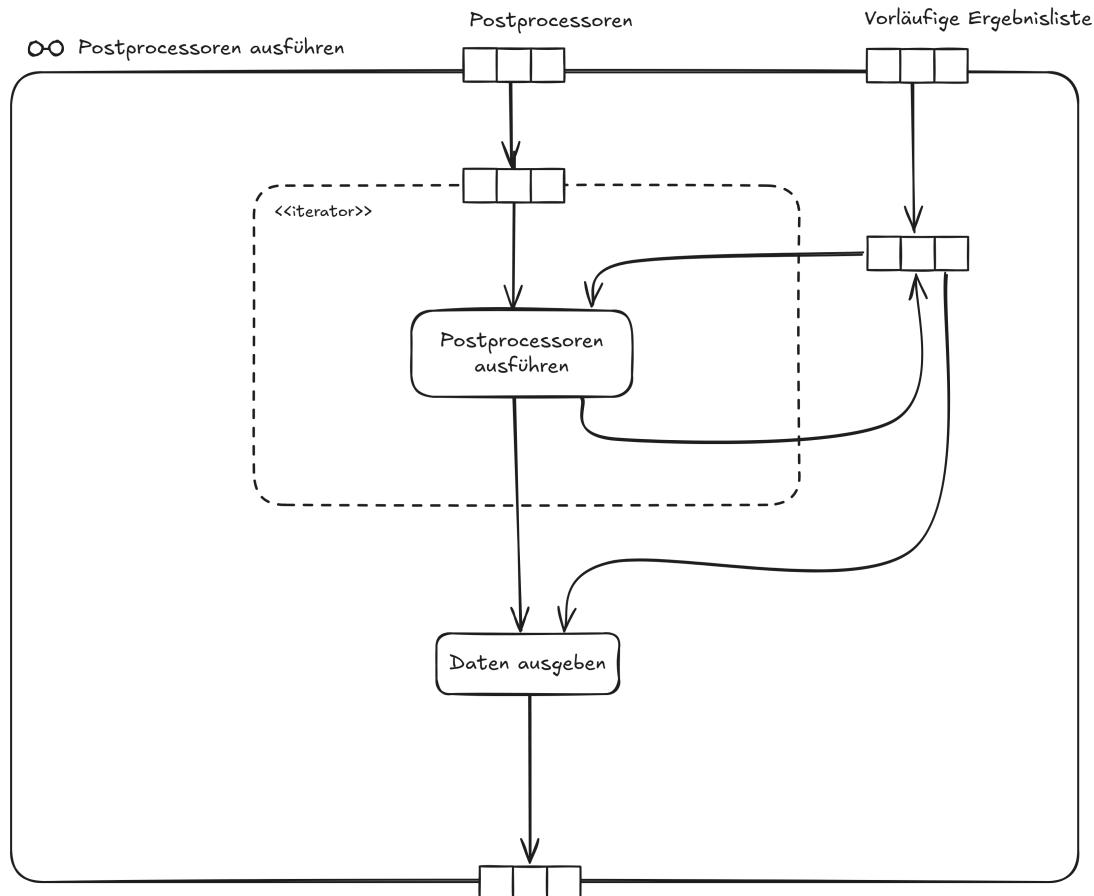


Abbildung 4.21: Nachverarbeitung mit den Postprocessoren

4.3.4 Preset

Ein Preset besteht aus einem JSON-Objekt, das mehrere Bestandteile umfasst: Ein Code-Template, Boilerplate-Code für Analyse und Testumgebung, allgemeine Konfigurationsdaten, spezifische Einstellungen für Collectoren und Postprocessoren im Analyseprozess sowie eine Sammlung von Testfällen für den Test-Modus.

Template Das Template definiert ein Grundgerüst für den von Anwender:innen zu schreibenden Code. Es enthält Markierungen, die bestimmte Bereiche als beschreibbar kennzeichnen und optional initialen Beispielcode in diesen Bereichen bereitstellen. Ein solcher Bereich beginnt mit dem Kommentar `// <user-code-start>` und endet mit `// <user-code-end>`, wobei beide Kommentare jeweils in einer eigenen Zeile und ohne weiteren Code stehen müssen. Alle übrigen Teile des Templates sind schreibgeschützt und dürfen von Anwender:innen nicht verändert werden. Ein beispielhaftes Template findet sich in Abschnitt 5.2.1

Die Webanwendung nutzt das Template, um den Codeeditor mit der korrekten Struktur zu initialisieren und die erlaubten Eingabebereiche festzulegen. Das Analysetool verwendet das Template, um den eingereichten Code zu validieren und sicherzustellen, dass die strukturellen Vorgaben eingehalten werden. Auf diese Weise wird verhindert, dass Anwender:innen das Systemverhalten durch nicht vorgesehene Codeänderungen beeinflusst. Weitere Details zur Validierung finden sich in Abschnitt 4.3.5.

Analyseteil Der Boilerplate-Code für die Analyse enthält einen Einstiegspunkt in Form einer `main`-Funktion, der die von Anwender:innen implementierten Funktionen mit vordefinierten, hartkodierten Eingabewerten ausführt. Dieser Code dient ausschließlich der Vorbereitung des Analyseprozesses.

Die Analysekonfiguration legt fest, welche Funktion, die von den Anwender:innen zu implementieren ist, als Einstiegspunkt für die Analyse Verwendung findet. Darüber hinaus werden sogenannte Collectoren definiert, die vor der Ausführung jedes einzelnen Debuggingschritts ausgeführt werden, um Informationen über den aktuellen Programmzustand zu erfassen. Ein Collector entscheidet darüber, ob ein neuer Analyseschritt erzeugt wird, indem er nach Ausführung ein Ergebnis zurückliefert. Das zurückgegebene Ergebnis ist maßgeblich für die inhaltliche Ausgestaltung des Analyseergebnisse verantwortlich. Wird von dem Collector kein Ergebnis zurückgegeben, so möchte er keinen neuen Analyseschritt erzeugen. Weitere Informationen dazu befinden sich in Unterkapitel 4.3.6.

Nach Abschluss der Analyse kommen optional Postprocessoren zum Einsatz, die die erzeugten Analyseschritte nachbearbeiten und deren strukturierte Weiterverarbeitung

durch die Webanwendung erleichtern. Mehr dazu in Unterkapitel 4.3.7.

Testteil Der Boilerplate-Code für die Testumgebung stellt ebenfalls eine `main`-Funktion bereit, die die von Anwender:innen implementierten Funktionen aufruft – jedoch unter Verwendung der Eingabewerte aus den definierten Testfällen. Testfälle enthalten vollständige Informationen über ihre Eingabewerte, die erwarteten Ausgaben sowie die tatsächlich vom eingereichten Programm erzeugten Ausgaben. Diese Eingaben werden vom Analysetool über die Standardkonsole bereitgestellt.

In der Testfallkonfiguration wird zwischen zwei Kategorien unterschieden: `public` und `private`. Öffentliche Testfälle sind von den Anwender:innen vollständig einsehbar. Die Konfiguration dieser Testfälle fließen direkt in das sichtbare Testergebnis ein und werden in der Webanwendung angezeigt, ergänzt um eine Zusammenfassung, wie viele Testfälle bestanden wurden und wie viele insgesamt definiert sind.

Private Testfälle hingegen sind für Anwender:innen nicht einsehbar. In das Testergebnis fließt hier ausschließlich die Anzahl der bestandenen Testfälle im Verhältnis zur Gesamtanzahl ein. Sie dienen primär der abschließenden Bewertung und sollen insbesondere verhindern, dass Anwender:innen ihre Lösung auf die bekannten öffentlichen Tests zuschneiden.

4.3.5 Code Validierung

Um sicherzustellen, dass der von der Webanwendung übermittelte C++-Code die vorgegebene Struktur des Templates einhält und nicht manipuliert wurde, etwa um potenziell schadhaften Code auszuführen, wird der Code serverseitig auf syntaktische Integrität geprüft. Hintergrund dieser Maßnahme ist, dass clientseitiger Schreibschutz im Editor umgangen werden könnte, indem beispielsweise ein modifizierter HTTP-Request abgesetzt wird, der den eigentlichen Schutzmechanismus der Webanwendung unterläuft.

Zu diesem Zweck wird der eingereichte C++-Code zunächst mithilfe der Python-API von *Clang* geparsst. Anschließend wird geprüft, ob sich im Code ein `#include`-Statement befindet. Ist dies der Fall, wird der Vorgang mit einer Fehlermeldung abgebrochen.

#include-Direktiven sind nicht erlaubt, da die Aufgabenstellungen ohne externe Bibliotheken lösbar sein sollen. Darüber hinaus soll auf diese Weise die Angriffsfläche für potenziell sicherheitskritische Nutzung der Plattform reduziert werden.

Im nächsten Schritt wird das ursprüngliche Template herangezogen, wobei alle als beschreibbar markierten Bereiche – also jene zwischen // <user-code-start> und // <user-code-end> – entfernt werden. Der verbleibende, nicht-editierbare Teil des Templates wird ebenfalls mit der Clang-API geparsst.

Für beide Codevarianten (eingereichter Code und bereinigtes Template) wird anschließend jeweils der Syntaxbaum durchlaufen. Dabei durchläuft der Validator alle Top-Level-Knoten des Syntaxbaums – etwa Funktions- oder Strukturdefinitionen – und extrahiert für jeden einzelnen Knoten dessen Quelltextrepräsentation, die anschließend als eigenständiger Eintrag in ein Set übernommen wird. Bei Funktionsdefinitionen wird dabei der eigentliche Funktionsrumpf entfernt, sodass lediglich die Signatur berücksichtigt wird. Kommentare und Whitespace werden bereits vom Parser verworfen.

Abschließend wird geprüft, ob das Set der Templateeinträge eine Teilmenge des Sets aus dem eingereichten Code darstellt. Ist dies nicht der Fall – etwa, weil eine im Template definierte Funktion im eingereichten Code fehlt oder verändert wurde, wird der Code als ungültig eingestuft.

4.3.6 Collectoren

Ein Collector ist eine Instanz einer vordefinierten Collector-Klasse innerhalb des Analysetools. Collectoren dienen dazu, bestimmte Aspekte des Programmlaufs zu beobachten und aufzuzeichnen – beispielsweise ausgeführte Codezeilen, Zustände von Datenstrukturen oder Funktionsaufrufe.

Im jeweiligen Preset ist definiert, welche Collector-Klassen instanziert und mit welchen Parametern diese konfiguriert werden sollen. Die zu instanzierenden Collector-Klassen werden mit einem eindeutigen Namen angesprochen. Damit die namentliche Zuordnung erfolgen kann, verfügt das Analysetool über eine interne Zuordnungstabelle: Eine

4.3 Analysetool

Mapping-Datenstruktur verknüpft jeden eindeutigen Typnamen mit der entsprechenden Collector-Klasse. Mehrere Collectoren derselben Klasse können parallel verwendet werden, sofern sie unterschiedlich parametrisiert sind – eine identische Konfiguration würde lediglich zu redundanten Ausführung führen. Zusätzlich wird jedem Collector ein eindeutiger Schlüssel zugewiesen, unter dem das Analyseergebnis des Collectors im jeweiligen Analyseschritt abgelegt wird.

4.3.6.1 Konfiguration eines Collectors

Die Konfiguration aller Collectoren erfolgt im Preset in Form einer JSON-Liste, wobei jeder Collector durch ein eigenes JSON-Objekt beschrieben wird. Dieses Objekt enthält drei zentrale Felder:

Das Feld `type` bezeichnet den eindeutigen Namen der Collector-Klasse und dient als Referenz auf die zuvor beschriebene interne Zuordnungstabelle. Das Feld `key` gibt den Bezeichner an, unter dem die Analyseergebnisse dieses Collectors später im Analyseschritt gespeichert werden. Das Feld `parameters` enthält eine Liste von Schlüssel-Wert-Paaren, mit denen der Collector individuell konfiguriert werden kann. In Abschnitt 5.2.2 wird ein Beispiel dieser Konfiguration gezeigt.

Um das Verhalten eines Sortieralgorithmus zu analysieren, kann etwa das Array `array` beobachtet werden, dessen Länge in der Variable `length` gespeichert ist. Hierzu wird im Preset ein Collector der Klasse `ArrayWatcher` konfiguriert, die auf genau dieses Array parametrisiert ist. Zusätzlich können weitere Collectoren der Klassen `CurrentLine` und `CurrentScope` aktiviert werden, die keine Parameter benötigen. Die zuletzt genannten Collectoren liefern beim Auftreten eines Analyseschritts ergänzend zu den Informationen des `ArrayWatchers` die aktuell ausgeführte Codezeile bzw. eine Auflistung der lokalen Variablen.

4.3.6.2 Arten von Collectoren

Innerhalb des Analysetools existieren zwei Arten von Collectoren, die sich hinsichtlich ihrer Funktion im Analyseprozess unterscheiden. First-Level-Collectoren sind dafür

zuständig, den jeweils beobachteten Zustand zu bewerten und eigenständig zu entscheiden, ob ein neuer Analyseschritt erzeugt werden soll. Wird ein solcher Schritt durch ein Collector ausgelöst, so stellt diese die zugehörigen Analyseinformationen in Form eines JSON-Objekts bereit. Diese Informationen werden unter dem zuvor konfigurierten Schlüssel abgelegt und sind im jeweiligen Analyseschritt des finalen Analyseergebnis sichtbar. Kommt es vor, dass mehrere First-Level-Collectoren innerhalb desselben Debuggingzustands gleichzeitig entscheiden, dass ein Analyseschritt erzeugt werden soll, so werden die jeweils gelieferten Informationen in einem gemeinsamen Analyseschritt zusammengeführt. Dabei entsteht ein gemeinsames JSON-Objekt, das die Schlüssel-Wert-Paare aller beteiligten Collectoren enthält, wobei jeweils der Schlüssel der vorkonfigurierte Schlüssel des Collectors und der Wert die jeweils extrahierte Information des entsprechenden Collectors ist.

Second-Level-Collectoren kommen nur dann zum Einsatz, wenn zuvor mindestens ein First-Level-Collector die Erzeugung eines Analyseschritts veranlasst hat. Sie dienen dazu, diesen Schritt um zusätzliche Kontextinformationen anzureichern – etwa die als Nächstes auszuführende Zeilennummer oder die zum jeweiligen Zeitpunkt im Gültigkeitsbereich befindlichen lokalen Variablen samt ihrer Werte. Die von ihnen ermittelten Daten werden in das Ergebnis des entsprechenden Analyseschritts integriert und gemeinsam mit den Ausgaben der First-Level-Collectoren zusammengeführt.

Bereits während der Initialisierung der Analyse wird jeder Collector klassifiziert, ob sie als First-Level- oder Second-Level-Collector agieren soll. Zur besseren Unterscheidbarkeit folgt die Benennung einer klaren Konvention: First-Level-Collectoren tragen das Suffix „Watcher“, während Second-Level-Collectoren mit dem Präfix „Current“ benannt werden.

Generell gilt, dass die meisten First-Level-Collectoren Veränderungen im Programmverhalten technisch erst nach Ausführung eines Debuggingschritts erkennen können – beispielsweise, weil sich Zustandsänderungen wie Modifikationen an Variablen oder Datenstrukturen erst dann manifestieren. Aus diesem Grund hat sich in der Umsetzung die Konvention etabliert, dass auch alle anderen Collectoren, deren ermittelten Informationen dem noch auszuführenden Debuggingschritt zugeordnet werden – wie etwa CurrentLine – ihre Informationen um einen Schritt verzögern, damit diese dem

kommenden Debuggingschritt zugeordnet werden können. Die entsprechenden Informationen werden dafür intern gepuffert und jeweils mit einem Schritt Verzögerung dem passenden Analyseschritt zugeordnet. Diese Vorgehensweise stellt sicher, dass alle Collectoren zeitlich synchronisierte Analyseschritte erzeugen und ihre jeweiligen Informationen konsistent demselben Programmschritt zuordnen. Nach Abschluss der Programmausführung ruft das Analysetool die Collectoren ein letztes Mal auf, um etwaig verbleibende gepufferte Informationen abzurufen und in die Analyseergebnisse zu übernehmen.

Diese Art der Zuordnung ermöglicht es, dass die durch CurrentLine angegebene Codezeile – anders als im klassischen Debugging – nicht die unmittelbar bevorstehende, sondern die bereits ausgeführte Zeile bezeichnet. Dadurch lässt sich einfacher nachvollziehen, welche Codezeile zu einer konkreten Zustandsänderung im Programm geführt hat.

Zu Beginn der Analyse setzt das Analysetool einen Breakpoint auf die erste Zeile der von Anwender:innen definierten Einstiegsfunktion. Anschließend wird jeder Collector initialisiert, indem deren Initialisierungsfunktion mit dem Kontext des momentanen Debuggerzustands ausgeführt wird.

4.3.6.3 Implementierung von Collectoren

Ein Collector implementiert ein Interface mit den in *Abbildung 4.22* dargestellten Methoden, die unterhalb der gestrichelten Linie in runden Rechtecken visualisiert sind: „is_reason_for_new_step“, „setup“, „pre_step“ und „step“.

Die Methode „is_reason_for_new_step“ wird einmal initial für alle Collectoren aufgerufen. Gibt sie True zurück, handelt es sich bei dem Collector um einen First-Level-Collector. Bei einem Rückgabewert von False wird sie als Second-Level-Collector klassifiziert.

Die Methode „setup“ wird im Schritt „Collectoren initialisieren“ der *Abbildung 4.18* ausgeführt. Ihr wird das aktuelle LLDB Frame übergeben, das den Kontext des Debuggers zur Initialisierung der Analyse bereitstellt.

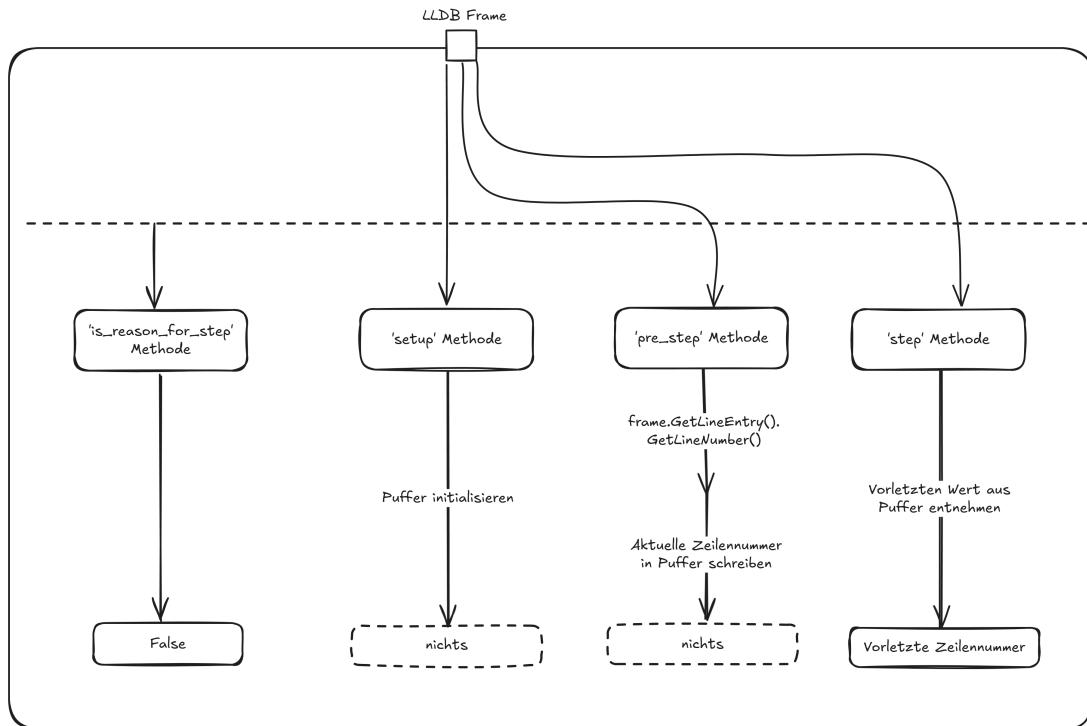


Abbildung 4.22: Implementierung des CurrentLine-Collectors

Nach jedem durchlaufenen Debuggingschritt im Rahmen des Verfahrens „Finde relevante Schritte“ wird zunächst die Methode „`pre_step`“ sowohl bei First-Level- als auch bei Second-Level-Collectoren ausgeführt. Dies ermöglicht insbesondere den Second-Level-Collectoren, die ermittelten Informationen, die dem kommenden Debuggingschritt zugeordnet werden, zwischenzuspeichern, sodass sie ihren Beitrag um einen Schritt verzögert liefern können.

In der darauffolgenden Methode „step“ werden zunächst alle First-Level-Collectoren aufgerufen. Falls einer von ihnen einen Analyseschritt erzeugt, folgen anschließend die Aufrufe der Second-Level-Collectoren, zu denen auch der CurrentLine-Collector gehört. Der Rückgabewert dieser Methode wird – sofern er nicht None ist – unter dem im Preset konfigurierten Schlüssel in den aktuellen Analyseschritt übernommen. Ist der Rückgabewert hingegen None, so möchte der Collector für diesen Debuggingschritt keinen neuen Analyseschritt erzeugen.

4.3.6.4 Verfügbare Collectoren

Die nachfolgenden Abschnitte beschreiben die einzelnen im Analysetool implementierten Collector-Typen sowie deren spezifisches Verhalten während der Analyse.

ArrayWatcher Ein Collector der `ArrayWatcher`-Klasse ist ein First-Level-Collector, der ein bestimmtes Array überwacht und einen neuen Analyseschritt erzeugt, sobald sich die Elemente des Arrays verändern. Eingesetzt wird dieser Collector typischerweise bei der Analyse von Sortieralgorithmen, um die einzelnen Sortierschritte nachvollziehbar und reproduzierbar zu machen.

In dem vorgegebenen Sortier-Grundgerüst werden das zu sortierende Array sowie die Anzahl seiner Elemente als Parameter an die Einstiegspunkt-Funktion übergeben. Die Namen dieser beiden Parameter werden in der Konfiguration des Collectors angegeben.

Während der Initialisierung liest der Collector sowohl die Referenz auf das Array als auch die Größe des Arrays aus und speichert diese Informationen intern ab. Diese Maßnahme dient dazu, den Collector robust gegenüber späteren Modifikationen der Pointer-Adresse des Arrays oder der Längenvariable zu machen.

Beim ersten Schritt der Analyse wird der Anfangszustand des Arrays erfasst und direkt in einem initialen Analyseschritt dokumentiert. Zwischen der Ausführung der darauf folgenden Debuggingschritten wird der Inhalt des Arrays kontinuierlich überwacht. Zu jedem Schritt wird der aktuelle Zustand neu berechnet und mit dem vorherigen verglichen. Sobald eine Änderung festgestellt wird, erzeugt der Collector einen neuen Analyseschritt, der den veränderten Zustand des Arrays enthält.

Dabei ist zu beachten, dass eine Veränderung technisch erst nach dem erfolgten Debuggingschritt, d. h. nach Abarbeitung der C++-Codezeile, welche die Veränderung durchführt, erkennbar wird. Aus diesem Grund muss das Ergebnis nicht aktiv gepuffert werden – es kann unmittelbar im Moment der Erkennung ausgegeben werden, da es bereits dem gerade ausgeführten Programmschritt zugeordnet wird.

Außerdem ist zu beachten, dass der `ArrayWatcher` ausschließlich tatsächliche Zustandsänderungen des Arrays erkennt. Wird ein Wert zwar überschrieben, entspricht der überschreibende Wert jedoch dem vorherigen Wert, so bleibt der Zustand unverändert. In einem solchen Fall wird kein neuer Analyseschritt erzeugt.

CurrentScope Ein Collector der `CurrentScope`-Klasse ist ein Second-Level-Collector, der alle lokalen Variablen des aktuellen Debugging-(Halte-)Zustands ausliest und dem Analyseschritt hinzufügt, sofern dieser zuvor durch einen First-Level-Collector ausgelöst wurde. Die erfassten Informationen werden von der Webanwendung in der tabellarischen Variablenansicht dargestellt.

Für jede erkannte lokale Variable werden der Name, der aktuelle Wert (bei Referenzen: der Wert der referenzierten Variable), der Typname als String, sowie ein Hinweis darauf, ob es sich um einen Pointer oder eine Referenz handelt, als strukturierte Datenpunkte dem Analyseschritt hinzugefügt.

Da die Informationen zum aktuellen Variablen-Zustand aus der Ausführung des vorangegangenen Debuggingschritts stammen wird hier kein Puffer benötigt.

Eine Konfiguration dieses Collectors im Preset erfordert keine Parameter.

CurrentLine Ein Collector der `CurrentLine`-Klasse ist ein Second-Level-Collector, der die aktuelle C++-Quellcode-Zeilenummer des jeweiligen noch auszuführenden Debuggingschritts erfasst und dem Analyseschritt hinzufügt, sofern dieser von einem First-Level-Collector ausgelöst wurde.

Diese Zeileninformation wird von der Webanwendung verwendet, um die entsprechende Codezeile im Editor visuell hervorzuheben.

Da die Informationen zur aktuellen Zeile aus des noch zu verarbeiteten Debuggingschritts stammen, stellt der Collector einen internen Puffer bereit und gibt jeweils den Zustand des vorangegangenen Debuggingschritts aus.

CompareOperationWatcher Ein Collector der CompareOperationWatcher-Klasse ist ein First-Level-Collector, der die Ausführung von Vergleichsoperationen im eingereichten Code überwacht. Erkennt der Collector für den zugeordneten Debuggingschritt eine beliebige Vergleichsoperation, wird ein neuer Analyseschritt generiert.

Im erzeugten Analyseschritt sind sowohl der Vergleichsoperator als auch die beiden Operanden enthalten – jeweils mit ihrem aktuellen Wert, ihrer textuellen Repräsentation und ihrer exakten Position im Quellcode. Die Positionsangaben umfassen Start- und Endkoordinaten im Format Zeile/Spalte.

Ein Collector der Klasse erfordert keine besondere Konfiguration.

Da sich mit der LLDB-Debugger-API zur Laufzeit nur schwer feststellen lässt, ob im vorangegangenen Debuggingschritt eine Vergleichsoperation ausgeführt wurde und an welcher Stelle im Quellcode diese stattfand, führt der Collector bereits in seiner Initialisierung eine syntaktische Analyse des gesamten eingereichten Codes durch. Dabei werden sämtliche Vergleichsoperationen identifiziert und deren Codezeilen gespeichert.

Bei Ausführung einer Analyse wird überprüft, ob die noch auszuführende Codezeile in der Liste der gefundenen Vergleichsoperationen auftaucht. Ist dies der Fall, so ist dieser Schritt ein Kandidat für den kommenden Analyseschritt. Noch vor Ausführung des nächsten Debuggingschritts werden die jeweiligen Operanden-Ausdrücke zur Laufzeit von dem Debugger ausgewertet, damit die ermittelten Operandenwerte anschließend zu dem resultierenden Analyseschritt hinzugefügt werden können. Da Ausdrücke jedoch Funktionsaufrufe oder andere ausführungserhebliche Operationen enthalten können, sind potenzielle Seiteneffekte bei der Auswertung nicht auszuschließen. Mögliche Auswirkungen solcher Seiteneffekte wurden im Rahmen der prototypischen Entwicklung jedoch nicht weiter untersucht oder berücksichtigt.

Da die Information zur aktuellen Zeile ermittelt wurde, bevor der kommende Debuggingschritt ausgeführt wurde, stellt der Collector einen internen Puffer bereit und gibt somit jeweils die Informationen aus, die im vorangegangenen Debuggingschritt errechnet wurden.

Die Collector-Klasse wird als Basisklasse für den `ArrayCompareOperationWatcher` benutzt, der die Funktionalität erweitert und nur Analyseschritte generiert, wenn sich die Vergleichsoperation auf ein zuvor konfiguriertes Array bezieht.

ArrayCompareOperationWatcher Ein Collector der `ArrayCompareOperationWatcher`-Klasse ist ein First-Level-Collector, der die Ausführung von Vergleichsoperationen im eingereichten Code überwacht. Erkennt der Collector für den zugeordneten Debuggingschritt eine Vergleichsoperation, bei der mindestens einer der Operanden ein Verweis auf ein bestimmtes Array ist, wird ein neuer Analyseschritt generiert.

Im erzeugten Analyseschritt sind sowohl der verwendete Vergleichsoperator als auch beide Operanden enthalten – jeweils mit ihrem aktuellen Wert, ihrer textuellen Darstellung sowie ihrer genauen Position im Quelltext. Darüber hinaus wird festgehalten, ob es sich bei einem oder beiden Operanden um Referenzen auf das beobachtete Array handelt und, falls zutreffend, auf welchen Index innerhalb des Arrays verwiesen wird. Die Positionsangaben umfassen Start- und Endkoordinaten im Format Zeile/Spalte.

Diese Informationen werden in der Visualisierung der Webanwendung dafür verwendet, um visuelle Blöcke zur Darstellung von Vergleichsausdrücken mit Pfeilverbindungen zu den betroffenen Array-Elementen anzuzeigen. Gleichzeitig werden die betroffenen Operanden sowohl im Codeeditor als auch in der Visualisierung farblich hervorgehoben, sodass eine visuelle Zuordnung zwischen Code und Datenstruktur möglich ist. Dieses Verhalten ist in *Abbildung 4.9* dargestellt.

Die Konfiguration des Collectors erfordert – analog zum `ArrayWatcher` – den Namen der zu beobachtenden Array-Variable sowie den Namen der Variablen, die die Anzahl der Elemente enthält.

Die Ermittlung aller potenziell relevanten Vergleichsoperationen erfolgt durch die Basisklasse `CompareOperationWatcher`. Für jede von ihr identifizierte Zeile mit einer Vergleichsoperation überprüft der Collector unmittelbar vor der Ausführung des entsprechenden Debuggingschritts, ob einer der Operanden auf das zu überwachende Array verweist. Nur wenn dies der Fall ist, wird ein entsprechender Analyseschritt erzeugt. Diese Prüfung erfolgt, indem der jeweilige textuelle Operanden-Ausdruck aus

4.3 Analysetool

dem Quellcode extrahiert und mithilfe des Referenzierungsoperators in einen neuen Ausdruck überführt wird: `&(<Operanden-Ausdruck>)`.

Dieser Ausdruck wird zur Laufzeit im Debugger ausgewertet. Da Ausdrücke jedoch Funktionsaufrufe oder andere ausführungserhebliche Operationen enthalten können, sind potenzielle Seiteneffekte bei der Auswertung nicht auszuschließen. Mögliche Auswirkungen solcher Seiteneffekte wurden im Rahmen der prototypischen Entwicklung jedoch nicht weiter untersucht oder berücksichtigt.

Führt die Auswertung zu keinem Fehler und ergibt sich daraus ein gültiger Pointer, wird geprüft, ob dessen Adresse in den Speicherbereich des beobachteten Arrays fällt.

Da die Information zur aktuellen Zeile ermittelt wurde, bevor der kommende Debuggingschritt ausgeführt wurde, stellt der Collector einen internen Puffer bereit und gibt jeweils die Informationen aus, die im vorangegangenen Debuggingschritt errechnet wurden.

RecursionWatcher Ein Collector der `RecursionWatcher`-Klasse ist ein First-Level-Collector, der Einsprünge in Funktionen sowie Rücksprünge aus Funktionen erkennt und für jedes dieser Ereignisse einen neuen Analyseschritt generiert. Im Fall eines Einsprungs enthält der Analyseschritt die Information, dass ein Funktionsaufruf stattgefunden hat, den Namen der Funktion, in der der Aufruf erfolgt ist, den Namen der aufgerufenen Funktion sowie die dort übergebenen Parameter – strukturiert im selben Format, wie es auch der `CurrentScope`-Collector bereitstellt (siehe Abschnitt 4.3.6.4). Bei einem Rücksprung enthält der Schritt die Information, dass eine Funktion verlassen wurde, den Namen der Funktion, aus der zurückgesprungen wurde, den Namen der Funktion, in die zurückgesprungen wird, sowie den Rückgabewert.

In der Quicksort-Visualisierung dient dieser Collector dazu, den Rekursionsstapel abzubilden. Die Webanwendung durchläuft dafür alle Analyseschritte, erkennt die Einsprung- und Rücksprungereignisse und rekonstruiert daraus den aktuellen Rekursionsstapel aller Aufrufe der Funktion `quickSortRecursive`, wie in *Abbildung 4.11* zu sehen ist.

Da keine konkreten Variablen oder Datenstrukturen überwacht werden, benötigt der `RecursionWatcher` keine Parameter in der Preset-Konfiguration.

Bei der Initialisierung speichert der Collector eine Referenz auf das Stackframe der Einstiegsfunktion des eingereichten Programms in einer internen Stack-Datenstruktur, die den aktuellen Rekursionsstapel modelliert. Zur Erkennung eines Ein- oder Rücksprungs wird bei nach der Ausführung eines jeden Debuggingschritts die aktuelle Stackframe-Referenz ausgelesen und mit dem obersten Element des internen Stacks verglichen. Ist die Referenz identisch mit dem obersten Frame, liegt kein Funktionswechsel vor, sondern ein Schritt innerhalb der aktuellen Funktion. Entspricht die Referenz dem zweitobersten Frame, handelt es sich um einen Rücksprung – die aktuelle Funktion wurde verlassen und die vorherige wieder aufgenommen. Weicht die Referenz sowohl vom obersten als auch vom zweitobersten Frame ab, wird auf einen neuen Einsprung geschlossen. In beiden Fällen – Einsprung oder Rücksprung – wird das interne Modell des Rekursionsstapels entsprechend aktualisiert.

Eine technische Herausforderung ergibt sich bei der Ermittlung des Rückgabewerts beim Rücksprung: Die LLDB-Debugger-API stellt zwar eine Funktion bereit, um den Rückgabewert der zuletzt verlassenen Funktion auszulesen, diese funktioniert jedoch nur in Verbindung mit einem expliziten StepOut-Kommando [37]. Bei der Analyse wird jedoch standardmäßig StepInto (vgl. 4.3.3) verwendet, wodurch der Rückgabewert nicht verfügbar ist (siehe Abschnitt 4.3.3). Als Lösung wird für jede Rücksprung-Instruktion im Maschinencode der analysierten Funktion zur Laufzeit ein Breakpoint gesetzt. Wird ein solcher Breakpoint erreicht, führt der Debugger automatisch ein StepOut aus, wodurch der Rückgabewert dann ermittelt werden kann. Da Rücksprung-Instruktionen wie `ret` oder `retq` ohnehin am Ende einer Funktion stehen, verhält sich der Debugger damit analog zu einem regulären StepInto.

Die Umsetzung erfolgt zur Laufzeit, indem der Debugger nach jedem Debuggingschritt prüft, ob für alle Rücksprung-Instruktionen der aktuellen Funktion bereits Breakpoints gesetzt wurden. Falls dies nicht der Fall ist, werden alle Maschinenbefehle der Funktion analysiert und für jede Rücksprung-Instruktion ein Breakpoint mit einem zugeordneten StepOut-Verhalten registriert.

Dabei ist zu beachten, dass jedes dieser beiden Ereignisse technisch erst nach Ausführung des kommenden Debuggingschritts erkennbar wird. Aus diesem Grund muss das Ergebnis nicht aktiv gepuffert werden – es kann unmittelbar im Moment der Erkennung

ausgegeben werden, da es bereits dem vorhergehenden Programmschritt zugeordnet wird.

BinaryTreeWatcher Ein Collector der `BinaryTreeWatcher`-Klasse ist ein First-Level-Collector, der Zustandsänderungen eines binären Baums erkennt, dessen Knoten über Pointer miteinander verbunden sind. Immer wenn sich die Struktur des Baums verändert, generiert der Collector einen neuen Analyseschritt. Er kommt insbesondere bei der Analyse von Algorithmen zum Einsatz, die auf Binärbäumen operieren, um deren schrittweise Veränderungen sichtbar zu machen.

Zur Bestimmung des aktuellen Baumzustands wird ausgehend vom Wurzelknoten eine rekursive Rekonstruktion der Baumstruktur in einer internen Python-Datenstruktur durchgeführt. Dabei wird berücksichtigt, dass durch fehlerhafte Implementierungen Zyklen entstehen können. Um endlose Rekursionen zu vermeiden, wird während der Traversierung eine Liste der bereits besuchten Knoten geführt, sodass Zyklen frühzeitig erkannt und unterbunden werden können.

Der Collector benötigt als Konfigurationsparameter den Namen der Pointer-Variable, die auf den Wurzelknoten des zu beobachtenden Baums verweist. In der Aufgabenstellung „*Insertion in BST*“ wird dieser über den Funktionsparameter `node` an die von Anwender:innen zu implementierende Einstiegsfunktion übergeben (siehe auch *Abbildung 4.12*).

4.3.7 Postprocessoren

Postprocessor-Klassen werden im Preset als Postprocessoren konfiguriert und instanziiert. Sie verarbeiten die von den Collectoren während der Analyse erzeugten Daten nach Abschluss der Programmausführung weiter und bereiten diese gezielt für die Darstellung in der Webanwendung auf. Die Verarbeitung erfolgt direkt im Analyseprozess, nachdem das eingereichte Programm vollständig ausgeführt und alle Analyseschritte durch die Collectoren erzeugt wurden. Der erste konfigurierte Postprocessor erhält die vollständige Liste aller Analyseschritte als interne Datenstruktur im Speicher, modifiziert

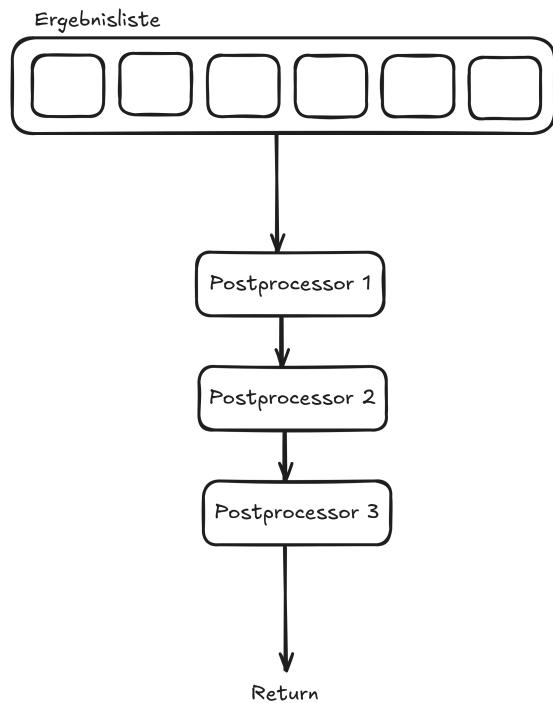


Abbildung 4.23: Datenfluss bei Nutzung von drei Postprocessoren

diese gemäß seiner eigenen Logik und übergibt das Ergebnis an den nächsten Postprocessor in der definierten Reihenfolge. Obwohl es technisch möglich ist, den Typ der Datenstruktur vollständig zu verändern, verbietet eine Konvention solche strukturellen Änderungen, um die Kompatibilität mit nachfolgenden Postprocessoren sicherzustellen. Die Übergabe erfolgt rein programmatisch, ohne zwischenzeitliche Persistierung in einer Datei – die Datenstruktur wird direkt im Speicher weitergereicht. Das final bearbeitete Ergebnis des letzten Postprocessors wird als Gesamtergebnis der Analyse an die Standardausgabe der Konsole ausgegeben. Von dort aus übernimmt der Executor die Ausgabe und leitet sie an die Webanwendung weiter.

In Abbildung 4.23 wird der vorgestellte Ablauf mithilfe von drei Postprocessoren vereinfacht dargestellt.

Die Konfiguration eines Postprocessors folgt einem ähnlichen Muster wie die eines Collectors. Über das Feld `type` wird die Postprocessor-Klasse anhand eines eindeutigen Bezeichners referenziert. Der Eintrag `key` legt fest, unter welchem Schlüssel

4.3 Analysetool

der Postprocessor seine gegebenenfalls erzeugten Zusatzinformationen innerhalb eines Analyseschritts ablegen darf. Über das Feld `parameters` kann die der Postprocessor schließlich durch eine Key-Value-Struktur individuell konfiguriert werden.

Damit die Postprocessoren aus der Konfiguration der jeweiligen Postprocessor-Klasse korrekt zugeordnet und erzeugt werden können, verfügt das Analysetool über eine interne Zuordnungstabelle: Eine Mapping-Datenstruktur verknüpft jeden eindeutigen Typnamen mit der entsprechenden Postprocessor-Klasse.

4.3.7.1 SkipInterSwappingSteps

Ein Postprocessor der `SkipInterSwappingSteps`-Klasse dient dazu, Analyseschritte herauszufiltern, die lediglich einen Zwischenschritt eines Dreiecktauschs zweier Arayelemente darstellen. Über den Parameter `array` wird festgelegt, auf welches vom `ArrayWatcher` beobachtete Array sich die Verarbeitung beziehen soll. Zwar besteht ein klassischer Dreiecktausch aus drei Anweisungen (temporäre Zwischenspeicherung, erste Zuweisung, zweite Zuweisung), jedoch verändern nur zwei dieser Anweisungen tatsächlich den Zustand des Arrays. Entsprechend werden vom `ArrayWatcher` auch nur zwei Analyseschritte erzeugt.

Damit diese beiden Schritte im späteren Verlauf korrekt als ein zusammenhängender Tauschvorgang und nicht als zwei voneinander unabhängige Ersetzungen interpretiert werden können, entfernt der Postprocessor den ersten dieser beiden Schritte. Im verbleibenden zweiten Schritt wird die Information des entfernten Vorgängerschritts eingebettet, um diesen nicht vollständig zu verlieren. Dies ermöglicht insbesondere der Webanwendung, beide ursprünglich betroffenen Codezeilen korrekt zu referenzieren, etwa zur Hervorhebung im Editor. Die Einbettung erfolgt über den in der Konfiguration definierten Schlüssel, unter dem der zuvor gelöschte Schritt im verbliebenen gespeichert wird.

Diese Vorverarbeitung ist notwendig, damit nachgelagerte Postprocessoren wie `KeepTrackOfItems` den Vorgang korrekt als Tausch erkennen und die Darstellung in der Visualisierung entsprechend umsetzen können.

Um einen Dreieckstausch zuverlässig zu erkennen, durchläuft der Postprocessor schrittweise alle Analyseschritte, die ein Ereignis des spezifizierten `ArrayWatcher` enthalten, und betrachtet jeweils Gruppen von drei aufeinanderfolgenden Schritten mit verändertem Arrayzustand.

Zunächst wird jeweils zwischen dem ersten und dem zweiten sowie zwischen dem zweiten und dem dritten Arrayzustand der Index ermittelt, an dem sich ein Wert verändert hat. Dabei wird vorausgesetzt, dass in jedem Analyseschritt höchstens ein Arrayelement verändert wird – eine Annahme, die nur dann verletzt werden könnte, wenn eine arrayverändernde Operation außerhalb des eingereichten Codes stattfindet, etwa in einer externen Hilfsfunktion. Da jedoch `#include`-Direktiven im einzureichenden Code untersagt sind, können solche externen Funktionen in diesem Kontext nicht eingebunden werden, wodurch diese Annahme als abgesichert gilt.

Ein Dreieckstausch wird dann erkannt, wenn drei zentrale Bedingungen erfüllt sind: Zum einen müssen sich die beiden festgestellten Indizes voneinander unterscheiden, was darauf hindeutet, dass zwei verschiedene Positionen im Array an aufeinanderfolgenden Schritten verändert wurden. Zum anderen müssen im zweiten der drei betrachteten Arrayzustände an genau diesen beiden Indizes identische Werte stehen – ein Hinweis darauf, dass ein Wert temporär dupliziert wurde. Schließlich muss jener Wert, der im ersten Arrayzustand an der ersten veränderten Position stand, im dritten Arrayzustand an der zweiten veränderten Position erscheinen. Diese Übereinstimmung lässt darauf schließen, dass ein Wert über eine Hilfsvariable getauscht wurde, wie es beim klassischen Dreieckstausch der Fall ist. In diesem Fall entfernt der Postprocessor den zweiten der drei Schritte aus dem Analyseergebnis, da er lediglich den temporären Zwischenschritt des Dreiecktauschs darstellt und für die spätere Interpretation in der Webanwendung nicht relevant ist.

Das beschriebene Vorgehen bringt ein potenzielles Problem mit sich: Wird der mittlere Schritt eines Dreiecktauschs gelöscht, so verschwinden damit auch alle weiteren Analyseinformationen, die von anderen Collectoren in diesem Schritt beigesteuert wurden. Dies kann dazu führen, dass bestimmte Ereignisse – etwa ein Funktionsrücksprung, der genau in diesem Schritt erfolgt – nicht mehr in der finalen Visualisierung erscheinen. Im Ergebnis wäre z. B. nicht nachvollziehbar, dass ein bestimmter Rückgabewert zur

Durchführung des Tauschs beigetragen hat. Auch wenn ein solches Szenario in der Praxis selten auftritt, kann es insbesondere dann problematisch werden, wenn mehrere Analyseebenen gleichzeitig betrachtet werden sollen.

4.3.7.2 KeepTrackOfItems

Ein Postprocessor der `KeepTrackOfItems`-Klasse verarbeitet die von `SkipInter-SwappingSteps` vorverarbeiteten Analyseschritte und ergänzt jene, die durch eine `ArrayWatcher`-Instanz erzeugt wurden, um Informationen, ob sich der jeweilige Analyseschritt um einen Tausch oder eine Ersetzung handelt, sowie ergänzende Informationen, die den Sachverhalt näher beschreiben.

Über den Parameter `array` wird angegeben, welcher Schlüssel im Analyseschritt auf das Ergebnis der relevanten `ArrayWatcher`-Instanz verweist und damit die Grundlage für die Verarbeitung durch den Postprocessor bildet. Im Fall eines Tauschs werden die beiden Indizes der betroffenen Elemente als Information ausgegeben. Handelt es sich um eine Ersetzung, werden der entsprechende Index sowie der alte und der neue Wert als Information ausgegeben. Die ermittelten Informationen, sowie der Diskriminator, ob es sich um einen Tausch oder eine Ersetzung handelt, werden unter dem konfigurierten `key` dem jeweiligen Analyseschritt hinzugefügt.

Das Ergebnis dient der Webanwendung als Grundlage für eine verbesserte Darstellung der Schritte des Sortieralgorithmus innerhalb der Visualisierung.

Zur Ermittlung von Tausch- und Ersetzungssereignissen wird jeder Analyseschritt mit folgendem Verfahren ausgewertet: Zwei aufeinanderfolgende Schritte, die ein Ergebnis des `ArrayWatchers` enthalten, werden miteinander verglichen. Zuerst wird der erste Index bestimmt, an dem sich die Werte der beiden Arrayzustände unterscheiden. Anschließend wird geprüft, ob ein weiterer abweichender Index vorhanden ist. Wird kein zweiter Unterschied festgestellt, handelt es sich um eine Ersetzung. Sind hingegen zwei unterschiedliche Positionen betroffen und wurden die jeweiligen Werte exakt gegeneinander getauscht, handelt es sich um einen Tausch.

4.3.8 Presets der bereitgestellten Aufgabenstellungen

Das Preset für den Bubblesort Algorithmus enthält die Konfiguration für Collectoren der Klassen `ArrayWatcher`, `CurrentScope`, `CurrentLine`, sowie Postprocessoren der Klassen `SkipInterSwappingSteps` und `KeepTrackOfItems`.

Das Preset für den Quicksort Algorithmus enthält die Konfiguration für Collectoren der Klassen `ArrayWatcher`, `RecursionWatcher`, `ArrayCompareOperationWatcher`, `CurrentScope`, `CurrentLine`, sowie Postprocessoren der Klassen `SkipInterSwappingSteps` und `KeepTrackOfItems`.

Das Preset für den Einfügealgorithmus in einen binären Suchbaum enthält die Konfiguration für Collectoren der Klassen `BinaryTreeWatcher`, `RecursionWatcher`, `CurrentScope` und `CurrentLine`, aber keine Postprocessoren.

4.4 Executor

Da das Analysetool als eigenständige CLI-Anwendung implementiert ist, besteht für die Webanwendung kein direkter Zugriff darauf. Um dessen Funktionalität dennoch nutzen zu können, wird der Executor als vermittelnde Schnittstelle eingesetzt. Er etabliert die Kommunikation zwischen Webanwendung und Analysetool.

Der Executor stellt einen HTTP-Endpoint bereit, über den Aufträge von der Webanwendung entgegengenommen werden. Ein Auftrag besteht aus einer JSON-Struktur, die angibt, ob eine Analyse oder ein Test durchzuführen ist, welche ID das zu verwendende Preset hat und welcher C++-Code von den Anwender:innen eingereicht wurde.

Nach Erhalt eines Auftrags ruft der Executor das referenzierte Preset aus der Preset-Datenbank ab und startet daraufhin einen Docker Container mit dem Analysetool-Image. Als Einstiegspunkt wird das CLI-Programm ausgeführt, wobei über Parameter festgelegt wird, ob sich der Vorgang auf eine Analyse oder einen Test bezieht, welches Preset verwendet wird und welcher C++-Code analysiert bzw. getestet werden soll.

4.4 Executor

Die Bearbeitung eines Auftrags beginnt mit dem Start des Analysetools und gilt als abgeschlossen, sobald dieses entweder ein Ergebnis oder eine Fehlermeldung ausgegeben hat und sich daraufhin automatisch beendet.

Während der Bearbeitung eines Auftrags bleibt die HTTP-Verbindung zwischen Webanwendung und Executor, über die der Auftrag ursprünglich eingereicht wurde, dauerhaft geöffnet. Für die Übertragung der Antwortdaten nutzt der Executor das Verfahren des Transfers mittels „chunked encoding“ (implementiert über den HTTP-Header Transfer-Encoding: chunked). Dadurch wird eine fortlaufende Übermittlung einzelner Datenabschnitte ermöglicht, die funktional einem Streamingverhalten entspricht. Statusmeldungen können so in Echtzeit an den Client übermittelt werden.

Ein naheliegenderer Ansatz zur Realisierung einer solchen Kommunikation wäre die Verwendung einer WebSocket-Verbindung. Diese bietet eine permanente bidirektionale Verbindung und eignet sich ebenfalls zur Übertragung von Daten in Abschnitten. Im Rahmen der Entwicklung wurde jedoch bewusst auf WebSockets verzichtet, da für diesen Anwendungsfall lediglich ein einmaliger Aufruf durch den Client erfolgt und anschließend nur noch eine einseitige Kommunikation vom Server aus notwendig ist. Das zugrunde liegende Kommunikationsmodell bleibt dadurch konzeptionell näher an einem klassischen HTTP-Request-Response-Schema. Ohne die Integration der Statusmeldungen würde die Verbindung einem herkömmlichen, synchronen Ablauf entsprechen, was die Implementierung und das Verständnis der Interaktion vereinfacht.

Ein alternativer Ansatz bestünde darin, dass der Analyseprozess beim Eingang eines Auftrags eine eindeutige ID zurückliefert. Über diese ID könnte der Client in regelmäßigen Abständen den aktuellen Verarbeitungsstand abfragen, einschließlich bereits empfangener Statusmeldungen sowie eines möglichen Fehlers oder Ergebnisses. Dieser Ansatz würde jedoch die Einführung einer persistenten Speicherlösung voraussetzen, um den Zustand jedes Auftrags zwischenzuspeichern. Zudem müssten gespeicherte Ergebnisse nach einer gewissen Zeit wieder gelöscht werden, um unnötige Ressourcenbindung zu vermeiden. Darüber hinaus ist ein solches Polling-Verfahren im Vergleich zu einer offenen Verbindung weniger effizient, da es zusätzliche Anfragen erzeugt und zu einer höheren Latenz führen kann. Aus diesen Gründen wurde entschieden, auf den Erhalt der ursprünglichen HTTP-Verbindung zurückzugreifen, um die Zuordnung von

Statusmeldungen, Fehlern oder Ergebnissen eindeutig dem anfragenden Client zu ermöglichen. Die Identifikation erfolgt somit implizit über die bestehende Verbindung, wodurch keine separate Auftrags-ID oder Zwischenspeicherung erforderlich ist. Der Zustand des Analyseprozesses wird direkt und in Echtzeit über diese Verbindung an den jeweiligen Client übermittelt.

Diese Statusmeldungen bestehen aus einzeiligen JSON-Objekten, die jeweils mit einem Zeilenumbruch abgeschlossen und in separaten Chunks übertragen werden. Dieses Format ist auch unter dem Namen *NDJSON* bekannt, einem einfachen Verfahren zur sequentiellen Übertragung von JSON-Nachrichten über eine Textverbindung [38]. Eine alternative Methode zur Abgrenzung einzelner Nachrichten stellt der in RFC 7464 spezifizierte Ansatz dar [39], bei dem jede JSON-Nachricht mit dem sogenannten *Record Separator* (RS, Unicode U+001E) eingeleitet wird.

Sobald die Bearbeitung abgeschlossen ist, wird ein weiteres JSON-Objekt übertragen, das entweder das Analyse- bzw. Testergebnis oder Informationen über einen aufgetretenen Fehler enthält. Die Art der Nachricht – Status, Resultat oder Fehler – wird über ein explizites Feld im jeweiligen JSON-Objekt spezifiziert.

Alle ausgehenden Nachrichten werden vom Analysetool über die Standardausgabe `stdout` ausgegeben. Der Executor liest diese Ausgabe fortlaufend aus und leitet sie unverändert an den Client weiter. Auch hier erfolgt die Ausgabe als einzeilige, durch Zeilenumbrüche terminierte JSON-Objekte.

Tritt eine Ausgabe über die Standardfehlerausgabe `stderr` auf, wird der Auftrag sofort abgebrochen. Die über `stderr` ausgegebene Fehlermeldung wird vom Executor übernommen, an den Client übermittelt und als Backend-Fehler klassifiziert.

Der Executor wurde als *Node.js*-Anwendung implementiert und nutzt *Express.js* [40] zur Bereitstellung der Webserver-Funktionalität. Er läuft innerhalb eines Docker Containers mit erweiterten Rechten, wodurch er in der Lage ist, eigenständig weitere Docker Container für die Ausführung der Analysetool-Instanzen zu starten.

Der Executor ist so konzipiert, dass er mehrere Aufträge parallel bearbeiten kann. Eine interne Zählmechanik begrenzt dabei die Anzahl gleichzeitig laufender Aufträge, um

eine Überlastung des Hostsystems durch zu viele parallele Anfragen zu verhindern. Aktuell ist die maximale Anzahl gleichzeitiger Aufträge auf fünf begrenzt. Dieser Wert ist im Quellcode als hartkodierte Konstante definiert und kann nicht über eine Konfigurationsdatei angepasst werden. Sobald die maximale Anzahl erreichter Aufträge überschritten würde, lehnt der Executor neue Anfragen mit einem Fehler ab und informiert darüber, dass gegenwärtig keine weiteren Aufträge angenommen werden können.

Durch das Starten des Analysetools innerhalb eines Docker Containers wird die Sicherheit bei der Ausführung von potenziell unvertrauenswürdigem Code aus der Webanwendung deutlich erhöht. Die Containerumgebung fungiert dabei als Sandbox, die einen Ausbruch aus dem isolierten Ausführungskontext erheblich erschwert.

Anstelle des Root-Users wird im Container ein eigener User eingerichtet, der ausschließlich über die notwendigen Dateiberechtigungen für den Workspace mit den Python-Quelltexten verfügt. Der Internetzugang des Containers ist deaktiviert, um externe Kommunikation zu verhindern. Zusätzlich werden mit der Option `--cap-drop=ALL` sämtliche standardmäßig erlaubten Linux-Fähigkeiten des Containers entzogen. Dazu zählen unter anderem die Fähigkeit, Dateiberechtigungen zu umgehen (`DAC_OVERRIDE`) oder Prozess-UIDs zu verändern (`SETUID`). Eine vollständige Liste der betroffenen Fähigkeiten ist in der Docker-Dokumentation einsehbar [41].

Darüber hinaus wird die maximale Arbeitsspeichernutzung des Containers auf 64 Megabyte begrenzt, um eine übermäßige Belastung der Serverressourcen zu vermeiden. Die CPU-Nutzung ist so limitiert, dass sie maximal der Leistung eines einzelnen CPU-Kerns entspricht. Die Anzahl gleichzeitig laufender Prozesse ist auf 40 beschränkt, um Fork-Bombs zu verhindern. Ebenso ist die Anzahl geöffneter Dateihandles auf 40 limitiert, um exzessive Dateizugriffe als potenzielles Angriffsvektor auszuschließen.

4.5 Reverse Proxy

Um die Webanwendung und den Executor über das Internet erreichbar zu machen, wird ein Reverse Proxy eingesetzt. Dieser ermöglicht es, beide Dienste über einen einzigen Port bereitzustellen, sodass lediglich eine Portfreigabe im System erforderlich

ist. Gleichzeitig schafft diese Architektur die Grundlage dafür, zukünftig weitere Dienste über denselben Reverse Proxy anzubinden, ohne zusätzliche Ports konfigurieren zu müssen.

Ein weiterer Vorteil besteht darin, dass lediglich eine einzige SSL-Verschlüsselung auf dem Port des Reverse Proxys eingerichtet werden muss, um die Kommunikation zwischen Client und Server abzusichern. Andernfalls wäre es erforderlich, für jeden einzelnen Dienst eine eigene SSL-Konfiguration vorzunehmen, was den administrativen Aufwand erhöht und die Systemkonfiguration komplexer gestaltet.

Zudem entfällt die Notwendigkeit, eine CORS-Policy für die Kommunikation zwischen Webanwendung und Executor zu definieren, da durch die gemeinsame Nutzung von Port und Domain keine Cross-Origin-Anfragen auftreten.

Der Reverse Proxy ist als *NGINX*-Server implementiert und leitet eingehende Anfragen wahlweise an die Webanwendung oder den Executor weiter. Die Entscheidung über die Weiterleitung erfolgt anhand des Pfad-Anteils der vom Client gesendeten URL. Anfragen, deren Pfad mit /api beginnt, werden an den Executor weitergeleitet. Alle übrigen Anfragen werden an die Webanwendung übergeben.

Darüber hinaus ist der *NGINX*-Reverse-Proxy so konfiguriert, dass er WebSocket-Verbindungen unterstützt. Diese Funktionalität wird insbesondere während der Entwicklungsphase für das Hot-Reload-Feature von *Next.js* benötigt. Obwohl auch *Next.js* als Reverse Proxy eingesetzt werden kann, fiel die Entscheidung zugunsten von *NGINX*, da dieser eine klarere Trennung der Dienste ermöglicht und zusätzlich die Möglichkeit bietet, das interne Pufferverhalten zu deaktivieren. Dies ist erforderlich, um eine verzögerungsfreie Echtzeitkommunikation zwischen Client und Executor zu gewährleisten.

4.6 Preset-Datenbank

Die Preset-Datenbank stellt zwei HTTP-Endpunkte bereit: einen zum Laden eines vollständigen Presets und einen weiteren zum Laden eines Templates eines Presets. Letzteres wird zunächst vom Preset-Datenbank-Service vorverarbeitet.

4.6 Preset-Datenbank

Der Preset-Endpunkt wird vom Executor benötigt, um anhand der von der Webanwendung übermittelten Preset-ID das zugehörige Preset ermitteln zu können. Der Template-Endpunkt hingegen liefert das entsprechende Codegerüst, das in der Webanwendung angezeigt und bearbeitet werden soll.

Dabei werden bearbeitbare sowie schreibgeschützte Abschnitte des initialen Codegerüsts identifiziert und in einem strukturierten JSON-Format ausgegeben. Dieses Format dient der Webanwendung als Grundlage, um den Codeeditor mit dem vorbereiteten Code zu füllen und die editierbaren Bereiche entsprechend auszuweisen.

Ein Preset ist eine Konfiguration, mit der das Analysetool gesteuert wird, um eine Analyse oder einen Test durchzuführen. In der Preset-Datenbank ist jedes Preset als Sammlung mehrerer Dateien organisiert, die in einem Verzeichnis auf dem Dateisystem des Preset-Datenbank-Services abgelegt sind.

Die Ordnerstruktur folgt dabei einem festen Schema: Jedes Preset befindet sich im Verzeichnis config/<preset-id>, wobei <preset-id> den eindeutigen Namen des jeweiligen Presets repräsentiert.

Die Datei config/<preset-id>/template.cpp enthält das Template, das Anwender:innen in der Webanwendung als Codegerüst im Codeeditor angezeigt bekommt. Innerhalb dieses Templates markieren spezielle Kommentarstrukturen, welche Bereiche bearbeitet werden dürfen und welche schreibgeschützt sind.

Die Konfiguration für den Analysevorgang ist in der Datei config/<preset-id>/analysis/analysis.json hinterlegt. Sie definiert die zu verwendenden Collectoren und Postprocessoren sowie den Namen der Einstiegsfunktion, die das Analysetool analysieren soll – üblicherweise die von Anwender:innen zu implementierende Funktion.

Der zugehörige Boilerplate-Code für die Analyse befindet sich in config/<preset-id>/analysis/main.cpp. Diese Datei stellt die int main()-Funktion bereit und ruft die von Anwender:innen implementierte Einstiegsfunktion mit fest definierten Eingabewerten auf. Diese Werte sind Bestandteil der Aufgabenstellung, die über die Webanwendung vermittelt wird.

Die Definition der Testfälle erfolgt in `config/<preset-id>/test/testcases.json`. Hier wird zwischen öffentlichen und privaten Testfällen unterschieden, die in der Webanwendung unterschiedlich visualisiert werden.

Der Boilerplate-Code für den Test befindet sich in `config/<preset-id>/test/main.cpp`. Diese Datei stellt ebenfalls eine `int main()`-Funktion bereit, ruft jedoch die Einstiegsfunktion der Anwender:innen mit Eingabewerten auf, die über die Standardeingabe `stdin` bereitgestellt werden. Die Ausgaben der Funktion werden über `stdout` ausgegeben, sodass das Analysetool sie zur Auswertung erfassen kann. Die Eingabe- und Ausgabewerte entnimmt das Analysetool dabei aus der zuvor genannten Datei `config/<preset-id>/test/testcases.json`.

4.7 Containerisierung

Jede Teilkomponente des Systems ist als eigenständiger Docker Container umgesetzt. Dies fördert die Entkopplung der Komponenten untereinander und vereinfacht deren Konfiguration sowie Wartung.

Durch den Einsatz von *Docker Compose* können sämtliche Komponenten gemeinsam und mit einem einzigen Befehl gestartet werden. Dies erleichtert insbesondere die Bereitstellung des Projekts auf einem Server oder in Entwicklungsumgebungen anderer Beteiligter, da das System mit minimalem Aufwand in kürzester Zeit lauffähig ist.

Der *develop*-Modus von *Docker Compose* unterstützt darüber hinaus die Synchronisation von Quelltextänderungen mit den laufenden Services. Änderungen können so ohne manuelle Zwischenschritte übernommen werden. Bei Bedarf lassen sich betroffene Services automatisch neu starten oder vollständig neu bauen.

Wie bereits in Abschnitt 4.3 erläutert, trägt Docker außerdem dazu bei, das Analysetool innerhalb eines isolierten Containers mit spezifischen Sicherheitsbeschränkungen auszuführen.

5 Details der technischen Realisierung

In den folgenden Unterkapiteln werden exemplarische Codeausschnitte gezeigt, um gezielt einzelne Implementierungsaspekte hervorzuheben. Der Fokus liegt dabei auf den jeweils relevantesten Bestandteilen der Umsetzung. Zu jeder Backendkomponente wird zudem jeweils das zugrundeliegende Dockerfile vorgestellt.

Zur besseren Lesbarkeit wurden die Ausschnitte redaktionell angepasst. Dies umfasst unter anderem Auslassungen durch Kommentare wie `/* ... */` sowie das Weglassen von Fehlerbehandlungen, sodass in der Regel nur der erwartete Programmablauf dargestellt wird.

Teile des hier vorgestellten Codes wurden auf Basis konkreter Anweisungen mit Unterstützung des KI-gestützten Tools *ChatGPT* [42] erstellt.

5.1 Webanwendung / Webserver

Im Folgenden wird die Umsetzung des Editors erläutert – insbesondere die Hervorhebung einzelner Zeilen und Codebereiche. Ergänzend dazu wird beschrieben, wie die Kursstruktur verwaltet wird.

5.1.1 Editor

Der eingebundene Codeeditor wird mit dem *Monaco Editor* bereitgestellt. Da das Projekt in *React* umgesetzt ist, kommt hierfür das *monaco/react* npm-Package [12] zum Einsatz.

Im Folgenden ist dargestellt, wie der Codeeditor in einer eigenen Komponente eingebettet wird:

```
1 import MonacoEditor from '@monaco-editor/react';
2 // ...
3 export default function Editor(* ... *) {
4     // ...
5     return (
6         <MonacoEditor
7             defaultValue={defaultValue}
8             onMount={handleEditorDidMount}
9             onChange={handleChange}
10            height="70vh"
11            language="cpp"
12            options={{
13                minimap: {
14                    enabled: false
15                },
16                scrollBeyondLastLine: false,
17                tabSize: 4
18            }} /> )
    }
```

In den Attributen `language` und `options` werden Syntax-Highlighting und die Anzahl von Leerzeichen pro Tabulator konfiguriert. Das Attribut `defaultValue` enthält den initialen Code, der im Editor angezeigt wird. Dabei handelt es sich um das von der Preset-Datenbank vorverarbeitete Template, in dem die editierbaren Codebereiche zunächst leer belassen wurden. Diese Bereiche werden in nachfolgenden Verarbeitungsschritten befüllt.

Die Callback-Funktion des `onMount`-Properties wird aufgerufen, sobald der *Monaco Editor* vollständig geladen ist und im DOM zur Verfügung steht. Als Parameter werden Kontrollobjekte übergeben, mit denen sich der Editor nachträglich manipulieren lässt, etwa zur Anpassung des Inhalts oder zur Einführung von Einschränkungen.

In dieser Initialisierungsfunktion wird mithilfe des npm-Packages *constrained-editor-plugin* der Editor schreibgeschützt gesetzt und die zuvor definierten bearbeitbaren Bereiche aktiviert:

5.1 Webanwendung / Webserver

```
1 import { constrainedEditor } from "constrained-editor-plugin";
2 /* ... */
3 export default function Editor({ template, /* ... */ }) {
4     const handleEditorDidMount = useCallback((editor: EditorType, monaco:
5         Monaco) => {
6         const model = editor.getModel()
7         const constrainedInstance = constrainedEditor(monaco)
8         constrainedInstance.initializeIn(editor)
9         const modifyableSections = template.ranges.map((range, index) => ({
10             range: [range.start.line, range.start.column, range.end.line,
11                 range.end.column],
12             allowMultiline: true,
13             label: `body${index}`
14         }))
15         constrainedInstance.addRestrictionsTo(model, modifyableSections)
16         /* ... */
17     })
18     /* ... */
19 }
```

In den Zeilen 5 und 7 wird der Editor mithilfe des Plugins *constrained-editor-plugin* schreibgeschützt gesetzt. Die editierbaren Bereiche werden in den Zeilen 8 bis 12 definiert, wobei das übergebene Template aus der Preset-Datenbank als Grundlage dient. Die Templatedaten werden lediglich in das vom Plugin geforderte Format überführt. In Zeile 13 werden die Bereiche auf das aktuelle Editor-Modell angewendet.

Jeder bearbeitbare Bereich erhält über das Attribut `label` eine eindeutige Kennung. Diese Labels ermöglichen es, die Inhalte der jeweiligen Bereiche im weiteren Verlauf gezielt zu modifizieren oder auszulesen:

```
1 model.updateValueInEditableRanges(
2     Object.fromEntries(
3         initialFunctionBodies.map(
4             (body, index) => [`body${index}`, body]
5         )))
6 
```

Die Variable `initialFunctionBodies` wurde zuvor entweder mit Daten aus dem Local Storage oder – falls dort keine vorhanden sind – mit dem Default-Code aus der

Preset-Datenbank initialisiert.

Die Methode `updateValueInEditableRanges` erwartet eine Key-Value-Struktur, in der jeder Key einem zuvor vergebenen `label` entspricht und der zugehörige Value den neuen Inhalt für den jeweiligen bearbeitbaren Codebereich darstellt. Auf diese Weise kann gezielt der Inhalt einzelner editierbarer Segmente aktualisiert werden.

Um die einzelnen `#pragma region`-Bereiche im Editor automatisch einzuklappen, wird die Einklapp-Aktion des Editors gezielt mit den entsprechenden Zeilennummern aufgerufen. Diese Zeilennummern werden aus dem Quelltext extrahiert, indem alle Zeilen identifiziert werden, die mit dem String `#pragma region` beginnen:

```
1 editor.trigger(null, 'editor.fold', {  
2     selectionLines: model.getValue().split('\n')  
3         .map(([line, index]) => [line, index])  
4         .filter(([line, index]) => line.startsWith('#pragma region '))  
5         .map(([line, index]) => index)  
6 })
```

Die im Editor hervorzuhebenden Zeilen, die aufgrund ihrer Ausführung im aktuell dargestellten Schritt der Visualisierung markiert werden sollen, werden über einen globalen Zustands-Store bereitgestellt. Sobald sich dieser Zustand ändert – beispielsweise, weil die Visualisierung auf einen neuen Schritt gewechselt hat –, übernimmt der folgende Code die Aktualisierung der Darstellung im Editor:

```
1 let previousLineDecorations = null;  
2 /* ... */  
3 // sobald sich `activeLines` ändert:  
4 previousLineDecorations?.clear()  
5 if (activeLines.length) {  
6     previousLineDecorations = highlightLines(editor, activeLines)  
7     editor.revealLine(activeLines[0])  
8 }  
9 /* ... */  
10 function highlightLines(editor: EditorType, lines: number[]) {  
11     return editor.createDecorationsCollection(lines.map(line => ({  
12         range: {  
13             startLineNumber: line,
```

```
14         startColumn: 1,
15         endLineNumber: line,
16         endColumn: 1
17     },
18     options: {
19         isWholeLine: true,
20         className: 'code-editor__active-line',
21         lineNumberClassName: 'code-editor__active-line',
22         linesDecorationsClassName: "code-editor__active-line-symbol"
23     }
24 });
25 }
```

`previousLineDecorations` dient dazu, die zuvor gesetzten Dekorationen im Editor zwischenzuspeichern, sodass sie bei einer Zustandsänderung über die Methode `clear` [43] entfernt werden können. Auf diese Weise wird verhindert, dass alte Markierungen bestehen bleiben, wenn neue Zeilen hervorgehoben werden sollen.

`activeLines` ist ein Array von Ganzzahlen, das die aktuell hervorzuhebenden Zeilennummern enthält. Die Methode `revealLine` [44] sorgt dafür, dass der Editor zur angegebenen Zeile scrollt, falls sie derzeit nicht im sichtbaren Bereich liegt.

Die Funktion `highlightLines` transformiert jede dieser Zeilennummern in eine JSON-Struktur, die dem Editor mitteilt, wie die betreffende Zeile zu dekorieren ist. Dabei wird jede betroffene Zeile als Markierung mit einer Länge von null Zeichen modelliert, wobei durch die Option `isWholeLine` sichergestellt wird, dass die gesamte Zeile hervorgehoben wird.

Die CSS-Klassen, die in den Zeilen 20 bis 22 angegeben sind, werden dann von der Methode `createDecorationsCollection` [45] verwendet, um die Zeile selbst, die zugehörige Zeilennummer sowie der Container der Zeilennummer visuell hervorzuheben. Dadurch erscheinen alle betroffenen Zeilen in blauer Hintergrundfarbe, ergänzt durch einen Rechtspfeil neben der Zeilennummer.

Auf ähnliche Weise wie bei der Hervorhebung ausgeführter Codezeilen werden auch in der Quicksort-Visualisierung die Vergleichsoperanden visuell markiert – also jene Elemente, die im aktuellen Schritt miteinander verglichen werden. Auch hierfür wird

ein Attribut im globalen Store bereitgestellt, das von der Visualisierung bei jedem Schritt aktualisiert wird:

```
1 function highlightTheRanges(editor: EditorType, highlightRanges:  
2   → HighlightRange[]) {  
3   return editor.createDecorationsCollection(highlightRanges.map(highlight =>  
4     ({  
5       range: {  
6         startLineNumber: highlight.range.start.line,  
7         startColumn: highlight.range.start.column,  
8         endLineNumber: highlight.range.end.line,  
9         endColumn: highlight.range.end.column  
10      },  
11      options: {  
12        className: cn(highlight.className, 'opacity-40'),  
13        hoverMessage: { value: highlight.hoverMessage }  
14      }  
15    }));  
16  }
```

In Zeile 10 kommt die Hilfsfunktion `cn` zum Einsatz. Diese Funktion bündelt mehrere CSS-Klassen zu einem einzigen, konsistenten Klassennamenstring, der von *Tailwind CSS* interpretiert werden kann. Sie wird hier verwendet, um den vorgegebenen Klassennamen `highlight.className` zusätzlich um eine Transparenz zu ergänzen, damit die Lesbarkeit erhöht wird. Im Fall von Vergleichsoperationen enthält `highlight.className` eine CSS-Klasse, die die betreffende Codestelle farblich hervorhebt.

Darüber hinaus wird über das Attribut `hoverMessage` ein erläuternder Text definiert, der angezeigt wird, sobald sich der Mauszeiger über der Markierung befindet. Dieser Text enthält den tatsächlichen Wert des Operanden und trägt dazu bei, die interne Logik des Sortieralgorithmus für Anwender:innen nachvollziehbar darzustellen.

Um bei Syntaxfehlern die betroffenen Codestellen im Editor visuell zu kennzeichnen, werden diese wie folgt unterschlängelt und mit einer entsprechenden Fehlermeldung versehen:

```
1 editor.removeAllMarkers('compiler')
2 editor.setModelMarkers(editor.getModel(), 'compiler', markers.map(marker =>
→ ({...marker, severity: severityMap[marker.severity] })))
```

Die Variable `markers` enthält dabei die entsprechenden Codebereiche samt Fehlermeldungstext und Typ – ob es etwa eine Warnung oder ein Error ist –, sodass sie direkt von der Methode `setModelMarkers` [46] verarbeitet werden kann.

Im Folgenden wird gezeigt, wie die Datenstruktur eines solchen Markers aufgebaut ist:

```
1 interface EditorMarker {
2     startLineNumber: number
3     startColumn: number
4     endLineNumber: number
5     endColumn: number
6     message: string
7     severity: 'error' | 'warning' | 'note' | 'info'
8 }
```

Die Eigenschaft `severity` muss zuvor in einen entsprechenden numerischen Wert überführt werden. Dafür wird das folgende Mapping benutzt [47]:

```
1 const severityMap = { error: 8, warning: 4, info: 2, hint: 1 }
```

Mit der Methode `removeAllMarkers` [48] werden alle zuvor gesetzten Marker entfernt, da sie veraltet sind und durch die neuen ersetzt.

5.1.2 Speicherung der Kursstruktur

Die hierarchische Kursstruktur, bestehend aus Kurs, Problemkategorie und Aufgabenstellung, wird auf dem Webserver in Form einer hartkodierten JSON-Struktur verwaltet. Zusätzlich enthält diese Struktur die konkrete Aufgabenstellung sowie die zugehörige Preset-ID `presetName`.

```
1 const db = [
2   {
3     id: "ads",
4     title: "Algorithmen und Datenstrukturen (ADS)",
5     description: "Master algorithms and data structures",
6     progress: 0,
7     chapters: [
8       {
9         id: "sorting",
10        title: "Sorting",
11        progress: 0,
12        tasks: [
13          {
14            id: "bubble-sort",
15            presetName: 'sort',
16            title: "Bubble Sort",
17            status: 'completed',
18            description: `Aufgabenstellung des Bubblesorts`
19          },
20          {
21            id: "insertion-sort",
22            presetName: 'sort',
23            title: "Insertion Sort",
24            status: 'completed',
25            description: 'Lorem Ipsum...'
26          },
27          // ...
28        ]
29      },
30      {
31        id: "sorting-recursion",
32        title: "Sorting with Recursion",
33        progress: 0,
34        tasks: [
35          {
36            id: "quick-sort",
37            presetName: 'quick-sort',
38            title: "Quick Sort",
39            status: 'not-started',
```

```
40             description: "Aufgabenstellung des Quicksorts"
41         },
42         // ...
43     ]
44 },
45 // ..
46 ]
47 }
48 ]
```

5.1.3 Dockerfile

Das Docker-Image für den Webserver basiert auf einem *Node*-Base-Image, wobei alle benötigten Pakete während des Build-Prozesses installiert werden. Das zugehörige Dockerfile ist speziell für die Entwicklungsumgebung ausgelegt.

Werden Quellcodedateien geändert, synchronisiert die in Datei 5.6 definierte *Docker Compose*-Konfiguration diese Änderungen automatisch in den laufenden Container. Bei Änderungen an der package.json hingegen wird der Container vollständig neu gebaut, um sicherzustellen, dass die Abhängigkeiten korrekt aktualisiert werden.

```
1 FROM node:22-alpine
2
3 WORKDIR /app
4 COPY package*.json ./
5 RUN npm install
6 COPY . .
7 EXPOSE 80
8
9 CMD ["npm", "run", "dev", "--", "--port", "80"]
```

5.2 Analysetool

5.2.1 Template

Exemplarisch wird im Folgenden das Template für den Quicksort-Algorithmus vorgestellt. Dabei ist vorgesehen, dass sämtliche Codezeilen im Standardfall schreibgeschützt sind. Ausgenommen hiervon sind jene Zeilen, die sich zwischen den Markierungen `// <user-code-start>` und `// <user-code-end>` befinden. Diese Bereiche sind für Anwender:innen editierbar. Der innerhalb dieses Abschnitts vorgegebene Code dient als initiale Vorlage und wird angezeigt, solange der*die Anwender:in noch keine eigenen Änderungen vorgenommen hat:

```
1 #pragma region prototypes
2
3 void quickSort(int*, int);
4 void quickSortRecursive(int*, int, int);
5 int partition(int*, int, int);
6 // <user-code-start>
7 // add custom prototypes here:
8 void swap(int&, int&);
9
10 // <user-code-end>
11 #pragma endregion prototypes
12
13
14 void quickSort(int* arr, int n) {
15 // <user-code-start>
16     quickSortRecursive(arr, 0, n - 1); // already implemented for you ;)
17 // <user-code-end>
18 }
19
20 void quickSortRecursive(int* arr, int low, int high) {
21 // <user-code-start>
22     // todo: add your code here
23 // <user-code-end>
24 }
```

5.2 Analysetool

```
26 int partition(int* arr, int low, int high) {  
27     // <user-code-start>  
28     // todo: add your code here  
29     // <user-code-end>  
30 }  
31  
32  
33 #pragma region custom_functions  
34 // <user-code-start>  
35 // insert custom functions here:  
36  
37 void swap(int& a, int& b) {  
38     int temp = a;  
39     a = b;  
40     b = temp;  
41 }  
42  
43 // <user-code-end>  
44 #pragma endregion custom_functions
```

5.2.2 Aufbau eines Presets

Exemplarisch wird hier das Preset des Quicksorts vorgestellt. Im Top-Level verfügt es über entrypointFunction, was festlegt, wie der Name der Funktion ist, die von den Anwender:innen implementiert wird. collect und postProcess hingegen sind Listen für die Konfiguration der Collectoren und Postprocessoren:

Exemplarisch wird hier das Preset für den Quicksort-Algorithmus vorgestellt. Im obersten Abschnitt der Konfigurationsdatei befindet sich das Attribut entrypointFunction, das angibt, welche Funktion von Anwender:innen implementiert werden soll.

Die Felder collect und postProcess enthalten jeweils Listen mit den Konfigurationen der Collectoren und Postprocessoren, die während der Analyse ausgeführt werden sollen.

```
1  {  
2      "entrypointFunction": "quickSort",
```

```
3   "collect": [ /* ... */ ],
4   "postProcess": [ /* ... */ ]
5 }
```

In den folgenden Codeabschnitten werden die Konfigurationen der Collectoren im Detail betrachtet.

Zunächst wird ein Collector der Klasse `ArrayWatcher` konfiguriert. Sie ist dafür zuständig, das Array `arr` zu überwachen, dessen Länge in der Variable `n` gespeichert ist. Da die Initialisierung der Collectoren vor Ausführung des Debuggingschritts in Zeile 15 des vorgestellten Template-Codes stattfindet, entsprechen `arr` und `n` den Parametern, die an die Einstiegsfunktion übergeben werden:

```
14 void quickSort(int* arr, int n) {
15 // <user-code-start>
```

Die vom `ArrayWatcher` erfassten Daten werden unter dem Schlüssel `array` im Analyseergebnis abgelegt.

Ebenfalls konfiguriert wird ein First-Level-Collector der Klasse `ArrayCompareOperationWatcher`, die – analog zum `ArrayWatcher` – auf dasselbe Array angewendet wird. Zusätzlich kommt ein First-Level-Collector der Klasse `RecursionWatcher` zum Einsatz, der Funktionsaufrufe überwacht und insbesondere Ein- und Austritte dokumentiert.

Darüber hinaus werden die beiden Second-Level-Collectoren der Klassen `CurrentLine` und `CurrentScope` eingebunden:

```
1 {
2   "entrypointFunction": "quickSort",
3   "collect": [
4     {
5       "type": "arrayWatcher",
6       "parameters": {
7         "name": "arr",
8         "size": "n"
9       },
10      "key": "array"
11    },
12  ],
```

5.2 Analysetool

```
12  {
13      "type": "recursionWatcher",
14      "key": "recursion"
15  },
16  {
17      "type": "arrayCompareOperationWatcher",
18      "parameters": {
19          "name": "arr",
20          "size": "n"
21      },
22      "key": "arrayComparisons"
23  },
24  {
25      "type": "currentLine",
26      "key": "line"
27  },
28  {
29      "type": "currentScope",
30      "key": "scope"
31  }
32 ],
33 "postProcess": [ /* ... */ ]
34 }
```

Hier ist die Konfiguration der Postprocessoren dargestellt. Der Parameter array verweist dabei jeweils auf den Schlüssel des zuvor konfigurierten ArrayWatchers, der das relevante Array überwacht hat und dessen Ergebnisse nun von den Postprocessoren weiterverarbeitet werden sollen.

```
1  {
2      "entrypointFunction": "quickSort",
3      "collect": [ /* ... */ ],
4      "postProcess": [
5          {
6              "type": "skipInterSwappingSteps",
7              "parameters": {
8                  "array": "array"
9              },
10             "key": "skippedStep"
11         }
12     ]
13 }
```

```
11     },
12     {
13         "type": "keepTrackOfItems",
14         "parameters": {
15             "array": "array"
16         },
17         "key": "event"
18     }
19 ]
20 }
```

5.2.3 Beispiel eines Analyseergebnisses

Das Ergebnis einer Analyse erfolgt immer als Liste an Analyseschritten:

```
1 [
2     { /* Schritt 1 */ },
3     { /* Schritt 2 */ },
4     { /* Schritt 3 */ },
5     { /* Schritt 4 */ }
6
7     /* ...weitere Schritte ... */
8 ]
```

Im folgenden Codeabschnitt ist der erste Eintrag eines Quicksort Analyseergebnisses gezeigt:

```
1 [
2     {
3         "array": [ "5", "2", "7", "1", "8", "3", "6", "4" ],
4         "recursion": null,
5         "arrayComparisons": null,
6         "line": null,
7         "scope": {
8             "arr": {
9                 "value": "0x00007ffeacfc2960",
10                "type": "int *",
11                "isPointer": true,
```

5.2 Analysetool

```
12     "isReference": false
13 },
14     "n": {
15         "value": "8",
16         "type": "int",
17         "isPointer": false,
18         "isReference": false
19     },
20 },
21     "skippedStep": null,
22     "event": null,
23 },
24 /* ... weitere Schritte ...*/
25 ]
```

Der folgend gezeigte Analyseschritt zeigt einen Tausch zwischen zwei Arrayelementen:

```
1 [
2 /* ... vorherige Schritte ...*/
3 {
4     "array": [ "2", "5", "7", "1", "8", "3", "6", "4" ],
5     "recursion": null,
6     "arrayComparisons": null,
7     "line": 43,
8     "scope": {
9         "a": { /* ... */ },
10        "b": { /* ... */ },
11        "temp": { /* ... */ }
12    },
13     "skippedStep": {
14         "array": [ "2", "2", "7", "1", "8", "3", "6", "4" ],
15         "recursion": null,
16         "arrayComparisons": null,
17         "line": 42,
18         "scope": { /* ... */ },
19         "skippedStep": null
20    },
21     "event": {
22         "type": "swap",
```

```
23     "index1": 0,
24     "index2": 1
25   }
26 }
27 /* ... weitere Schritte ...*/
28 ]
```

Im Folgenden ist ein Analyseschritt dargestellt, der den Rücksprung aus der Funktion `partition` protokolliert hat:

```
1 [
2  /* ... vorherige Schritte ...*/
3  {
4    "array": null,
5    "recursion": {
6      "type": "step_out",
7      "from": "partition(int*, int, int)",
8      "to": "quickSortRecursive(int*, int, int)",
9      "returnValue": "3"
10    },
11    "arrayComparisons": null,
12    "line": 34,
13    "scope": {
14      "arr": { /* ... */ },
15      "low": { /* ... */ },
16      "high": { /* ... */ },
17      "pi": { /* ... */ }
18    },
19    "skippedStep": null,
20    "event": null,
21  }
22 /* ... weitere Schritte ...*/
23 ]
```

5.2.4 Anbindung von LLDB

In diesem Beispielcode wird der *LLDB*-Debugger initialisiert, ein Breakpoint an der zu implementierenden Einstiegsfunktion gesetzt und die Ausführung des Programms

5.2 Analysetool

gestartet. Anschließend wird auf das Erreichen des gesetzten Breakpoints gewartet. Der vorliegende Code wurde mit Unterstützung von ChatGPT umgesetzt:

```
1 def setup_debugger(function_name, executable_filename):
2     debugger = lldb.SBDebugger.Create()
3     debugger.SetAsync(False)
4
5     target = debugger.CreateTarget(executable_filename)
6
7     # Set Breakpoint at beginning of the algorithm
8     target.BreakpointCreateByName(function_name,
9         ↪ target.GetExecutable().GetFilename())
10
11    # Launch the process. Since we specified synchronous mode, we won't return
12    ↪ from this function until we hit the breakpoint at main
13    launch_info = lldb.SBLaunchInfo(None)
14    launch_info.SetLaunchFlags(1 << lldb.eLaunchFlagDisableASLR) # relevant for
15    ↪ some computer architectures
16    error = lldb.SBError() # captures error messages, check for it with
17    ↪ error.Fail()
18    process = target.Launch(launch_info, error)
19
20    thread = process.GetThreadAtIndex(0)
21    frame = thread.GetFrameAtIndex(0)
22
23
24    return frame, process, thread, debugger
```

Im folgenden Beispielcode wird eine Klasse definiert, mit der zur Laufzeit der Inhalt eines Arrays eingelesen werden kann. Dazu werden im Konstruktor der Variablenname des Pointers auf das Array, sowie der Variablenname der Variable, die die Anzahl der Elemente angibt, übergeben. Mit der Methode `get` kann anschließend das vollständige Array zur Laufzeit als Tupel ausgelesen werden. Die `get`-Methode liest dafür jeden Index des Arrays mithilfe der Methode `GetChildAtIndex` [49] separat aus.

```
1 class Array:
2     def __init__(self, frame, array_name, size_name):
3         size_var = frame.FindVariable(size_name)
4         self.array_var = frame.FindVariable(array_name)
```

```
5     self.size = int(size_var.GetValue(), 0)
6
7     def get(self):
8         values = (self.array_var.GetChildAtIndex(i, lldb.eDynamicCanRunTarget,
9             → True).GetValue()
10            for i in range(self.size))
11        return tuple(int(val, 0) if type(val) is int else val
12                    for val in values)
```

Im folgenden Beispielcode wird gezeigt, wie die lokalen Variablen eines gegebenen Frames mithilfe von `frame.GetVariables(...)` [50] ausgelesen werden können. Beim Auslesen lokaler Variablen werden Referenzen standardmäßig wie Pointer behandelt. Die Methode `GetValue` [51] liefert in diesem Fall die Adresse der referenzierten Variable zurück:

```
1 varlist = frame.GetVariables(True, True, False, False)
2 local_scope = {var.GetName(): {
3     'value': var.GetValue(),
4     'type': var.GetType().GetDisplayName(),
5     'isPointer': var.GetType().IsPointerType(),
6     'isReference': var.GetType().IsReferenceType(),
7 } for var in varlist}
```

Um stattdessen den tatsächlichen Wert der referenzierten Variable zu erhalten, kann folgende Ausdrucksform verwendet werden:

```
1 value = var.Dereference().GetValue() if var.GetType().IsReferenceType() else
→ var.GetValue()
```

5.2.5 CLI-Parameter mit JSON

Im folgenden Codebeispiel wird gezeigt, wie das Analysetool aufgerufen wird. Dabei wurden etliche Parameter ausgelassen, die den erstellten Docker-Container mit Restriktionen absichern. Im Abschnitt 5.3.3 wird näher auf diese Restriktionen eingegangen.

```
1 docker run --rm analysetool "{ ... Preset ... }" \" Code \\""
```

5.2.6 Validierung des übermittelten Codes

Im folgenden Beispielcode wird gezeigt, wie mithilfe des *clang*-Moduls die syntaktische Struktur eines Codes analysiert werden kann. Dazu wird zunächst der Quelltext geparsst. Anschließend extrahiert die Funktion `extractStructure` alle Top-Level-Einträge des Syntaxbaums und überführt sie in ein Set, wobei implementierte Funktionsrümpfe gezielt entfernt werden.

Ziel dieser Analyse ist es, die Struktur des von dem*der Anwender:in eingereichten Codes mit der des Templates zu vergleichen. Im Regelfall sollte das vom Template generierte Set eine Teilmenge des Sets des eingereichten Codes sein:

```
1  from clang import cindex
2
3  # parse template code into syntax tree
4  index = cindex.Index.create()
5  template_tree = index.parse('/tmp/template.cpp', args=['-std=c++17']).cursor
6
7  # parse user code into syntax tree
8  index = cindex.Index.create()
9  usercode_tree = index.parse('/tmp/user.cpp', args=['-std=c++17']).cursor
10
11 # generate sets of top level entries of both codes
12 template_structure = set(extract_structure(template_tree))
13 user_structure = set(extract_structure(usercode_tree))
14
15 # check if it passes the condition
16 if not template_structure.issubset(user_structure):
17     raise CodeException("The structure of the user code does not match the
    ↪ expected structure.")
```

Die Funktion `extract_structure` ist wie folgt definiert:

```
1  from clang import cindex
2
3  def extract_structure(node, remove_function_implementation=True):
4      items = []
5      for child in node.get_children():
6          spelling = ' '.join(c.spelling for c in child.get_tokens())
```

```
7     if remove_function_implementation and child.kind ==  
8         cindex.CursorKind.FUNCTION_DECL:  
9             for c in child.get_children():  
10                 if c.kind == cindex.CursorKind.COMPOUND_STMT:  
11                     compound = ' '.join(c.spelling for c in c.get_tokens())  
12                     spelling = spelling[:-len(compound)]  
13                     break  
14             items.append(spelling)  
15         return items
```

5.2.7 Kompilierung

Das Analysetool übernimmt die über die Kommandozeile übergebenen Quelltexte und speichert sie temporär im Verzeichnis /tmp ab. Anschließend werden diese Dateien mithilfe der *Clang-CLI* [52] kompiliert und im nächsten Schritt gelinkt:

```
1 # compilation:  
2 clang++-19 -fdiagnostics-parseable-fixits -fno-caret-diagnostics -g -c  
3     ↳ <sourcecodefile>.cpp -o <sourcecodefile>.o  
4  
5 # linking:  
6 clang++-19 -fdiagnostics-parseable-fixits -fno-caret-diagnostics -g <file1>.o  
7     ↳ <file2>.o ... -o a.out
```

5.2.8 Syntaxfehler

Gibt der oben genannte Kompilierungsvorgang eine Ausgabe auf der Standardfehlerausgabe `stderr` zurück, wird diese Ausgabe geparsst und in ein JSON-Format überführt. Da die Fehlermeldungen in `stderr` lediglich eine Startposition angeben, jedoch keinen vollständigen Codebereich, wird im resultierenden JSON-Dokument zunächst ein Bereich mit einer Länge von einem Zeichen hinterlegt.

Trotzdem wird bereits jetzt ein vollständiger Codebereich (bestehend aus Start- und Endposition) im JSON-Format vorgesehen. Dies ermöglicht es, künftig eine genauere

5.2 Analysetool

Fehlerextraktion – etwa mit exakten Angaben von Start- und Endzeile sowie Spaltenpositionen – zu integrieren, ohne dabei das JSON-Schema anpassen zu müssen. Die Webanwendung kann somit bereits auf dieses Format reagieren, ohne bei einer späteren Erweiterung geändert werden zu müssen.

Eine mögliche Fehlerausgabe des Compilers ist im folgenden Ausschnitt dargestellt:

```
1 /tmp/main.cpp:16:15: error: use of undeclared identifier
  ↵ 'thisVariableDoesntExist'
2 /tmp/main.cpp:18:45: error: expected ';' after expression
```

Die Extraktion der relevanten Informationen aus einer solchen Fehlermeldung ist mit der folgenden Funktion umgesetzt. Diese wurde mit Unterstützung von ChatGPT erstellt:

```
1 def parse_clang_output(output, restrict_to_filename: str):
2     regex = r'^(.??:(\d+):(\d+): (\w+): (.+)$'
3     matches = re.finditer(regex, output, re.MULTILINE)
4     markers = []
5
6     for match in matches:
7         file, line, column, level, message = match.groups()
8         if file != restrict_to_filename:
9             continue
10        markers.append({
11            "startLineNumber": int(line),
12            "startColumn": int(column),
13            "endLineNumber": int(line),
14            "endColumn": int(column) + 1,
15            "message": message.strip(),
16            "severity": level or 'info'
17        })
18
19    return markers
```

Das Ergebnis der Funktion zur zuvor gezeigten Fehlermeldung sieht folgendermaßen aus:

```
1 [
2   {
3     "startLineNumber": 16,
```

```
4     "startColumn": 15,
5     "endLineNumber": 16,
6     "endColumn": 16,
7     "message": "use of undeclared identifier 'thisVariableDoesntExist'",  
8     "severity": "error"
9 },
10 {
11     "startLineNumber": 18,
12     "startColumn": 45,
13     "endLineNumber": 18,
14     "endColumn": 46,
15     "message": "expected ';' after expression",
16     "severity": "error"
17 }
18 ]
```

5.2.9 Collectoren

Bekanntmachung der Collectoren Collector-Klassen werden in einer zentralen Datei wie folgt registriert:

```
1 collectors = {
2     'arrayWatcher': ArrayWatcher,
3     'currentScope': CurrentScope,
4     'currentLine': CurrentLine,
5     'recursionWatcher': RecursionWatcher,
6     'compareOperationWatcher': CompareOperationWatcher,
7     'arrayCompareOperationWatcher': ArrayCompareOperationWatcher,
8     'binaryTreeWatcher': BinaryTreeWatcher,
9 }
```

Übergabe der Konfiguration in das Pythonprogramm Um die in der Konfiguration definierten Collectoren in Python zu erzeugen, wird die Liste der Collector-Konfigurationen vom CollectorManager übernommen. Für jede Konfiguration wird die Methode Collector.from_dict aufgerufen, wodurch der jeweilige Collector erstellt und in eine interne Liste aufgenommen wird.

5.2 Analysetool

Die Methode `from_dict` ermittelt anhand eines zuvor aufgebauten Mappings vom Typnamen zur zugehörigen Collector-Klasse die passende Implementierung. Anschließend wird der generische Collector-Klassenkonstruktor mit der übergebenen Konfiguration aufgerufen.

Die enthaltenen Parameter werden von dem Konstruktor schließlich an die `setup`-Funktion übergeben, die von jeder Collector-Klassen-Implementierung individuell definiert wird, um die Konfigurationsdaten weiterzuverarbeiten (siehe z. B. `CurrentLine`).

```
1 collect_manager = CollectorManager.from_dict(collect_configs, frame, tokens)
2
3 class CollectorManager:
4     def from_dict(config, frame: SBFrame, tokens: cindex.Token):
5         collectors = [Collector.from_dict(c, frame, tokens) for c in config]
6         return CollectorManager(collectors)
7     # ...
8
9 class Collector:
10    def from_dict(config, frame: SBFrame, tokens: cindex.Token):
11        from .register import get_collector
12        collector_name = config['type']
13        collector = get_collector(collector_name)
14        key = config['key']
15        parameters = config.get('parameters', {})
16        return collector(key, parameters, frame, tokens)
17
18    def __init__(self, key, parameters, frame: SBFrame, tokens: cindex.Token):
19        self.key = key
20        self.parameters = parameters
21        self.tokens = tokens
22        self.setup(frame, **parameters)
23    # ...
```

CurrentLine-Collector Im Folgenden wird die Implementierung des `CurrentLine`-Collectors vorgestellt:

```
1 class CurrentLine(BufferedCollector):
2     def buffered_step(self, frame):
3         return frame.GetLineEntry().GetLine()
4
5     def is_reason_for_new_step(self):
6         return False
```

Dabei ist zu beachten, dass nicht die Methode `step`, sondern `buffered_step` aufgerufen wurde. Dies ergibt sich daraus, dass die Collector-Klasse von `BufferedCollector` erbt, der – wie in Abschnitt 4.3.6 erläutert – das verzögerte Verhalten eines Collectors implementiert.

BufferedCollector Der `BufferedCollector` ruft bei jedem Aufruf von `pre_step` die Methode `buffered_step` auf, speichert deren Rückgabewert in einem internen Puffer mit der Größe von genau einem Element und gibt bei der Ausführung der eigentlichen `step`-Methode stets das gepufferte Element aus dem vorherigen Schritt zurück.

Falls nach der Ausführung des letzten Debuggingschritts noch ein Element im Puffer verbleibt, wird dieses am Ende des Analyseprozesses vom Pythonprogramm über `get_buffered_step` abgerufen:

```
1 class BufferedCollector(Collector):
2     def __init__(self, key, parameters, frame: SBFrame, tokens: cindex.Token):
3         super().__init__(key, parameters, frame, tokens)
4         self.previous_step = self.get_initial_value()
5         self.current_step = None
6         self.setup(frame, **parameters)
7
8     def buffered_step(self, frame: SBFrame):
9         raise NotImplementedError("BufferedCollector hasn't implemented its
10                                → buffered_step function")
11
12     def get_initial_value(self):
13         return None
14
15     def pre_step(self, frame: SBFrame):
```

5.2 Analysetool

```
15         self.current_step = self.previous_step
16         self.previous_step = self.buffered_step(frame)
17
18     def process_step(self, frame: SBFrame):
19         return self.key, self.current_step
20
21     def get_buffered_step(self):
22         return self.key, self.previous_step
```

ArrayWatcher Im Folgenden wird die Implementierung des ArrayWatchers gezeigt. Dieser erbt direkt von der Basisklasse Collector, da keine Pufferung erforderlich ist – das zu beobachtende Ereignis ist ohnehin bereits um einen Schritt verzögert.

Zur Referenzierung des Arrays wird die Hilfsklasse `Array` verwendet (siehe frühere, verkürzte Darstellung). In jedem Schritt wird der aktuelle Zustand des Arrays mit dem zuvor gespeicherten verglichen. Bei einer Änderung wird ein neuer Analyseschritt erzeugt.

Da der interne Zustand zum Speichern des vorherigen Array-Zustands initial auf `None` gesetzt ist, wird beim ersten Aufruf der `step`-Methode direkt ein Analyseschritt generiert – `None` unterscheidet sich definitionsgemäß vom ersten tatsächlichen Zustand des Arrays:

```
1 class ArrayWatcher(Collector):
2     def setup(self, frame, *, name, size):
3         self.array = Array(frame, name, size)
4         self.previous = None
5
6     def step(self, frame: SBFrame):
7         new = self.array.get()
8         if new != self.previous:
9             self.previous = new
10            return new
11        return None
12
13    def is_reason_for_new_step(self):
14        return True
```

CompareOperationWatcher-Collector Der CompareOperationWatcher dient als Grundlage für den in der Quicksort-Aufgabenstellung verwendeten ArrayCompareOperationWatcher. Im Gegensatz zu letzterem generiert er für jede erkannte Vergleichsoperation einen Analyseschritt – unabhängig davon, ob einer der Operanden auf das überwachte Array verweist.

Während der Initialisierung analysiert der Watcher den geparssten Syntaxbaum des eingereichten Codes und durchsucht diesen rekursiv nach Vergleichsoperationen. Anschließend wird geprüft, ob es sich bei der jeweiligen Operation um eine der gesuchten Vergleichsoperationen handelt. Ist dies der Fall, werden die entsprechende Codestelle sowie der Quelltext beider Operanden extrahiert. Das Ergebnis ist eine Liste aller im Code enthaltenen Vergleichsoperationen. Der vorliegende Code wurde mit Unterstützung von ChatGPT erstellt:

```
1  class CompareOperationWatcher(BufferedCollector):
2      def extract_compare_operations(node):
3          results = []
4          if node.kind == cindex.CursorKind.BINARY_OPERATOR:
5              bin_ops = { '<', '>', '<=', '>=', '==', '!= ' }
6              raw = ''.join(t.spelling for t in node.get_tokens())
7
8              children = list(node.get_children())
9
10             if len(children) != 2:
11                 raise Exception(f'unexpected number of children for binary
12                               operator: {len(children)}')
13
14             child1 = children[0]
15             child2 = children[1]
16
17             lhs = ''.join(t.spelling for t in child1.get_tokens())
18             rhs = ''.join(t.spelling for t in child2.get_tokens())
19             op = raw[len(lhs):-len(rhs)]
20
21             if op in bin_ops:
22                 # extract range
23                 extent = node.extent
24                 start = extent.start
```

5.2 Analysetool

```
24         end = extent.end
25
26         results.append({
27             'lhs': lhs,
28             'op': op,
29             'rhs': rhs,
30             'range': {
31                 'start': { 'line': start.line, 'column': start.column
32                         ↵ },
33                 'end': { 'line': end.line, 'column': end.column }
34             },
35             'lhsRange': {
36                 'start': { 'line': child1.extent.start.line, 'column':
37                         ↵ child1.extent.start.column },
38                 'end': { 'line': child1.extent.end.line, 'column':
39                         ↵ child1.extent.end.column }
40             },
41             'rhsRange': {
42                 'start': { 'line': child2.extent.start.line, 'column':
43                         ↵ child2.extent.start.column },
44                 'end': { 'line': child2.extent.end.line, 'column':
45                         ↵ child2.extent.end.column }
46             }
47         })
48
49     for child in node.get_children():
50         results.extend(CompareOperationWatcher.extract_compare_]
51                         ↵ operations(child))
52
53     return results
54
55 def setup(self, frame: SBFrame):
56     self.compare_operations = CompareOperationWatcher.extract_compare_]
57                         ↵ operations(self.tokens.cursor)
58
59 def buffered_step(self, frame: SBFrame):
60     # ...
61
62 def is_reason_for_new_step(self):
```

```
56     # ...
57
```

Vor Ausführung eines Debuggingschritts wird überprüft, ob sich die aktuell auszuführende Codezeile mit einer Zeile deckt, in der zuvor eine Vergleichsoperation identifiziert wurde. Falls ja, werden alle Vergleichsoperationen in dieser Zeile als Analyseschritt ausgegeben:

```
1  class CompareOperationWatcher(BufferedCollector):
2      def extract_compare_operations(node):
3          # ...
4
5      def setup(self, frame: SBFrame):
6          # ...
7
8      def buffered_step(self, frame: SBFrame):
9          line = frame.GetLineEntry().GetLine()
10         result_list = []
11         for op in self.compare_operations:
12             if op['range'][ 'start'][ 'line'] <= line <=
13                 op['range'][ 'end'][ 'line']:
14                 result_list.append({
15                     'operation': op,
16                     'lhsValue': frame.EvaluateExpression(op[ 'lhs']).GetValue(),
17                     'rhsValue': frame.EvaluateExpression(op[ 'rhs']).GetValue()
18                 })
19         if result_list == []:
20             return None
21         return result_list
22
23     def is_reason_for_new_step(self):
24         # ...
```

ArrayCompareOperationWatcher-Collector Der ArrayCompareOperationWatcher erweitert den CompareOperationWatcher und erzeugt nur dann einen Analyseschritt, wenn mindestens einer der Operanden auf ein bestimmtes Array verweist.

5.2 Analysetool

Zur Überprüfung dieses Zustands wird bei jeder erkannten Vergleichsoperation der Ausdruck beider Operanden zur Laufzeit mit einem nachträglich hinzugefügten Referenzierungsoperator ausgewertet. Ergibt die Evaluation eine Adresse, die in den Adressbereich des angegebenen Arrays fällt, wird die betreffende Vergleichsoperation als Analyseschritt ausgegeben.

```
1  class ArrayCompareOperationWatcher(CompareOperationWatcher):
2      def setup(self, frame: SBFrame, *, name, size):
3          super().setup(frame)
4          self.array = Array(frame, name, size)
5
6      def buffered_step(self, frame: SBFrame):
7          line_entry = frame.GetLineEntry()
8          line = line_entry.GetLine()
9          result_list = []
10         for op in self.compare_operations:
11             if op['range']['start']['line'] <= line <=
12                 op['range']['end']['line']:
13                 lhs = frame.EvaluateExpression('&(' + op['lhs'] + ')')
14                 rhs = frame.EvaluateExpression('&(' + op['rhs'] + ')')
15                 lhs_index = self.array.get_referenced_index(lhs)
16                 rhs_index = self.array.get_referenced_index(rhs)
17                 if lhs_index is not None or rhs_index is not None:
18                     lhs_value = frame.EvaluateExpression(op['lhs']).GetValue()
19                     rhs_value = frame.EvaluateExpression(op['rhs']).GetValue()
20                     result_list.append({
21                         'operation': op,
22                         'lhs': {
23                             'index': lhs_index,
24                             'value': lhs_value
25                         },
26                         'rhs': {
27                             'index': rhs_index,
28                             'value': rhs_value
29                         }
30                     })
31         if result_list == []:
32             return None
33         return result_list
```

5.2.10 Postprocessoren

SkipInterSwappingSteps Im Folgenden wird die Implementierung der Postprocessor-Klasse SkipInterSwappingSteps vorgestellt. Neben den Methoden `process` und `setup`, die von jeder Postprocessor-Implementierung bereitgestellt werden müssen, enthält dieser Postprocessor die Hilfsfunktionen `handle_step` und `find_changed_index`, die in den nachfolgenden Codeausschnitten näher erläutert werden:

```
1  class SkipInterSwappingSteps(Postprocessor):
2      def setup(self, array):
3          self.skip_next = False
4          self.skip_information = None
5          self.array_key = array
6
7      def process(self, collected_list):
8          # ...
9
10     def handle_step(self, current_step, index, collected_list):
11         # ...
12
13     def find_changed_index(self, list1, list2, startIndex=0):
14         # ...
```

Im Folgenden wird die `process`-Methode näher erläutert. Da der Postprocessor ausschließlich Analyseschritte verarbeiten soll, die in Zusammenhang mit dem in der Konfiguration benannten ArrayWatcher stehen, wird zunächst die Hilfsliste `filtered_with_indices` erstellt. Diese enthält nur solche Analyseschritte, bei denen ein Bezug zum entsprechenden ArrayWatcher besteht. Gleichzeitig wird eine Maskierungsliste erstellt, in der Einträge mit dem Wert `False` kennzeichnen, dass der zugehörige Analyseschritt entfernt werden soll.

Die Methode `handle_step` verarbeitet die relevanten Analyseschritte und gibt `False` zurück, wenn ein Schritt gelöscht werden soll, andernfalls `True`. In der Verarbeitungsschleife wird die Maskierungsliste anhand der ursprünglichen Indizes der Gesamtmenge erstellt, während `handle_step` auf den gefilterten Teilbereich der Schritte angewendet wird:

5.2 Analysetool

```
1  class SkipInterSwappingSteps(Postprocessor):
2      def process(self, collected_list):
3          # deep copy the list to not mutate the given list
4          lst = [{**copy.deepcopy(step), self.key: None} for step in
5                  ↳ collected_list]
6
7          # Get a list of analysis steps that include an ArrayWatcher result
8          filtered_with_indices = [(i, x) for i, x in enumerate(lst)
9                                  if x.get(self.array_key, None) is not None]
10
11         # Process all analysis steps as they were all containing the
12         # → ArrayWatcher result but keep items that don't have it inbetween.
13         mask = [True] * len(lst)
14         for filtered_index, (actual_index, step) in
15             enumerate(filtered_with_indices):
16             mask[actual_index] = self.handle_step(step, filtered_index,
17                 ↳ filtered_with_indices)
18
19         return [item for keep, item in zip(mask, lst) if keep]
20     # ...
```

Die Methode `handle_step` vergleicht jeweils drei aufeinanderfolgende Analyseschritte mit Bezug zum ArrayWatcher und prüft, ob die in Abschnitt 4.3.7.1 beschriebene Bedingung zutrifft. Falls ein Schritt entfernt werden soll, wird dessen Inhalt in der Variable `self.skip_information` zwischengespeichert.

Im darauffolgenden Schritt wird dann der jeweils nächste Analyseschritt manipuliert, indem unter dem vorkonfigurierten Schlüssel die zwischengespeicherte Information aus `self.skip_information` ergänzt wird:

```
1  class SkipInterSwappingSteps(Postprocessor):
2      def handle_step(self, current_step, index, collected_list):
3          if index == 0 or index == len(collected_list) - 1:
4              current_step[self.key] = self.skip_information
5              return True
6
7          first = collected_list[index - 1][self.array_key]
8          second = collected_list[index - 0][self.array_key]
9          third = collected_list[index + 1][self.array_key]
```

```
10     change_at_second = self.find_changed_index(first, second)
11     change_at_third = self.find_changed_index(second, third)
12
13     if change_at_second != change_at_third and second[change_at_second] ==
14         ↵ second[change_at_third] and first[change_at_second] ==
15         ↵ third[change_at_third]:
16         self.skip_information = current_step
17         return False
18
19
20     current_step[self.key] = self.skip_information
21     self.skip_information = None
22
23     return True
24
25 # ...
```

Die Hilfsmethode `find_changed_index`, welche zur Ermittlung veränderter Arraypositionen dient, ist wie folgt definiert:

```
1 class SkipInterSwappingSteps(Postprocessor):
2     def find_changed_index(self, list1, list2, startIndex=0):
3         if len(list1) != len(list2):
4             raise ValueError("The two lists must have the same length.")
5         for i in range(startIndex, len(list1)):
6             if list1[i] != list2[i]:
7                 return i
8         return None
9
10 # ...
```

KeepTrackOfItems Die Postprocessor-Klasse `KeepTrackOfItems` dient dazu, aus der Veränderung zwischen zwei aufeinanderfolgenden Analyseschritten, die einen Schritt des konfigurierten `ArrayWatcher` enthalten, zu ermitteln, ob es sich dabei um eine Ersetzung oder eine Vertauschung handelt. Zur Vereinfachung des Vorgehens wurden, analog zu `SkipInterSwappingStep`, die Hilfsmethoden `handle_step` und `find_changed_index` definiert:

```
1 class KeepTrackOfItems(Postprocessor):
2     def setup(self, array):
```

5.2 Analysetool

```
3         self.array_key = array
4
5     def process(self, collected_step):
6         # ...
7
8     def handle_step(self, current_step, index, collected_list):
9         # ...
10
11    def find_changed_index(self, list1, list2, startIndex=0):
12        # ...
```

Wie bereits in der Postprocessor-Klasse SkipInterSwappingStep wird auch hier eine gefilterte Hilfsliste `filtered` erstellt, da nur solche Analyseschritte betrachtet werden, die in Bezug zum konfigurierten ArrayWatcher stehen. Im Gegensatz zu SkipInterSwappingStep ist es hier jedoch nicht erforderlich, Schritte zu entfernen. Entsprechend wird keine Maskierungsliste benötigt, und die Verknüpfung der gefilterten Liste mit den Originalindizes entfällt. Dadurch ist die Implementierung der `process`-Methode konzeptionell ähnlich, jedoch insgesamt weniger komplex:

```
1 class KeepTrackOfItems(Postprocessor):
2     def process(self, collected_step):
3         lst = [{**copy.deepcopy(step), self.key: None} for step in
4                ↳ collected_step]
4         filtered = list(filter(lambda x: self.array_key in x and
5                ↳ x[self.array_key] is not None, lst))
5         for i, current_step in enumerate(filtered):
6             self.handle_step(current_step, i, filtered)
7         return lst
8     # ...
```

Die Hilfsfunktion `handle_step` prüft, ob zwischen zwei Schritten ein oder zwei Indizes verändert wurden. Ist genau ein Index betroffen, handelt es sich um eine Ersatzung. Sind zwei Indizes betroffen und die Werte wurden tatsächlich gegeneinander ausgetauscht, liegt eine Vertauschung vor:

```
1 class KeepTrackOfItems(Postprocessor):
2     def handle_step(self, current_step, index, collected_list):
3         if index < 1:
```

```
4         return
5
6     previous_list = collected_list[index - 1][self.array_key]
7     current_list = current_step[self.array_key]
8
9     first_change = self.find_changed_index(previous_list, current_list)
10    second_change = self.find_changed_index(previous_list, current_list,
11        ↪   first_change + 1)
12
13    if second_change is None:
14        # no swap behaviour, only one item changed, so we assume a new item
15        ↪   got added
16        current_step[self.key] = {
17            'type': 'replace',
18            'index': first_change,
19            'oldValue': current_list[first_change],
20            'newValue': current_list[first_change]
21        }
22
23    # potential swap behaviour
24    if previous_list[first_change] == current_list[second_change] and
25        ↪   previous_list[second_change] == current_list[first_change]:
26        # it's indeed a swap
27        current_step[self.key] = {
28            'type': 'swap',
29            'index1': first_change,
30            'index2': second_change
31        }
32
33    return
34
35    # ...
```

5.2.11 Testing

Soll das Analysetool den übergebenen Code anhand definierter Testfälle überprüfen, wird für Sortieralgorithmen die entsprechende `int main()`-Funktion aus dem Preset dafür verwendet. Diese erwartet die Testfälle über die Standardeingabe `stdin` und verar-

5.2 Analysetool

beitet sie wie folgt: Zunächst wird die Anzahl der auszuführenden Testfälle übergeben. Für jeden Testfall folgt die Anzahl der Array-Elemente, gefolgt von den jeweiligen Elementen als Ganzzahlen.

Die Ausgabe des Programms erfolgt ebenfalls über `stdout` in einem entsprechenden Format für jeden ausgeführten Testcase: Zunächst die Anzahl der Elemente des sortierten Arrays, gefolgt von den sortierten Werten.

```
1 #include <iostream>
2 void sort(int*, int);
3
4 int main() {
5     int testCases;
6     std::cin >> testCases;
7     while (testCases--) {
8         int n;
9         std::cin >> n;
10        int* arr = new int[n];
11        for (int i = 0; i < n; i++) {
12            std::cin >> arr[i];
13        }
14        sort(arr, n);
15        std::cout << n << " ";
16        for (int i = 0; i < n; i++) {
17            std::cout << arr[i] << " ";
18        }
19        delete[] arr;
20    }
21 }
```

Das Analysetool übergibt die Eingabewerte programmatisch an die zu testende Executable, wie im folgenden Beispiel dargestellt. Zunächst wird der gesamte Inhalt für `stdin` generiert, indem alle relevanten Eingabewerte entsprechend der beschriebenen Struktur durch Leerzeichen getrennt zusammengesetzt werden. Anschließend wird die Executable mit diesem Eingabewert gestartet, wobei die Ausgaben in der Variable `p` gespeichert werden.

Da die Ausgabe eine flache Folge von Ganzzahlen ist, wird diese mithilfe der Hilfsfunktion `chunks` in separate Ergebnisse pro Testfall aufgeteilt. Diese werden schließlich mit den erwarteten Ausgaben verglichen, um die Resultate der Testfälle zu ermitteln.

```
1  stdin = str(len(test_cases)) + ' ' + ' '.join(map(test_case_to_stdin,
2      ↵  test_cases))
2  p = subprocess.run([executable_filename], input=stdin, text=True,
3      ↵  capture_output=True)
3
4  stdout = list(map(int, p.stdout.strip().split(' ')))
5  output_values = chunks(stdout)
6
7  testcase_result = [
8      'testcase': testcase,
9      'passed': output == test_case['expected']
10 } for output, test_case in zip(output_values, test_cases)]
11
12 # ...
13
14 def test_case_to_stdin(test_case):
15     return str(len(test_case['input'])) + ' ' + ' '.join(map(str,
16         ↵  test_case['input']))
17
17 def chunks(lst):
18     if not lst:
19         return
20     n = lst[0]
21     yield lst[1:n+1]
22     yield from chunks(lst[n+1:])
```

5.2.12 Dockerfile

Im zugehörigen Dockerfile werden die benötigten Komponenten *Clang*, *LLDB* und *Python* installiert.

Zunächst wird ein externes APT-Repository für die LLVM-Toolchain eingebunden, indem der entsprechende GPG-Schlüssel zur Vertrauensstellung importiert und das Repo-

5.2 Analysetool

sitory hinzugefügt wird. Anschließend wird die Paketliste aktualisiert. Daraufhin werden die relevanten Pakete clang, lldb und python3 installiert, einschließlich der *Clang-Python-Bindings*.

```
1 ENV llvmVersion=19
2 RUN apt update -yq
3 RUN apt install -yq lsb-release wget software-properties-common apt-utils
4 RUN wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -
5 RUN add-apt-repository -y "deb http://apt.llvm.org/focal/
   ↳ llvm-toolchain-focal-${llvmVersion} main"
6 RUN apt update -yq
7 RUN apt install -yq clang-${llvmVersion} lldb-${llvmVersion} lld-${llvmVersion}
   ↳ python3 python3-clang-${llvmVersion} python3-pip
8 RUN pip install clang
```

Anschließend werden die Umgebungsvariablen LLDB_DEBUGSERVER_PATH und PYTHONPATH gesetzt, um die korrekte Ausführung und Integration von LLDB und Python-Modulen sicherzustellen.

```
1 ENV LLDB_DEBUGSERVER_PATH=/usr/bin/lldb-server-19
2 ENV PYTHONPATH=/usr/lib/python3/dist-packages:/app
```

Ein dedizierter User mit eingeschränkten Rechten wird eingerichtet, um sicherheitskritische Operationen im Container zu vermeiden. Die Anwendung wird im Verzeichnis /app ausgeführt, welches dem neuen User zugewiesen wird.

```
1 RUN groupadd -r myuser && useradd -r -g myuser myuser
2 RUN chown -R myuser:myuser /app
3 USER myuser
```

Schließlich wird das CLI-Tool durch Angabe des Python-Skripts run.py als Einstiegspunkt definiert:

```
1 ENTRYPOINT ["python3", "/app/run.py"]
```

5.3 Executor

5.3.1 Bereitstellung des HTTP-Endpoints mit Streamingverhalten

Im Folgenden wird gezeigt, wie der Executor seinen HTTP-Endpunkt bereitstellt.

Im oberen Abschnitt des Codes ist zu erkennen, wie der Executor die Anzahl aktiver Verbindungen überwacht und sicherstellt, dass nicht mehr als fünf Analyseprozesse gleichzeitig ausgeführt werden. Dabei ist zu beachten, dass im hier gezeigten Ausschnitt auf Fehlerbehandlung verzichtet wurde. Diese ist jedoch in der tatsächlichen Implementierung zwingend erforderlich, um sicherzustellen, dass die Variable `connectionCount` auch im Fehlerfall korrekt dekrementiert wird.

Für das Streamingverhalten der Antwort wird der HTTP-Header `Transfer-Encoding: chunked` gesetzt. Die Funktion `sendJson` transformiert ein übergebenes Objekt in eine einzeilige JSON-Darstellung und hängt einen Zeilenumbruch `\n` an, der – wie in Unterkapitel 4.4 vereinbart – als Trennzeichen zwischen Nachrichten dient.

Die Funktion `runBuild` startet den Analyseprozess und liefert das finale Ergebnis zurück, sobald die Ausführung abgeschlossen ist. Statusmeldungen, die während des Prozesses erzeugt werden, werden über eine Callback-Funktion verarbeitet und unmittelbar an die Webanwendung weitergeleitet.

```
1 let connectionCount = 0
2 const maxConnectionCount = 5
3
4 app.post('/build', async (req, res) => {
5     if (connectionCount >= maxConnectionCount) {
6         /* send back an error */
7     }
8     connectionCount += 1
9
10    /* ... */
11
12    res.setHeader('Content-Type', 'application/json')
```

5.3 Executor

```
13     res.setHeader('Transfer-Encoding', 'chunked')
14     res.flushHeaders()
15
16     const result = await runBuild(type, presetName, code, message => {
17         sendJson(res, {
18             type: 'status',
19             message
20         })
21     })
22
23     sendJson(res, result)
24
25     connectionCount -= 1
26 }
27
28 function sendJson(res, data) {
29     res.write(JSON.stringify(data) + '\n')
30 }
```

5.3.2 Starten des Analysetools

Als Nächstes wird gezeigt, wie die Funktion `runBuild` implementiert ist.

Da im eingehenden Request lediglich der Identifier des gewünschten Presets übergeben wird, wird das zugehörige Preset zunächst mithilfe der Funktion `fetchPreset` von der Preset-Datenbank geladen. Anschließend wird das Analysetool in einem Docker Container gestartet.

Da das Analysetool mithilfe von *Docker Compose* gebaut wird (vgl. Abschnitt 5.6), ist der konkrete Name des verwendeten Docker-Images abhängig von der jeweiligen Ausführungsumgebung. Aus diesem Grund wird der Image-Name in einer `.env`-Datei definiert und wurde zur Laufzeit in die Variable `dockerImage` geladen.

Zur Erhöhung der Sicherheit werden beim Starten des Containers zusätzliche Docker-Flags gesetzt, die bestimmte Sicherheitsrichtlinien durchsetzen. Detaillierte Informationen hierzu finden sich in Abschnitt 5.3.3.

Der Parameter type dient als Diskriminator und legt fest, ob es sich bei dem Auftrag um eine Analyse oder um einen Test handelt.

Für die Auswertung der Konsolenausgabe wird das npm-Package *split2* verwendet. Durch die Weiterleitung der Standardausgabe via `child.stdout.pipe(split2())` wird die Methode `.on('data', line => ...)` für jede einzelne Zeile aufgerufen. Dies ist insofern hilfreich, als die vom Analysetool ausgegebenen JSON-Nachrichten jeweils einzeilig sind und durch Zeilenumbrüche voneinander getrennt werden.

Eingehende Statusmeldungen werden an die übergebene Callback-Funktion weitergeleitet. Alle übrigen Nachrichten werden als finales Ergebnis oder als Fehler klassifiziert und direkt zurückgegeben:

```
1 import split2 from 'split2'
2
3 async function runBuild(type, presetName, code, onStatusUpdate) {
4     const preset = await fetchPreset(presetName)
5
6     return new Promise((resolve, reject) => {
7         const child = spawn(
8             'docker', ['run', '--rm', '-i', ...containerRestrictions,
9             ↳ dockerImage, type, JSON.stringify(preset),
10            ↳ JSON.stringify(code)],
11            { cwd: path.join(process.cwd()) }
12        );
13
14         child.stdout
15             .pipe(split2())
16             .on('data', line => {
17                 if (!line.trim()) {
18                     return
19                 }
20                 const data = JSON.parse(line)
21                 if (data.type === 'status') {
22                     onStatusUpdate?.(data.message)
23                 } else {
24                     resolve(data)
25                 }
26             })
27     });
28 }
```

5.3 Executor

```
24         }
25     })
26   })
27 }
```

5.3.3 Containerrestriktionen

In der folgenden Konfiguration sind die Docker-Flags aufgelistet, die verwendet werden, um den Container des Analysetools sicherer auszuführen. Unter anderem wird der Netzwerkzugang deaktiviert, um eine Kommunikation mit externen Systemen zu unterbinden. Zusätzlich wird ein Arbeitsspeicherlimit von maximal 64 Megabyte festgelegt, die Anzahl gleichzeitig laufender Prozesse begrenzt sowie die CPU-Auslastung auf die Kapazität eines einzelnen Kerns limitiert:

```
1
2 const containerRestrictions = [
3   '--network=none', // No network access
4   '--cap-drop=ALL', // Drop all capabilities
5   '--security-opt=no-new-privileges', // No new privileges
6   '-m=64m', // Memory limit of 64MB
7   '--cpus=1', // Limit to 100% of a CPU
8   '--read-only', // Read-only filesystem
9   '--tmpfs=/tmp:rw,exec', // Temporary filesystem for /tmp
10  '--ulimit', 'nproc=40:40', // Limit processes to 40 to prevent fork bombs
11  '--ulimit', 'nofile=40:40', // Limit open files to 40
12 ]
```

5.3.4 Anfragelogik der Webanwendung

Im Folgenden wird gezeigt, wie die Webanwendung eine Anfrage an den Executor stellt und die Antwort als Stream verarbeitet.

Dazu wird die Hilfsfunktion `readChunks` verwendet, die einen `AsyncGenerator` zurückgibt. Dadurch ist es möglich, mithilfe einer `for await`-Schleife nacheinander auf alle eingehenden Nachrichten zu warten und diese jeweils einzeln zu verarbeiten.

Die Funktion `readChunks` liest die HTTP-Response als Stream ein und reagiert auf neu eintreffende Daten. Sobald Daten empfangen werden, kombiniert sie diese mit eventuell bereits teilweise empfangenen Zeilen und extrahiert daraus vollständige Zeilen. Diese werden anschließend in JSON geparsst und jeweils als nächster Schritt des Iterators weitergegeben.

```

1  const response = await fetch(endpoint, {
2      method: 'POST',
3      headers: {
4          'Content-Type': 'application/json',
5      },
6      body: JSON.stringify(payload),
7  })
8
9  const reader = response.body.getReader();
10
11 for await (const data of readChunks(reader)) {
12     /* `data` is one json response of type status, result, etc. */
13 }
14
15 /* ... */
16 async function* readChunks(reader) {
17     const decoder = new TextDecoder('utf-8')
18     let buffer = ''
19     while (true) {
20         const { done, value } = await reader.read()
21         if (done) {
22             break
23         }
24         buffer += decoder.decode(value, { stream: true })
25
26         let lines = buffer.split('\n')
27         buffer = lines.pop() || '' // Keep the last incomplete line in the
28         ↪ buffer
29         for (const line of lines) {
30             yield JSON.parse(line)
31         }
32     }
33     if (buffer.trim()) {

```

5.3 Executor

```
33         yield JSON.parse(buffer) // Process any remaining data in the buffer
34     }
35 }
```

5.3.5 Dockerfile

Im Folgenden wird das Dockerfile vorgestellt, das eine typische *Node.js*-Umgebung bereitstellt, alle benötigten npm-Packages installiert und anschließend den Server startet.

Eine Besonderheit dieser Konfiguration besteht darin, dass im Docker-Image zusätzlich *Docker* selbst installiert wird, um ein sogenanntes „Docker-in-Docker“-Verhalten zu ermöglichen. Dies ist notwendig, damit der Executor innerhalb seines eigenen Containers weitere Docker Container starten kann, insbesondere für das Ausführen des Analysetools.

```
1 FROM node:22-bookworm
2
3 # add docker capabilities
4 RUN wget -O- get.docker.com | sh
5
6 WORKDIR /app
7 COPY package*.json ./
8 RUN npm install
9 COPY . .
10
11 EXPOSE 80
12
13 CMD [ "node", "index.js" ]
```

5.4 Reverse Proxy

5.4.1 Dockerfile

Das Dockerfile basiert auf einem offiziellen *NGINX*-Image und lädt die Konfiguration, die in Abschnitt 5.4.2 beschrieben ist:

```
1 FROM nginx:latest
2
3 COPY nginx.conf /etc/nginx/nginx.conf
```

5.4.2 Konfiguration von NGINX

Die Konfiguration von *NGINX* ist so ausgelegt, dass Anfragen mit dem Pfadpräfix `/api` an den Executor weitergeleitet werden, während alle übrigen Anfragen an den Webserver gerichtet sind.

Darüber hinaus wird der interne Puffermechanismus für die Kommunikation mit dem Executor deaktiviert, um eine verzögerungsfreie Datenübertragung im Sinne einer Echtzeitverbindung zu ermöglichen.

Für Verbindungen zum Webserver sind zusätzlich WebSocket-Verbindungen erlaubt, sodass das Hot-Restart-Feature des Entwicklungsservers unterstützt wird.

```
1 # Reverse proxy for
2 # - web server: from frontend:80 -> /
3 # - task runner: from executor:80 -> /api
4 events {}
5
6 http {
7     server {
8         listen 80;
9         server_name localhost;
10
11         # Reverse proxy for task runner
12         location /api/ {
```

5.5 Preset-Datenbank

```
13         proxy_pass http://executor:80/;
14         proxy_set_header Host $host;
15         proxy_set_header X-Real-IP $remote_addr;
16         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
17         proxy_set_header X-Forwarded-Proto $scheme;
18
19         # disable buffering
20         proxy_buffering off;
21     }
22
23     # Reverse proxy for web server
24     location / {
25         proxy_pass http://frontend:80/;
26         proxy_set_header Host $host;
27         proxy_set_header X-Real-IP $remote_addr;
28         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
29         proxy_set_header X-Forwarded-Proto $scheme;
30
31         # websocket support, needed for Next.js dev server
32         # https://stackoverflow.com/a/14969925/6368046
33         proxy_http_version 1.1;
34         proxy_set_header Upgrade $http_upgrade;
35         proxy_set_header Connection "upgrade";
36         proxy_read_timeout 86400;
37     }
38 }
39 }
```

5.5 Preset-Datenbank

5.5.1 Auslieferung eines Presets

Das Preset wird entsprechend der in Abschnitt 4.6 dargestellten Dateistruktur in den Container der Preset-Datenbank eingebunden.

Ein HTTP-Endpoint der Preset-Datenbank ist dafür zuständig, die zugehörigen Dateien vollständig einzulesen und sie in einer gebündelten JSON-Antwort bereitzustellen.

Zur Ermittlung der entsprechenden Inhalte kommt die Funktion `getConfig` zum Einsatz:

```
1  export async function getConfig(presetName) {
2      const path = `./config/${presetName}`
3      const files = [
4          'analysis/analysis.json',
5          'analysis/main.cpp',
6          'test/main.cpp',
7          'test/testcases.json',
8          'template.cpp'
9      ]
10     const readFiles = await Promise.all(
11         files.map(file => readFile(`${path}/${file}`, 'utf-8')).catch(() =>
12             null)
13     )
14     if (readFiles.some(content => content === null)) {
15         return null
16     }
17     const fileContents = Object.fromEntries(
18         readFiles.map((content, index) => [files[index], content])
19     )
20
21     return {
22         presetName,
23         template: fileContents['template.cpp'],
24         analysis: {
25             config: JSON.parse(fileContents['analysis/analysis.json']),
26             main: fileContents['analysis/main.cpp'],
27         },
28         test: {
29             main: fileContents['test/main.cpp'],
30             testcases: JSON.parse(fileContents['test/testcases.json']),
31         }
32     }
33 }
```

5.5.2 Auslieferung eines Templates

Außerdem stellt ein weiterer Endpoint das Template in vorverarbeiteter Form zur Verfügung. Hierzu wird die Template-Datei eingelesen und zeilenweise analysiert. In Zeile 16 wird jede Zeile mit regulären Ausdrücken geprüft. Trifft einer der Ausdrücke zu, so ergibt sich daraus ein Schlüssel-Wert-Paar, das zur Fallunterscheidung verwendet wird: Es wird geprüft, ob es sich um den Beginn oder das Ende eines Bereichs handelt, der von Anwender:innen bearbeitet werden darf. Wird eine Startmarkierung erkannt, so wird eine neue Variable `userCodeRange` mit den Startpositionen initialisiert. Sobald eine Endmarkierung erkannt wird, wird der Bereich vervollständigt und in die Ergebnisliste übernommen:

```
1 const markers = {
2   userCodeStart: /^\\s*/\\/\s*<user-code-start>\\s$/,
3   userCodeEnd: /^\\s*/\\/\s*<user-code-end>\\s$/,
4 }
5
6 export async function getTemplate(presetName) {
7   const templatePath = `./config/${presetName}/template.cpp`
8   const data = await readFile(templatePath, 'utf-8')
9
10  const lines = data.split('\\n')
11  let lineNumber = 0
12  let result = ''
13  const ranges = []
14  let userCodeRange = null
15  for (const line of lines) {
16    const match = Object.entries(markers).find(([marker, regex]) =>
17      line.match(regex))
18    if (match) {
19      const marker = match[0]
20      if (marker === 'userCodeStart') {
21        result += '\\n'
22        lineNumber += 1
23        userCodeRange = {
24          start: { line: lineNumber, column: 1, index: result.length
25            },
26        }
27      }
28    }
29  }
30  return result + userCodeRange?.start?.index ? '' : '\\n' + result
31}
```

```
24         end: null,
25         initialCode: ''
26     }
27 }
28 else if (marker === 'userCodeEnd') {
29     if (userCodeRange) {
30         userCodeRange.end = { line: lineNumber, column: 1, index:
31             ↪ result.length }
32         userCodeRange.initialCode =
33             ↪ userCodeRange.initialCode.slice(0, -1) // remove the
34             ↪ last \n
35         ranges.push(userCodeRange)
36         userCodeRange = null
37     }
38 }
39 else if (!userCodeRange) {
40     lineNumber += 1
41     result += line + '\n'
42 }
43 else {
44     userCodeRange.initialCode += line + '\n'
45 }
46 return {
47     presetName,
48     template: result,
49     ranges
50 }
```

5.5.3 Dockerfile

Das Dockerfile der Preset-Datenbank basiert auf einer typischen *Node.js*-Umgebung. Darin werden zunächst alle benötigten *npm*-Pakete installiert, bevor anschließend der Server über den definierten Einstiegspunkt gestartet wird:

5.6 Docker Compose File

```
1 FROM node:22-bookworm
2
3 WORKDIR /app
4 COPY package*.json ./
5 RUN npm install
6 COPY . .
7
8 RUN groupadd -r myuser && useradd -r -g myuser myuser
9 RUN chown -R myuser:myuser /app
10 USER myuser
11
12 EXPOSE 80
13
14 CMD ["node", "db.js"]
```

5.6 Docker Compose File

5.6.1 Für Produktivumgebungen

Im Folgenden wird das docker-compose-File vorgestellt, das für den Einsatz in Produktivumgebungen vorgesehen ist. Dabei werden die jeweils zugehörigen Dockerfiles für den Bau der Images verwendet. Der Port für den Reverse Proxy wird über eine .env-Datei festgelegt.

Da der Executor zur Ausführung des Analyseprozesses selbstständig Docker Container starten muss, wird in Zeile 16 das erforderliche Volume eingebunden, um Zugriff auf den Docker-Daemon zu erhalten.

Damit das Analysetool nach dem Hochfahren des Systems sofort vom Executor gestartet werden kann, wird es direkt beim Systemstart einmalig gebaut. Zu diesem Zweck wird der ENTRYPOINT so gesetzt, dass er sofort wieder terminiert.

```
1 services:
2   preset-db:
3     build: ./preset-db
```

```
4      restart: unless-stopped
5
6  reverse-proxy:
7      build: ./reverse-proxy
8      restart: unless-stopped
9      ports:
10         - "$PORT:80"
11
12  executor:
13      build: ./executor
14      restart: unless-stopped
15      volumes:
16         - /var/run/docker.sock:/var/run/docker.sock
17      env_file:
18         - .env
19
20  # we just need it to be built, not run
21  # it will error because there's no command
22  analysis-tool:
23      build: ./analysis-tool
24      entrypoint: /bin/bash -c "echo it's ok for me to be stopped :)" # we want
25          ↵ it to be built, but we don't want it to be run
26
27  frontend:
28      build:
29          context: ./frontend
30          dockerfile: Dockerfile.prod
31      restart: unless-stopped
```

5.6.2 Für Entwicklungsumgebungen

Die Konfiguration für die Entwicklungsumgebung fällt umfangreicher aus. Hier kommen insbesondere die `develop`-Sektionen zum Einsatz, um den Entwicklungsworkflow zu unterstützen.

In diesen Abschnitten wird festgelegt, welche Projektdateien beobachtet werden sollen, sodass Änderungen automatisch in die jeweiligen Docker Container synchronisiert wer-

5.6 Docker Compose File

den. Je nach Einstellung kann dies auch einen Neustart oder sogar einen kompletten Neubau des betroffenen Services auslösen:

```
1 services:
2   preset-db:
3     build: ./preset-db
4     restart: no
5     develop:
6       watch:
7         - action: sync+restart
8           path: ./preset-db
9           target: /app
10          - action: rebuild
11            path: ./preset-db/package.json
12
13   reverse-proxy:
14     build: ./reverse-proxy
15     restart: no
16     ports:
17       - "$PORT:80"
18     develop:
19       watch:
20         - action: sync+restart
21           path: ./nginx.conf
22           target: /etc/nginx/nginx.conf
23
24   executor:
25     build: ./executor
26     restart: no
27     volumes:
28       - /var/run/docker.sock:/var/run/docker.sock
29     env_file:
30       - .env
31     develop:
32       watch:
33         - action: sync+restart
34           path: ./executor
35           target: /app
36         - action: rebuild
```

```
37      path: ./executor/package.json
38
39      # we just need it to be built, not run
40      # it will error because there's no command
41      analysis-tool:
42          build: ./analysis-tool
43          entrypoint: /bin/bash -c "while true; do sleep 1000; done" # we want it to
44          ↪ be built, every time we change the code, but we don't want it to run
45      develop:
46          watch:
47              - action: rebuild
48                  path: ./analysis-tool
49
50      frontend:
51          build:
52              context: ./frontend
53              dockerfile: Dockerfile.dev
54          develop:
55              watch:
56                  - action: sync
57                      path: ./frontend
58                      target: /app
59                  - action: rebuild
60                      path: ./frontend/package.json
61          restart: no
```

Voraussetzung für die Nutzung der Konfiguration der `develop`-Sektion ist, dass die Container mit dem `--watch`-Flag gestartet werden. Dieses Flag aktiviert die Beobachtung der im `docker-compose`-File angegebenen Projektverzeichnisse, sodass Änderungen an den Quelltexten automatisch erkannt und synchronisiert werden können:

```
1 docker compose up --build --watch
```

6 Fazit und Ausblick

6.1 Fazit

Sämtliche in Kapitel 2 beschriebenen Ziele und Anforderungen wurden vollständig umgesetzt. Die Webanwendung bietet eine Auswahl unterschiedlicher Programmieraufgaben und stellt zu jeder Aufgabe die Ansichten „Instructions“, „Visualization“ und „Tests“ sowie einen integrierten Codeeditor bereit. Über den Executor im Backend wird der von Anwender:innen eingegebene Code mithilfe des Analysetools ausgeführt, analysiert und in der Visualisierungsansicht wie gefordert dargestellt. Die Umsetzung umfasst sowohl die Animation von Tausch- und Ersetzungsschritten im Bubblesort-Algorithmus als auch die Visualisierung der rekursiven Natur des Quicksorts, einschließlich der automatischen Erkennung relevanter Vergleichsoperationen. Der Editor ist ausschließlich in jenen Bereichen editierbar, die in der Aufgabenstellung vorgesehen sind. Darüber hinaus bietet das System die Möglichkeit, den eingegebenen Code anhand vordefinierter Testfälle auszuführen und zu validieren.

Eine Registrierung ist wie gefordert nicht erforderlich. Der eingegebene Code wird lokal im Browser des Nutzers gespeichert. Die Zuordnung zum Analyseergebnis erfolgt über die bestehende HTTP-Verbindung, sodass keine personenbezogene Identifikation notwendig ist.

Die Architektur erlaubt zudem eine einfache Erweiterung: Neue Collector-Klassen können als einzelne Dateien ergänzt und über einen Eintrag in der Registrierungsfunktion verfügbar gemacht werden, ohne dass dadurch die Komplexität des Analysetools zunimmt. Lediglich die Webanwendung erfordert bei der Einführung eines neuen Algorithmus eine eigenständige Visualisierungskomponente, da hier bislang nur begrenzt modulare Strukturen vorliegen.

Über die ursprünglichen Anforderungen hinaus wurden zentrale Sicherheitsmaßnahmen implementiert. Hierzu zählt eine Validierung des eingereichten Codes auf potenziell schädliches Verhalten sowie die Ausführung in einer streng isolierten Containerumgebung. Die modulare Backend-Architektur mit Reverse Proxy als Gateway und getrenntem Executor ermöglicht ferner die horizontale Skalierung, beispielsweise durch den Einsatz eines Load-Balancers.

6.2 Ausblick

Für eine zukünftige Weiterentwicklung bietet es sich an, das Backend um eine Datenbank zu erweitern, die für die Verwaltung und Speicherung von Kursen, Kategorien und Aufgabenstellungen zuständig ist. Diese Informationen sind bislang noch fest im Webserver verankert und könnten durch eine Datenbank dynamischer und flexibler gepflegt werden.

Ebenso wäre die Einführung einer optionalen Registrierung für Anwender:innen sinnvoll. Damit könnten eingegebene Lösungen plattformübergreifend gespeichert und wiederhergestellt sowie Bearbeitungsfortschritte in der Datenbank erfasst werden. Auf diese Weise könnten die aktuell lediglich simulierten Fortschrittsbalken und Badges, die den Lernfortschritt visualisieren sollen, mit tatsächlichen Nutzungsdaten hinterlegt werden.

Darüber hinaus ließe sich das Testsystem weiterentwickeln: Anstelle eines reinen Vergleichs von Ein- und Ausgaben könnten die Testfälle auch das Verhalten des Sortieralgorithmus analysieren. Dies würde es ermöglichen, beispielsweise zwischen Bubblesort und Insertionsort zu unterscheiden, auch wenn beide die korrekte Endausgabe liefern. Eine solche Analyse ist durch das bestehende Analysetool bereits prinzipiell möglich. Aufbauend auf diesem Konzept könnte die Visualisierungsansicht künftig auch sofort Rückmeldung geben, sobald das tatsächliche Verhalten des Algorithmus von der erwarteten Lösungsstrategie abweicht.

Da derzeit der größte Aufwand beim Hinzufügen neuer Problemkategorien oder Aufgabenstellungen in der Umsetzung der jeweiligen Visualisierung in der Webanwendung

6.2 Ausblick

liegt, sollte langfristig eine modulare Herangehensweise anvisiert werden. Im Idealfall ließe sich die Visualisierung durch ein webbasiertes Konfigurationstool teilweise oder sogar vollständig erstellen.

Auch für das Analysetool selbst ist eine sinnvolle Erweiterung denkbar: So könnte die Möglichkeit geschaffen werden, zusätzliche Collector-Klassen über ein Docker Volume ins System einzubinden. Dies würde erlauben, auf ein vorgefertigtes offizielles Docker Image zurückzugreifen, ohne das Analysetool bei jeder Erweiterung neu bauen zu müssen. Für eine solche Erweiterbarkeit wäre eine standardisierte Schnittstelle zur Registrierung externer Collector-Klassen erforderlich.

Quellenverzeichnis

- [1] „Clang, Compiler für C++-Codes“, Adresse: <https://clang.llvm.org/>, besucht am 22.07.2025.
- [2] „Clang GitHub Repository“, Adresse: <https://github.com/llvm/llvm-project/tree/main/clang>, besucht am 22.07.2025.
- [3] „LLVM Homepage“, Adresse: <https://llvm.org/>, besucht am 24.07.2025.
- [4] „LLVM GitHub Repository“, Adresse: <https://github.com/llvm/llvm-project>, besucht am 24.07.2025.
- [5] „GCC Homepage“, Adresse: <https://gcc.gnu.org/>, besucht am 24.07.2025.
- [6] „GCC GitHub Repository“, Adresse: <https://github.com/gcc-mirror/gcc>, besucht am 24.07.2025.
- [7] „LLDB“, Adresse: <https://lldb.llvm.org/>, besucht am 22.07.2025.
- [8] „LLDB GitHub Repository“, Adresse: <https://github.com/llvm/llvm-project/tree/main/lldb>, besucht am 24.07.2025.
- [9] „LLDB Python Dokumentation“, Adresse: https://lldb.llvm.org/python_api.html, besucht am 24.07.2025.
- [10] „Python Homepage“, Adresse: <https://www.python.org/>, besucht am 24.07.2025.
- [11] „Python GitHub Repository“, Adresse: <https://github.com/python/cpython>, besucht am 24.07.2025.
- [12] „Monaco Editor mit React Integration“, Adresse: <https://www.npmjs.com/package/@monaco-editor/react>, besucht am 22.07.2025.
- [13] „Monaco Editor GitHub Repository“, Adresse: <https://github.com/suren-atoyan/monaco-react>, besucht am 24.07.2025.

6.2 Ausblick

- [14] „Visual Studio Code“, Adresse: <https://code.visualstudio.com/>, besucht am 22.07.2025.
- [15] „Visual Studio Code GitHub Repository“, Adresse: <https://github.com/microsoft/vscode>, besucht am 24.07.2025.
- [16] „Docker Homepage“, Adresse: <https://www.docker.com/>, besucht am 24.07.2025.
- [17] „Nextjs: Framework zur Full-Stack Webseitenentwicklung“, Adresse: <https://nextjs.org/>, besucht am 22.07.2025.
- [18] „Nextjs GitHub Repository“, Adresse: <https://github.com/vercel/next.js>, besucht am 24.07.2025.
- [19] „Node.js“, Adresse: <https://nodejs.org/en>, besucht am 30.07.2025.
- [20] „Node.js GitHub Repository“, Adresse: <https://github.com/nodejs/node>, besucht am 30.07.2025.
- [21] „NGINX“, Adresse: <https://nginx.org/>, besucht am 30.07.2025.
- [22] „NGINX GitHub Repository“, Adresse: <https://github.com/nginx/nginx>, besucht am 30.07.2025.
- [23] „React“, Adresse: <https://react.dev/>, besucht am 30.07.2025.
- [24] „React GitHub Repository“, Adresse: <https://github.com/facebook/react>, besucht am 30.07.2025.
- [25] „UI Komponenten Bibliothek shadcn/ui“, Adresse: <https://ui.shadcn.com/>, besucht am 24.07.2025.
- [26] „shadcn/ui GitHub Repository“, Adresse: <https://github.com/shadcn-ui/ui>, besucht am 24.07.2025.
- [27] „Tailwind CSS“, Adresse: <https://tailwindcss.com/>, besucht am 24.07.2025.
- [28] „Tailwind CSS GitHub Repository“, Adresse: <https://github.com/tailwindlabs/tailwindcss>, besucht am 24.07.2025.
- [29] „KI-Tool v0“, Adresse: <https://v0.dev/>, besucht am 24.07.2025.
- [30] „Motion: Zur Animation von DOM-Zustandsänderungen“, Adresse: <https://motion.dev/>, besucht am 22.07.2025.

- [31] „*D3 Bibliothek: Zur Visualisierung von Graphen*“, Adresse: <https://d3js.org/>, besucht am 22.07.2025.
- [32] „*D3 GitHub Repository*“, Adresse: <https://github.com/d3/d3>, besucht am 24.07.2025.
- [33] „*zustand GitHub Repository*“, Adresse: <https://github.com/pmnndrs/zustand>, besucht am 24.07.2025.
- [34] „*constrained-editor-plugin*“, Adresse: <https://github.com/Pranomvignesh/constrained-editor-plugin>, besucht am 22.07.2025.
- [35] „*LLDB Python: lldb.SBThread.StepInto Methode*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBThread.html#lldb.SBThread.StepInto, besucht am 24.07.2025.
- [36] „*LLDB Python: lldb.SBThread.StepOut Methode*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBThread.html#lldb.SBThread.StepOut, besucht am 24.07.2025.
- [37] „*Einschränkung der Berechnung des Return-Values innerhalb LLDB*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBThread.htmllldb.SBThread.return_value, besucht am 22.07.2025.
- [38] „*NDJSON*“, Adresse: <https://github.com/ndjson/ndjson-spec>, besucht am 24.07.2025.
- [39] „*RFC 7464: JSON Text Sequences*“, Adresse: <https://datatracker.ietf.org/doc/html/rfc7464>, besucht am 24.07.2025.
- [40] „*Express.js GitHub Repository*“, Adresse: <https://github.com/expressjs/express>, besucht am 02.08.2025.
- [41] „*Docker Runtime privilege and Linux capabilities*“, Adresse: <https://docs.docker.com/engine/containers/run/#runtime-privilege-and-linux-capabilities>, besucht am 22.07.2025.
- [42] „*KI-Tool ChatGPT*“, Adresse: <https://chatgpt.com/>, besucht am 24.07.2025.

6.2 Ausblick

- [43] „Monaco Dokumentation: *editor.IEditorDecorationsCollection.clear Methode*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/interfaces/editor.IEditorDecorationsCollection.html#clear>, besucht am 24.07.2025.
- [44] „Monaco Dokumentation: *editor.IStandaloneCodeEditor.revealLine Methode*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/interfaces/editor.IStandaloneCodeEditor.html#revealLine>, besucht am 24.07.2025.
- [45] „Monaco Dokumentation: *editor.IStandaloneCodeEditor.createDecorationsCollection Methode*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/interfaces/editor.IStandaloneCodeEditor.html#createDecorationsCollection>, besucht am 24.07.2025.
- [46] „Monaco Dokumentation: *editor.setModelMarkers Methode*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/functions/editor.setModelMarkers.html>, besucht am 24.07.2025.
- [47] „Monaco Dokumentation: *MarkerSeverity Enumeration*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/enums/MarkerSeverity.html>, besucht am 24.07.2025.
- [48] „Monaco Dokumentation: *editor.removeAllMarkers Methode*“, Adresse: <https://microsoft.github.io/monaco-editor/typedoc/functions/editor.removeAllMarkers.html>, besucht am 24.07.2025.
- [49] „LLDB Python: *lldb.SBValue.GetChildAtIndex Methode*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBValue.html#lldb.SBValue.GetChildAtIndex, besucht am 24.07.2025.
- [50] „LLDB Python: *lldb.SBFrame.GetVariables Methode*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBFrame.html#lldb.SBFrame.GetVariables, besucht am 24.07.2025.
- [51] „LLDB Python: *lldb.SBValue.GetValue Methode*“, Adresse: https://lldb.llvm.org/python_api/lldb.SBValue.html#lldb.SBValue.GetValue, besucht am 24.07.2025.

- [52] „Clang Kommandozeilen Dokumentation“, Adresse: <https://clang.llvm.org/docs/CommandGuide/clang.html>, besucht am 24.07.2025.

Abbildungsverzeichnis

4.1	Überblick über den Aufbau des Systems	15
4.2	Screenshots der grafischen Oberfläche	19
4.3	Initiale Seite einer Problemstellung	21
4.4	Initiale Visualisierungsansicht	24
4.5	Initiale Seite einer Problemstellung	26
4.6	Fehlerhafte Implementierung eines Bubblesorts	28
4.7	Initialer Zustand der Quicksort Visualisierung	29
4.8	Markierung des Pivot-Elements in der Quicksort-Visualisierung	30
4.9	Vergleichsoperation in der Quicksort-Visualisierung	31
4.10	Rückgabewert der partition-Funktion	32
4.11	Rekursionsschritt in der Quicksort-Visualisierung	33
4.12	Visualisierung eines Einfüge-Algorithmus in einen Binärbaum	34
4.13	Erfolgreiche Testansicht	36
4.14	Fehlgeschlagene Testansicht	37
4.15	Syntaxfehler, während Computermaus über dem „f“ von „foo“ schwebt	40
4.16	Initialer Codeeditor des Bubblesort-Algorithmus mit ausgeklappten #pragma regions	40
4.17	Komponenten des Analysetools	43
4.18	Implementierung der Analysephase	46
4.19	Auswahl der Debuggingschritte zur Analyse	48
4.20	Erzeugung eines Analyseschritts	49
4.21	Nachverarbeitung mit den Postprocessoren	50
4.22	Implementierung des CurrentLine-Collectors	57
4.23	Datenfluss bei Nutzung von drei Postprocessoren	65