# Praxis der Softwareentwicklung: Entwicklung eines relationalen Debuggers

## Entwurfsdokument


Benedikt Wagner
udpto@student.kit.edu

Chiara Staudenmaier
uzhtd@student.kit.edu

Etienne Brunner
urmlp@student.kit.edu

Joana Plewnia
uhfpm@student.kit.edu

Pascal Zwick
uyqpk@student.kit.edu

Ulla Scheler
ujuhe@student.kit.edu

Betreuer: Mihai Herda, Michael Kirsten


10. Dezember 2017

# Inhaltsverzeichnis

# 1 Einleitung

Dieses Dokument dokumentiert die Ergebnisse der Entwurfsphase (*28.11.-22.12.2017*) im Rahmen des Moduls Praxis der Softwareentwicklung (PSE) am Lehrstuhl „Anwendungsorientierte formale Verifikation - Prof. Dr. Beckert“ am Karlsruher Institut für Technologie (KIT).
Hierbei handelt es sich um den Entwurf des Produkts *DIbugger*, welches im Pflichtenheft definiert wurde. Das Entwurfsdokument beschreibt die Paketeinteilung, Beschreibung der Klassen und Abläufe und genaue Spezifikation der Abhängigkeiten und Kernkomponenten.
Die Implementierung während der Implementierungsphase (*09.01.-02.02.2018*) wird anhand der Vorgaben in diesem Dokument durchgeführt.
Hierbei werden die aus der Softwaretechnik bekannten Prinzipien, wie etwa Geheimnisprinzip und Kapselungsprizip oder das Prinzip der losen Kopplung berücksichtigt. Wie genau die Einhaltung der Prinzipien sichergestellt wird, wird an den jeweiligen Stellen im Dokument erwähnt.

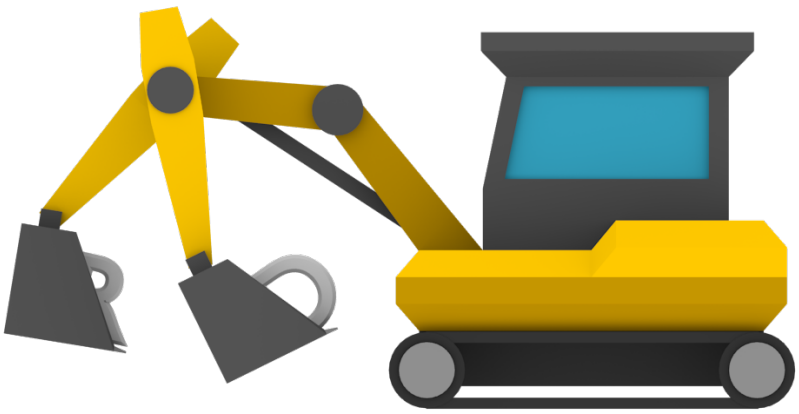

Abbildung 1: Produktlogo

# 2 Paketeinteilung

## 2.1 Übersicht

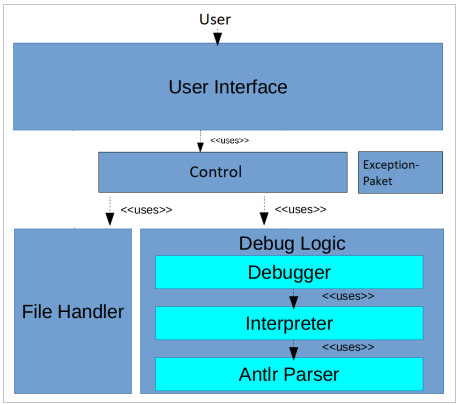


Abbildung 2: Architekturdiagramm

Das Produkt ist aufgeteilt in die Pakete *Control*, *UserInterface*, *FileHandler* und *DebugLogic*. Die *DebugLogic* besteht aus den Unterpaketen *Debugger*, *Interpreter* und *AntlrParser*.
Hierbei wird das Architekturmuster Model-View-Controller (MVC) eingesetzt, um einen flexiblen Programmentwurf zu ermöglichen und die Erweiterbarkeit des Produkts sicher zu stellen. Die Pakete *DebugLogic* und *FileHandler* sind hierbei das Modell, welches die darzustellenden Daten enthält. Das *UserInterface* ist die Präsentationsschicht, welche Benutzereingaben annimmt und die darzustellenden Werte über ein Beobachtermuster erhält. Die Steuerung, welche Benutzereingaben von der Präsentationsschicht erhält und diese auswertet, wird vom *Control*-Paket bereitgestellt.
Alle Pakete stellen ihre Funktionalität über Fassadenklassen nach aussen zur Verfügung. Die genauen Schnittstellen sind also durch diese Klassen definiert.

## 2.2 User Interface

**Aufgaben** Das Paket *UserInterface* stellt die Möglichkeit zur Kommunikation des Nutzers mit dem Produkt dar. Hierbei dient dieses Paket als View Teil des MVC-Konzepts.

### Schnittstellen

- Angebotene Schnittstellen:
  Es wird eine Fassade angeboten, welche es ermöglicht, Variablen, Programmtexte und Eingaben anzuzeigen.

- Genutzte Schnittstellen:
  Dieses Paket nutzt die Fassade des Paketes *Control* und deren angebotenen Schnittstellen.

**Benutztrelation** Das Paket *UserInterface* benutzt die Control-Fassade, um an jegliche, durch den Benutzer angeforderte, Information zu gelangen, bzw. das vom Benutzer geforderte auszuführen.

## 2.3 Control

**Aufgaben** Das Paket *Control* entspricht dem Kontrollsubsystem gemäß dem Architekturstil MVC.
Es dient der Entgegennahme von Benutzerinteraktionen auf der Benutzeroberfläche und Steuerung der Interaktion zwischen den Subsystemen *UserInterface* und *DebugLogic*.
Schaltflächen, sowie Eingabefelder sind Teil des Kontrollsubsystems und werden im Paket *UserInterface* mit Präsentationskomponenten wie dem Variableninspektor zusammengefasst.

**Schnittstellen** Die vom Paket bereitgestellten Methoden sind über eine Fassade aufrufbar.
Die Methoden verursachen Zustandsveränderung des Datenmodells *DebugLogic*.
Beispielsweise kann das Modell aufgefordert werden, einen eingegebenen Quelltext oder spezifizierten Haltepunkt zu speichern oder zu löschen.
Weiter kann das Modell dazu aufgefordert werden, datenbezogene Aktionen auszuführen wie das Starten eines Debugvorgangs, oder Durchführen eines Einzelschrittes. Zusätzlich steuert das Paket Speicher- oder Ladeaufträge an das Paket *FileHandler* geben.

**Benutztrelation** *Control* benutzt die Pakete *DebugLogic* und *FileHandler*.

## 2.4 File Handler

**Aufgaben** Das Paket FileHandler stellt die Funktionalität zum Lesen, Schreiben, Parsen und Interpretieren von sämtlichen Dateien bereit und siedelt sich im Model Teil des MVC-Konzepts an. Dabei wandelt dieser eine Konfigurationsdatei, welche auf dem Dateisystem gespeichert ist, in eine virtuelle Datei um. Diese besteht aus einer Klassenstruktur, welche äquivalent zur Definition des Speicherformats ist, also Zuweisungen und Blöcke. Weiter erzeugt der FileHandler Objekte der Konfigurations-, Sprach- und Einstellungsdateien und kann diese nach außen weitergeben.

### Schnittstellen

- Genutzte Schnittstellen:
  Der FileHandler benötigt keine Schnittstellen anderer Programmpakete, da er an unterster Stelle in der Benutztrelation steht.

- Angebotene Schnittstellen:
  Es werden eine Fassade und drei Klassen angeboten. Diese repräsentieren die Dateien für Produkteinstellungen, Sprachen (Übersetzungen der GUI) und Laufkonfigurationen.

**Benutztrelation** Der FileHandler hat keine Unterpakete Somit entstehen auch keine Abhängigkeiten zu anderen Paketen.

## 2.5 Debug Logic

Das Paket *DebugLogic* stellt den Model Teil der MVC Architektur dar. Die interne Struktur des Paketes ist eine intransparente 3-Schichten-Architektur.
Die unterste Schicht stellt das Subpaket *DebugLogic.AntlrParser* dar. Es erzeugt aus einfachen Zeichenketten Ableitungsbäume nach den Ableitungsregeln der in 7 gegebenen Grammatiken.
Darauf aufbauend in der mittleren Schicht finden sich die Subpakete *DebugLogic.TraceGenerator* und *DebugLogic.RelationalExpressionGenerator*, die beide die Aufgabe haben, diese Ableitungsbäume durch interpretieren in eine abstrakte und leicht handhabbare Form zu bringen. Da beide Subpakete eine gemeinsame Schicht darstellen, findet hier auch ein hohes Maß an Kommunikation statt.

In der obersten Schicht ist das Subpaket *DebugLogic.Debugger* angesiedelt. Dieses nutzt die abstrakten Repräsentationen und führt den eigentlichen Debugprozess darauf aus.

### 2.5.1 Debugger

**Aufgaben** Der Debugger nutzt die von den Subpaketen *DebugLogic.TraceGenerator* und *DebugLogic.RelationalExpressionGenerator* erzeugten Informationen, um Watch-Expressions und bedingte Breakpoints auszuwerten, sowie die üblichen Debugmechanismen zu steuern.

**Schnittstellen** Als oberste Schicht des Paketes *DebugLogic* stellt dieses Subpaket die gleichen Schnittstellen wie die DebugLogic bereit. Diese können in 3 der entsprechenden Fassadenklasse entnommen werden.

**Benutztrelation** Um die üblichen Debugmechanismen wie Schritte und Weiter durchführen zu können, nutzt dieses Subpaket den vom Subpaket *DebugLogic.Interpreter* bereitgestellten Trace-Iterator. Um WatchExpressions und bedingte Breakpoints auszuwerten und zu repräsentieren, nutzt dieses Subpaket die vom Subpaket *DebugLogic.Interpreter* bereitgestellte abstrakte Repräsentationen.

**Interpreter** Dieses Paket ist dafür verantwortlich, die bereits vom *DebugLogic.AntlrParser* geparsten Nutzereingaben so zu verarbeiten, dass der *DebugLogic.Debugger* damit weiterarbeiten kann. Nimmt das Paket vom *DebugLogic.AntlrParser* den Quelltext eines (WLang-) Programms entgegen, erzeugt es einen Pfad über den gesamten Programmfluss des Programms, sodass später darüber iteriert werden kann. Nimmt das Paket Zeichenketten entgegen, die Watch-Expressions und bedingte Breakpoints beschreiben, interpretiert es diese und stellt sie abstrakt dar. Innerhalb dieses Paketes wird auch auf semantische Fehler geprüft, etwa das Fehlen eines return-Statements.

#### Schnittstellen

- Angebotene Funktionalität:
  Stellt einen Iterator über den Ausführungspfad eines gegebenen Programmes zur Verfügung. Erzeugt aus gegebenen Zeichenketten für Watch-Expressions und bedingte Breakpoints eine abstrakte Repräsentation, sodass diese dann leicht ausgewertet werden kann.

- Genutzte Funktionalität:
  Nutzt Syntax-Prüfung und Syntaxbaum-Erzeugung des Subpakets *DebugLogic.AntlrParser*.

**Benutztrelation** Dieses Unterpaket benutzt das Unterpaket *DebugLogic.AntlrParser*, um damit aus den reinen Zeichenketten einen Syntaxbaum gemäß der in 7 gegebenen Grammatik für die Sprache WLang erzeugen zu lassen.

### 2.5.2 Antlr Parser

Dieses Paket beinhaltet nur Klassen, welche von der Antlr Bibliothek auf Basis der WLang Grammatik generiert werden und somit nicht per Hand geschrieben sind.

**Aufgaben** Dieses Unterpaket parst die Eingaben des Nutzers (d.h. sowohl Programmtexte als auch Variablen und Ausdrücke für bedingte Breakpoints und Watch-Expressions) gemäß der in 7 gegebenen Grammatik.

#### Schnittstellen

- Angebotene Funktionalität:
  Prüft die textbasierten Eingaben des Nutzers auf Übereinstimmung mit der gegebenen Grammatik und erzeugt aus der Eingabe einen ablaufbaren Syntaxbaum, der dann vom Unterpaket *DebugLogic.RelationalExpressionGenerator* weiter ausgewertet werden kann.

- Genutzte Funktionalität:
  Benötigt keine Schnittstellen anderer Programmpakete, da das Paket an unterster Stelle der Benutztrelation steht.

**Benutztrelation** Der Antlr Parser hat keine Unterpakete und steht an unterster Stelle der Benutztrelation. Somit entstehen keine Abhängigkeiten zu anderen Paketen.

# 3 Beschreibung wichtiger Klassen

Detaillierte Beschreibung aller Klassen. Das beinhaltet (JavaDoc) Beschreibungen zu allen Me- thoden, Konstruktoren, Packages und Klassen. Was hier nicht reingehört sind

private Felder und Methoden. Das sind Implementierungsdetails.

## 3.1 Klassen im Paket „User Interface“

### Fassade

- Klassenbeschreibung:
  Die Fassade der Benutzeroberfläche (GUIFacade) dient zur Kommunikation mit den anderen Paketen. Um die Benutzeroberfläche einfach austauschen zu können, ist nur die Fassade mit den anderen Paketen verbunden, sodass alle anderen Klassen im Paket User Interface einfach ausgetauscht werden können.

- Methoden:

  - showProgramText(String programText, int id)
    Ermöglicht es einen Programmtext in einem bestimmten Programmfeld (durch ID gekennzeichnet) anzuzeigen

  - reset()
    Ermöglicht es, alle angezeigten Elemente wieder in ihren Ursprungszustand zurückzusetzen

  - showInput(int program, String inputVariables)
    Ermöglicht es, die Eingabevariablen für ein bestimmtes Programm (durch ID gekennzeichnet) anzeigen zu lassen

  - showVariables(int program, List<String> vars)
    Ermöglicht es, die aktuelle Variablenbelegung eines ausgewählten Programms anzuzeigen

  - update()
    Teil des Beobachter-Entwurfsmusters, aktualisiert die Elemente der Benutzeroberfläche, die über die Fassade bei einem Subjekt angemeldet sind.

  - GUIFacade(MainInterface mainInterface)
    Konstruktor für die Fassade. Hier wird ein sogenanntes MainInterface übergeben, welches die Grundlage der Benutzeroberfläche darstellt

  - showError(String s)
    Ermöglicht es, eine Fehlermeldung anzeigen zu lassen

- showWarning(String s)
  Ermöglicht es eine Warnmeldung anzeigen zu lassen

### MainInterface

- Klassenbeschreibung:
  Das sogenannte MainInterface bildet die Grundlage der Benutzeroberfläche. Diese Klasse enthält ein Liste an ProgramPanels , sowie ein CommandPanel und je ein WatchExpression bzw. CondBreakpointPanel. Sie ist verantwortlich für das Anzeigen von Menüs und diesen vier Panelarten.

### ProgramPanel

- Klassenbeschreibung:
  Ein ProgramPanel ist eine Anzeigeeinheit, die alle wichtigen Informationen zu einem einzelnen Programm anzeigt. Hierzu zählen der Programmtext, die Eingabevariablen, die Schrittgröße, der Programmname und die aktuelle Variablenbelegung.

### CommandPanel

- Klassenbeschreibung:
  Ein CommandPanel ist ein Singleton, welches die Buttons zur Kontrolle des Debugvorgangs anzeigt.

- Methoden:

  - getCommandPanel(): CommandPanel
    Gibt das aktuelle CommandPanel zurück, falls es schon existiert, sonst wird ein neues erstellt

### ExpressionPanel

- Klassenbeschreibung:
  Bei dieser Klasse handelt es sich um eine abstrakte Klasse, welche das Anzeigen von Ausdrücken ermöglicht. Für die Ausdrücke WatchExpression und CondBreakpoint wurden nicht abstrakte Unterklassen entworfen.

- Methoden:

  - update()
    Teil des Beobachter-Entwurfsmusters, aktualisiert die Elemente des Expressionpanels, die bei einem Subjekt angemeldet sind.

#### WatchExpressionPanel

- Klassenbeschreibung:
  Ein WatchExpressionPanel ist ein ExpressionPanel, welches Watch-Expressions anzeigt und verwaltbar macht. Diese Klasse ist ein Singleton.

- Methoden :

  - getWatchExpressionPanel(): WatchExpressionPanel
    Gibt das aktuelle WatchExpressionPanel zurück, falls es schon existiert, sonst wird ein neues erstellt

#### CondBreakpointPanel

- Klassenbeschreibung:
  Ein CondBreakpointPanel ist ein ExpressionPanel, welches konditionale Breakpoints anzeigt und verwaltbar macht. Diese Klasse ist ein Singleton.

- Methoden :

  - getConBreakpointPanel(): CondBreakpointPanel
    Gibt das aktuelle CondBreakpointPanel zurück, falls es schon existiert, sonst wird ein neues erstellt

#### DebuggerPopUp

- Klassenbeschreibung:
  Ein DebuggerPopUp ist ein JDialog, welcher auf diese Produkt ausgelegt ist. Es handelt sich um eine abstrakte Klasse, welche die nicht abstrakten Unterklassen ErrorPopUp, WarningPopUp, ArrayValuePopUp und VariableSuggestionPopUp hat. Die Klasse DebuggerPopUp stellt ein Dekorierer Entwurfsmuster mit JDialog dar.

- Methoden:

#### ErrorPopUp

- Klassenbeschreibung:
  Ein ErrorPopUp ist ein DebuggerPopUp, welches eine Fehlermeldung anzeigt.

#### WarningPopUp

- Klassenbeschreibung:
  Ein WarningPopUp ist ein DebuggerPopUp, welches eine Warnung anzeigt, welche ignoriert werden kann, ohne die Funktionalität des Produkts zu schmälern.

#### ArrayValuePopUp

- Klassenbeschreibung:
  Ein ArrayValuePopUp ist ein DebuggerPopUp, welches die Werte eines Arrays anzeigt. Es kann vom Variableninspektor des ProgramPanels aufgerufen werden.

#### VariableSuggestionPopUp

- Klassenbeschreibung:
  Ein VariableSuggestionPopUp ist ein DebuggerPopUp, welches angezeigt wird, um darin z.B. Art und Intervall der Variablenvorschläge eingeben zu können.

## 3.2 Klassen im Paket „Control“

### 3.2.1 Unterpaket Debugger

### 3.2.2 Unterpaket Interpreter

## 3.3 Klassen im Paket „DebugLogic“

## 3.4 Klassen im Paket „FileHandler“

### 3.4.1 Unterpaket FileHandler.Facade

#### FileHandlerFacade

- Klassenbeschreibung:
  Speichert alle verfügbaren Sprachen und hilft bei der Erzeugung von Konfigurationsdateien.

- Methoden:

  - loadConfig(File file) : ConfigurationFile
    Lädt die angegebene Datei als Konfigurationsdatei und gibt diese zurück.

  - saveConfig(ConfigurationFile config)
    Speichert die angegebene Konfiguration an den darin gespeicherten Dateipfad.

  - getPropertiesFile() : PropertiesFile
    Gibt die im Voraus geladene Einstellungsdatei zurück.

  - getLanguages() : List<String>
    Gibt alle Sprachnamen in einer Liste als String zurück.

  - getLanguageFile(String langID) : LanguageFile
    Gibt die zur langID passende Sprachdatei zurück, falls diese existiert, sonst wird eine LanguageNotFoundException geworfen.

#### ConfigurationFile

- Klassenbeschreibung:
  Diese Klasse speichert eine Konfiguration des Debuggers.

- Methoden:

  - getSystemFile() : File
    Gibt ein java.io.File Objekt zurück, welchen die Datei im Dateisystem des Nutzers repräsentiert.

  - getProgramText(int programID) : String
    Gibt den Programmtext von Programm programID als String zurück.

  - getStepSize(int programID) : int
    Gibt die Schrittgröße von Program programID zurück.

  - getInputValue(int programID, String identifier) : String
    Gibt den Eingabewert von Programm programID für die Variable identifier zurück, falls diese existeirt, sonst wird null zurückgegeben.

  - getLatestExecutionLine(int programID) : int
    Gibt die Position von Programm programID im Programmablaufbaum als int zurück.

  - getVariablesOfInspector(int programID) : List<String>
    Gibt eine Liste von variablen Namen zurück, welche im Variablen Inspektor eingeblendet sind.

  - getWEScopeBegin(int expressionID) : List<int>
    Gibt eine Liste der Anfanggrenze der Bereichsintervalle für die WatchExpression expressionID zurück

  - getWEScopeEnd(int expressionID) : List<int>
    Gibt eine Liste der Endgrenze der Bereichsintervalle für die WatchExpression expressionID zurück.

  - getCBScopeBegin(int breakpointID) : List<int>
    Äquivalente Funktion für bedingte Breakpoints zu getWEScopeBegin

  - getCBScopeEnd(int breakpointID) : List<int>
    Äquivalente Funktion für bedingte Breakpoints zu getWEScopeEnd

  - getBreakpoints(int programID) : List<int>
    Gibt eine Liste der Zeilen der Breakpoints für Programm programID zurück.

### PropertiesFile

- Klassenbeschreibung:
  Diese Klasse speichert die Einstellungen des Debuggers.

- getSelectedLanguage() : String
  Gibt den SprachIdentifier der eingestellten Sprache zurück.

- getLastConfigurationFile() : ConfigurationFile
  Gibt eine Repräsentation der zuletzt aktiven Konfiguration zurück.

- getMaxWhileIterations() : int
  Gibt die eingestellte Obergrenze für Schleifendurchläufe wieder.

- getMaxFunctionCalls() : int
  Gibt die eingestelle Obergrenze für Funktionsaufrufe wieder.

### LanguageFile

- Klassenbeschreibung:
  Stellt eine Übersetzung der GUI zu einer bestimmtem Sprache bereit.

- getTranslation(String textID) : String
  Gibt den Text (Übersetzung) zu der übergebenen textID zurück.

### FileReader

- Klassenbeschreibung:
  Dient als Schnittstelle von der FileHandlerFacade zum Dateisystem für das Lesen von verschiedene Dateiformate.

### FileWriter

- Klassenbeschreibung:
  Dient als Schnittstelle von der FileHandlerFacade zum Dateisystem für das Schreiben in verschiedene Dateiformate.

### 3.4.2 Unterpaket FileHandler.RDBF

## 3.5 Klassen im Paket „Exceptions“

...

# 4 Verwendete Design Patterns

## 4.1 Patterns im Paket User Interface

Durch die Nutzung von Swing besteht der Grundaufbau des User Interfaces aus einem Kompositum. Durch anonyme Instanzen von Action Listenern wird das Befehlsmuster implementiert.
Des Weiteren werden die Entwurfsmuster Vererbung (z.B. DebuggerPopUp und ErrorPopUp) und Beobachter (durch das MVC-Konzept vorgegeben) in den Klassen ExpressionPanel und ProgramPanel implementiert. Die Klassen CondBreakpointPanel und WatchExpressionPanel stellen Singletons dar.

## 4.2 Patterns im Paket Control

## 4.3 Patterns im Paket Debug Logic

## 4.4 Patterns im Paket File Handler

# 5 Charakteristische Abläufe

In diesem Kapitel werden charakteristische Abläufe des Produkts, wie der erste Programmaufruf und die Anwendungsfälle, anhand von Sequenzdiagrammen dargestellt und erklärt.

## 5.1 Erster Programmaufruf

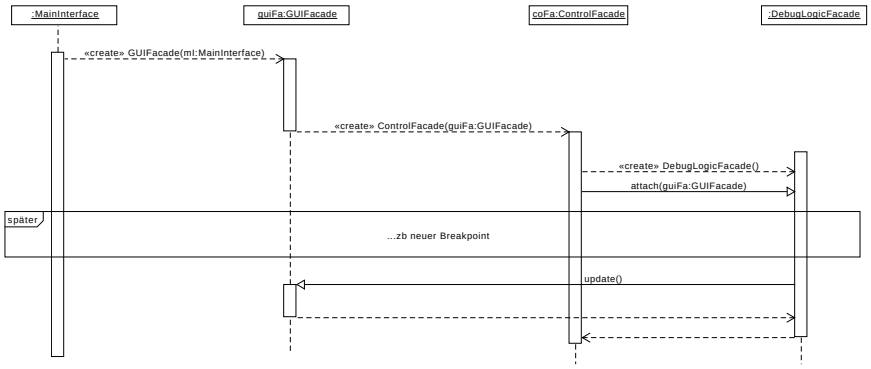


Abbildung 3: Sequenzdiagramm: Erster Programmaufruf

Wird das Produkt gestartet, erstellt die Main-Methode des MainInterface die GUIFacade und übergibt sich selbt. Die GUIFacade speichert das MainInterface und erstellt ihrerseits die ControlFacade, welche wiederum die DebugLogicFacade erstellt. Die ControlFacade und DebugLogicFacade erstellen intern Instanzen der Klassen ihrer Pakete.
Die GUIFacade wird bei diesem Prozess bis zur DebugLogic weitergereicht, um dort als Observer angemeldet werden zu können. Wird später dann zum Beispiel ein Breakpoint hinzugefügt, wird die GUIFacade benachrichtigt und kann sich updaten.

## 5.2 Konfigurationsdatei laden

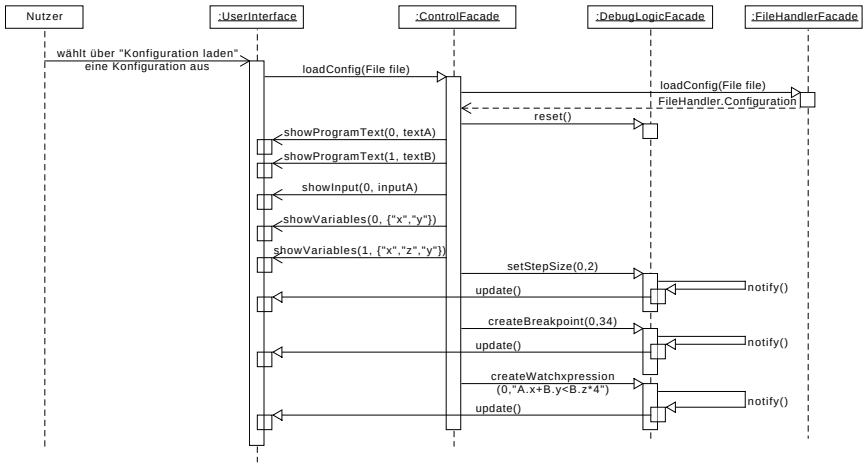


Abbildung 4: Sequenzdiagramm: Laden einer Konfigurationsdatei

Wählt der Benutzer über den Menpeintrag „Konfigurationsdatei laden“ eine Konfiguration aus, gibt das UserInterface diesen Befehl an die Control weiter, welche ein Configuration Objekt vom FileHandler erhält.
Die Control ruft anschließend Methoden der GUIFacade auf, um die Programmtexte, Eingabevariablen und die im Variableninspektor anzuzeigende Variablen anzuzeigen. Außerdem ruft die Control Mathoden der DebugLogicFacade auf, um für jedes Programm die Breakpoints, Watch-Expressions und Schrittgrößen festzulegen. Über diese Änderungen wird das UserInterface als Observer benachrichtigt und kann diese ebenfalls anzeigen

## 5.3 Konfigurationsdatei speichern

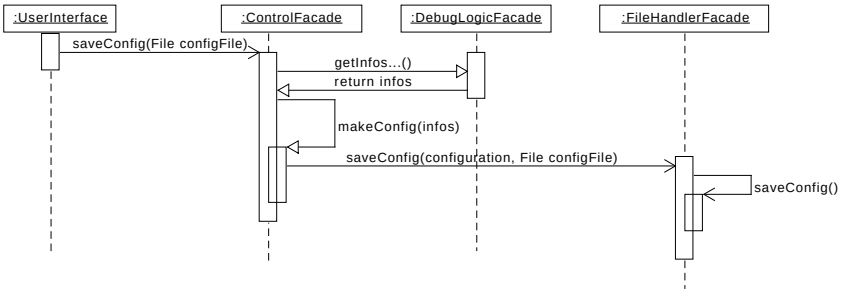


Abbildung 5: Sequenzdigramm: Speichern einer Konfigurationsdatei

Möchte der Benutzer eine Konfigurationsdaatei speicher, reicht das UserInterface den Speicherort an die ControlFacade weiter. Die Control sammelt die benötigten Daten in einer Configuration Instanz. Dieses Objekt wird mit dem angegebenen Speicherort an die FileHandlerFacade weitergegeben, welche dann die Konfigurationsdatei auf dem Rechner des Benutzers speichert.

## 5.4 AF10: Hinzufügen von Programmen

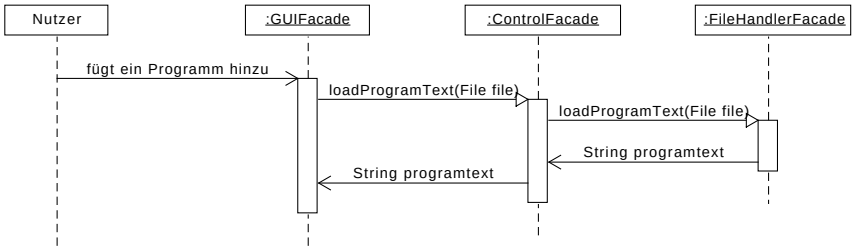


Abbildung 6: Sequenzdiagramm: Hinzufügen von Programmen

AF10 ist falsch

## 5.5 AF20: Ändern von Programmen

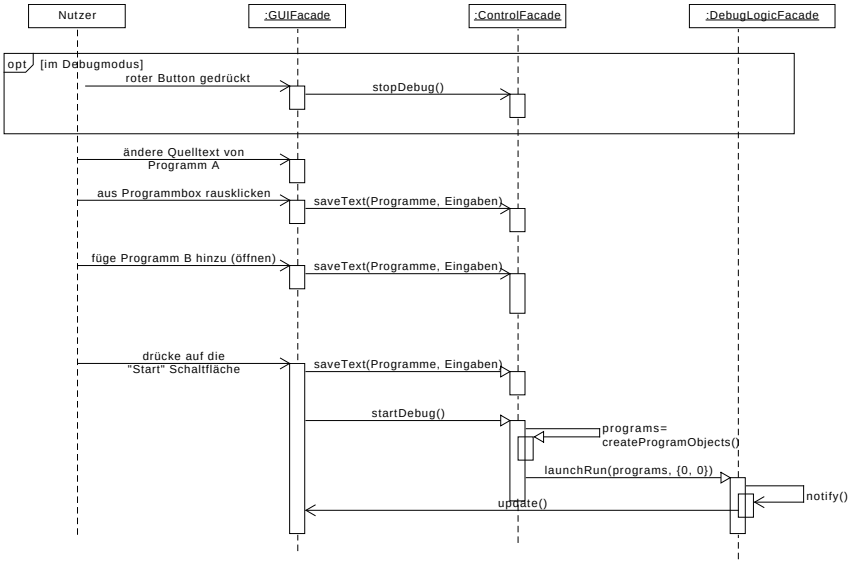


Abbildung 7: Sequenzdiagramm: Ändern von Programmen

Möchte der Benutzer einen Programmtext editieren, muss er gegebenenfalls zunächst den Debugmodus beenden. Anschließend lässt sich der Programmtext im Textfeld bearbeiten. Sobald der Benutzer außerhalb des Textfelds klickt, gibt das MainInterface den neuen Programmtext und die Eingabeavariablen an die ControlFacade weiter.
Fügt der Benutzer einen neuen Programmtext durch Öffnen einer Datei hinzu, gibt das MainInterface diesen ebenfalls mit den angegebenen Eingabevariablen an die ControlFacade weiter.
Sobald der Benutzer die Start-Schaltfläche auswählt um den Debugmodus zu starten, gibt das MainInterface erneut alle eingegebenen Texte weiter und ruft schließlich startDebug() der ControlFacade auf. Diese erstellt aus den gespeicherten Informationen Programm-Instanzen und gibt diese an die DebugLogicFacade weiter und startet damit den Debug-Lauf.

## 5.6 AF30: Setzen von Breakpoints

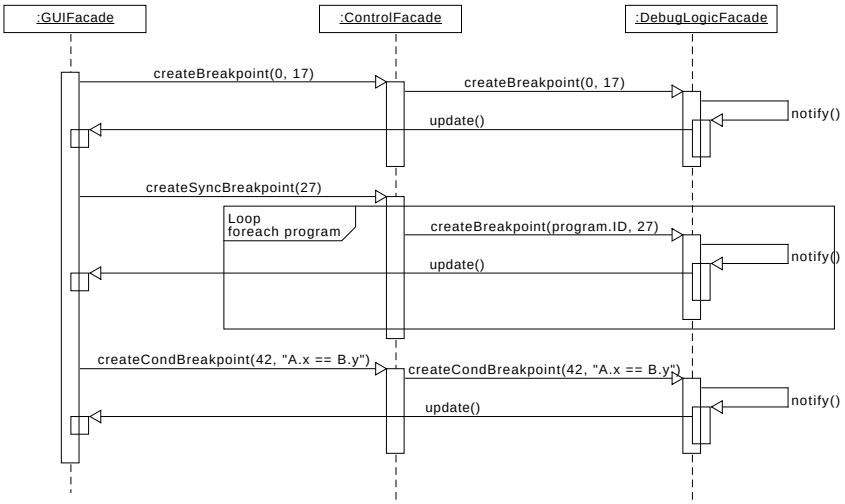


Abbildung 8: Sequenzdiagramm: Setzen von Breakpoints

Setzt der Benutzer einen Breakpoint in eine Zeile in einem Programm, reicht das MainInterface diese Information an die ControlFacade weiter, welche dann createBreakpoint mit der Programm-ID und der Zeile an die DebugLogicFacade weitergibt. Setzte der Benutzer jedoch einen Breakpoint in allen Programmen, teilt das MainInterface dies der ControlFacade mit. Die ControlFacade ruft für jedes Programm die Methode createBreakpoint der DebugLogicFacade auf. Beim hinzufügen von bedingten Breakpoints gibt das MainInterface die ID des Breakpoints und den vom Benutzer angegebenen Ausdruck an die ControlFacade weiter. Die Control reicht diese Informationen ihrerseits an die DebugLogicFacade weiter, welche den Breakpoint ab diesem Zeitpunkt auswertet.

## 5.7 AF40: Hinzufügen von Watch-Expressions

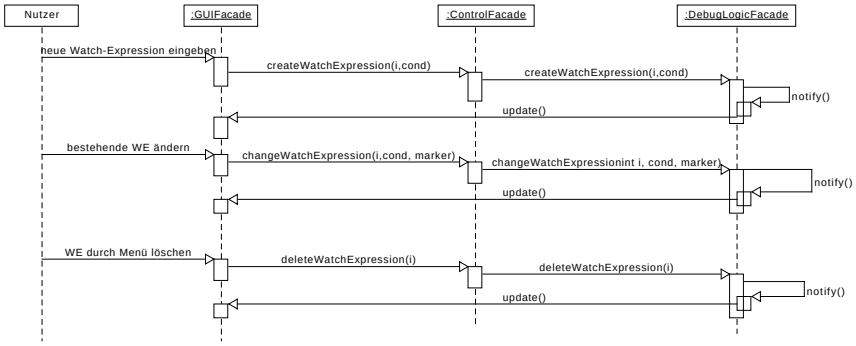


Abbildung 9: Sequenzdiagramm: Hinzufügen von Watch-Expressions

Dieser Vorgänge sind für Watch-Expressions und bedingte Breakpoints identisch.
Gibt der Benutzer eine neue Watch-Expression an, wird die Bedingung und die ID vom MainInterface über die Control an die DebugLogicFacade weitergegeben. Ändert der Benutzer die Watch-Expression, zB indem er die Bereichsbindung angibt, wird dies ebenfalls über die Control an die DebugLogicFacade weitergegeben. Auch beim Löschen einer Watch-Expression über das entsprechende Menü der Benutzeroberfläche, erhält die DebugLogic diese Information über die Control und stoppt das Auswerten dieser Watch-Expression.

## 5.8 AF50: Programme debuggen

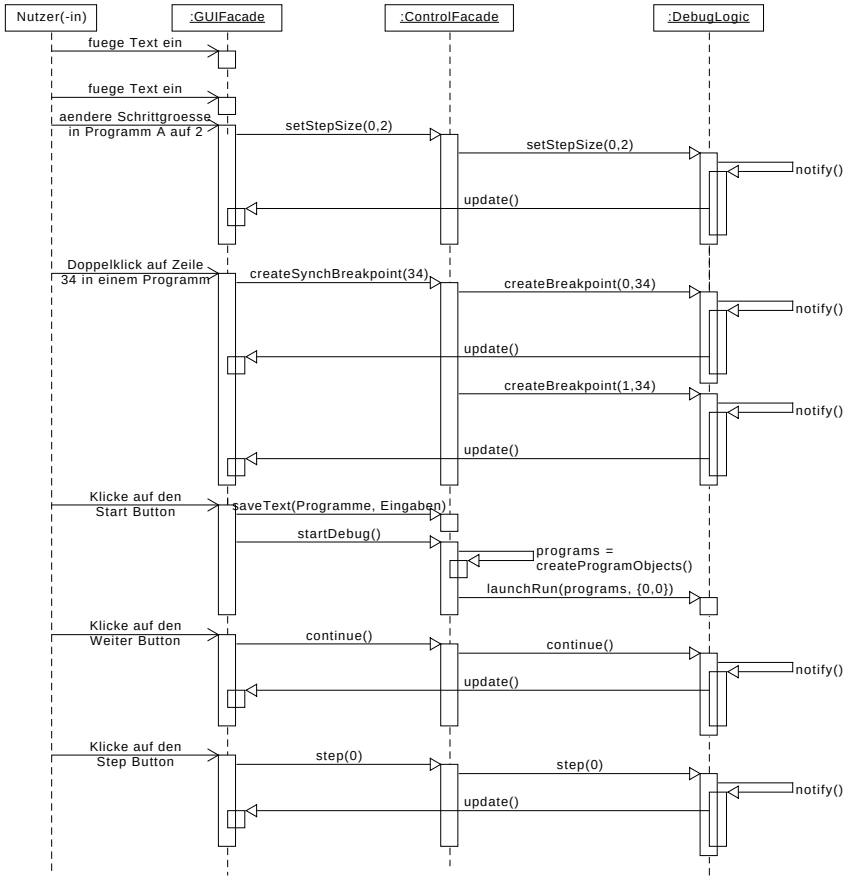


Abbildung 10: Sequenzdiagramm: Debuggen von Programmen

Dieses Sequenzdiagramm fasst die Schritte des Nutzers bei einem Debug-Lauf zusammen. Hierbei werden zuvor Programme hinzugefügt, die Schrittgröße geändert und Breakpoints hinzugefügt. Sobald der Benutzer auf Start klickt, werden seine Eingaben final gespeichert und die Control startet den DebugLauf der DebugLogic. Wenn der Benutzer durch Weiter oder Schritt durch den DebugLauf navigiert, werden diese Befehle über die Control an die DebugLogic weitergegeben. Die Schritte werden von der DebugLogic ausgeführt, und anschließend wird die GUIFacade als Beobachter dazu aufgefordert, die aktualisierten Werte anzuzeigen.

## 5.9 Erzeugung abstrakter Strukturen im Subpaket Interpreter

In diesem Abschnitt soll die Funktionalität des Subpaketes *DebugLogic.Interpreter* beschrieben werden. Der Interpreter hat im Wesentlichen zwei Aufgaben: Einerseits muss er für WatchExpressions und bedingte Breakpoints eine abstrakte Struktur aufbauen, die sich einfach auswerten lässt. Andererseits hat er die Aufgabe, den kompletten Programmverlauf(im Folgenden „Trace“) eines Programmes zu berechnen und einen Iterator darüber bereitzustellen.

### 5.9.1 Erzeugung einer abstrakten Repräsentation für Watch-Expressions und bedingte Breakpoints

Eine Watch-Expression kann entweder aus einem Term bestehen, oder aus einer Bedingung. Ein bedingter Breakpoint hingegegen muss aus einer Bedingung bestehen. Der Zusammenhang von Bedingungen und Termen soll zunächst besprochen werden. In Dia-

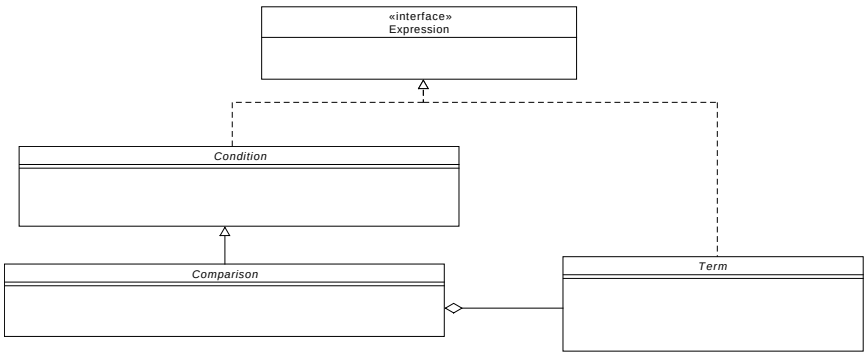


Abbildung 11: Terme und Bedingungen

gramm 11 sind der Übersichtlichkeit halber nur die abstrakten Klassen ohne Methoden oder Attribute zu sehen. So sind etwa die zahlreichen Unterklassen von *Condition, Comparison* und *Term* ausgespart. Bei *Term* und *Condition* handelt es sich überdies um ein Kompositum. Dies wird genauer in 4 erklärt. Wichtig ist hierbei vor allem, dass ein Vergleich eine spezielle Bedingung darstellt, die zwei Terme enthält und diese miteinander vergleicht. Jede Expression ist auswertbar. So nimmt eine Bedingung stehts einen boolschen Wert an, während ein Term sich zu einem beliebigen Datentyp auswerten kann. Wir betrachten das Diagramm 12 Das Debuggerpaket erzeugt einen neuen Bedingten Breakpoint und übergibt dazu eine Zeichenkette, die diesen spezifiziert. Der Bedingte Breakpoint nutzt das Paket *AntlrParser*, um einen Syntaxbaum zu erzeugen. Dann wird ein *ConditionGenerationVisitor* gestartet, der diesen Syntaxbaum abläuft und dabei eine *Condition* erzeugt. Da eine *Comparison* etwa auch zwei *Term* Objecte benötigt, wird von

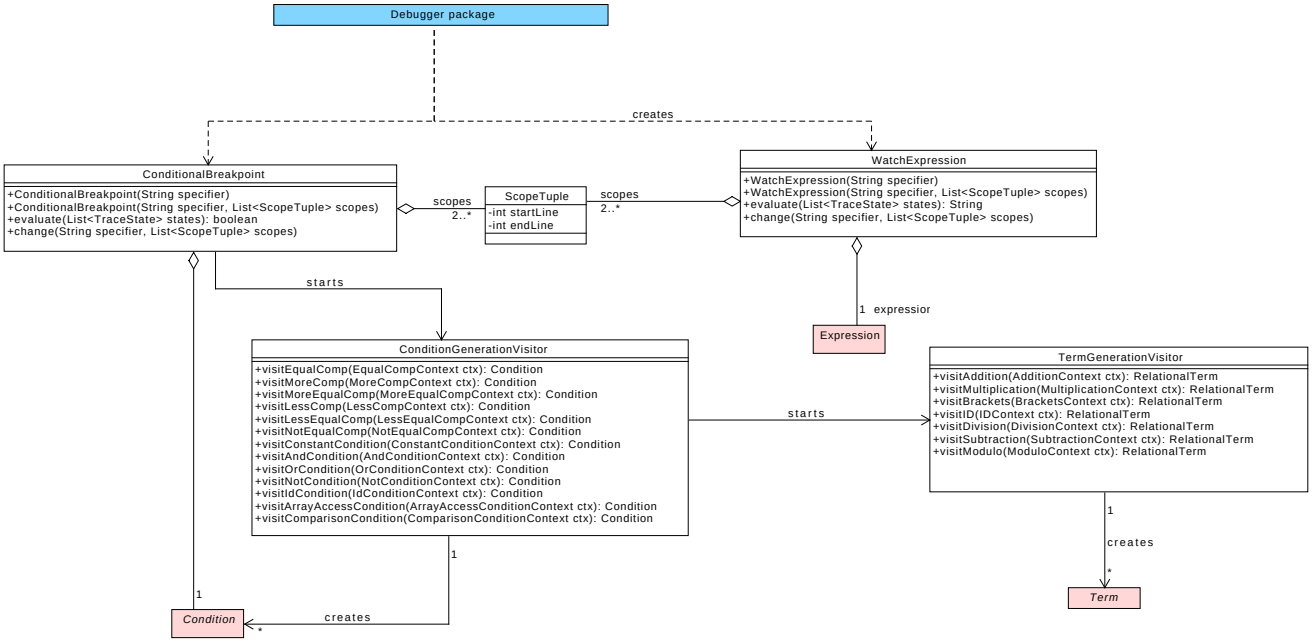


Abbildung 12: Watch-Expressions und Bedingte Breakpoints

diesem an entsprechender Stelle der *TermGenerationVisitor* aufgerufen. Beim Erzeugen einer WatchExpression passiert das gleiche, wobei es sich auch um reine Terme handeln kann, sodass ein *ConditionGenerationVisitor* eventuell gar nicht gebraucht wird.

### 5.9.2 Erzeugung des Traces

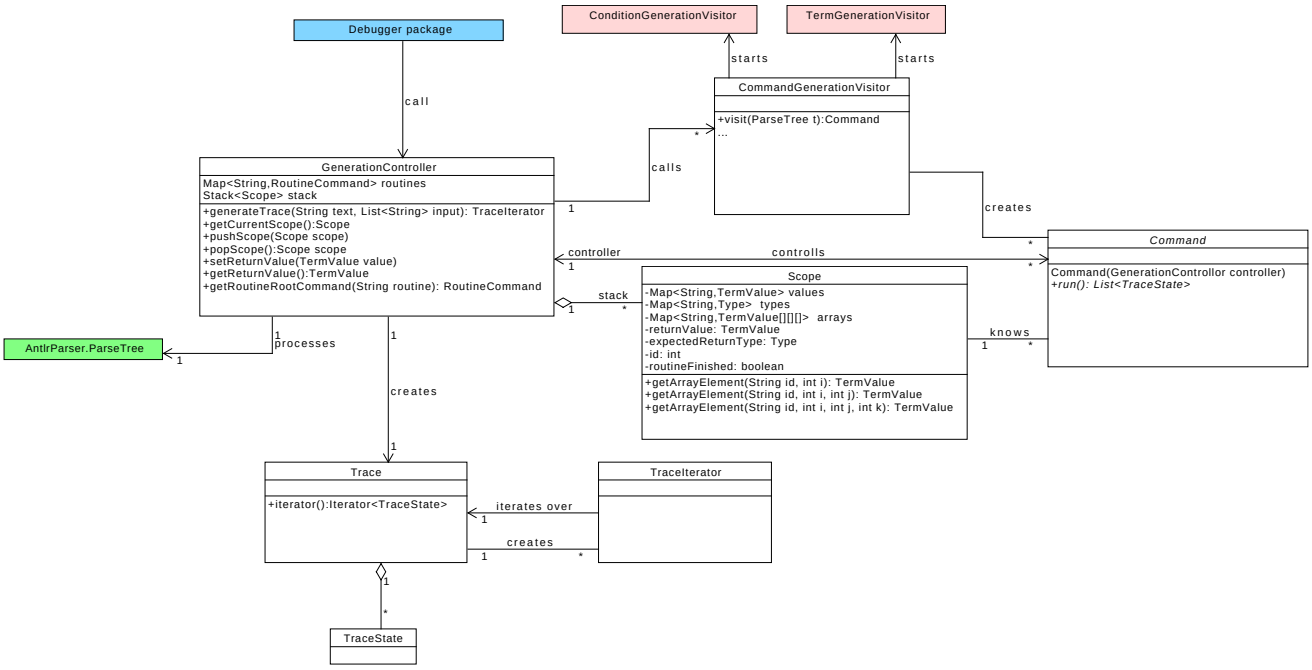


Abbildung 13: An der Tracegenerierung beteiligte Klassen

Gegeben sie das folgende (Wlang-)Programm:

```
int foo() {
    return 1+2+3;
}
int main(){
    int x;
    int y=3;
    while(x<(y+7)%4){
        x=x+1;
    }
    return c;
}
```

Die Zusammenarbeit der in 13 gegebenen Klassen soll nun an diesem Beispiel erklärt werden. Die Klasse *GenerationController* steuert das Verfahren der Traceerzeugung. Sie bekommt in der Methode *generateTrace()* den Quelltext und die Eingaben für diesen übergeben. Zunächst wird vom Subpaket *DebugLogic.AntlrParser* ein *ParseTree* erzeugt. Für obiges Beispielprogramm ist dieser in 14 zu sehen. Über diesen Baum läuft

Abbildung 14: Beispiel für einen Ableitungsbaum

der *CommandGenerationVisitor* und erzeugt für jede Routine einen Baum aus Befehlen(*Command*- Kompositum, siehe 4). Dabei verwendet er zwei weitere Visitor, die ihm Bedingungen und Terme in Form der in 11 gegebenen Klassen erzeugen. Die Wurzeln dieser Bäume aus Befehlen werden im *GenerationController* in der Map *routines* gespeichert, sodass sie dann über ihren Routinennamen aufrufbar sind. Die Objektstruktur nach diesem Schritt sieht dann wie im Objektdiagramm 15 aus. Die tatsächliche Aus-

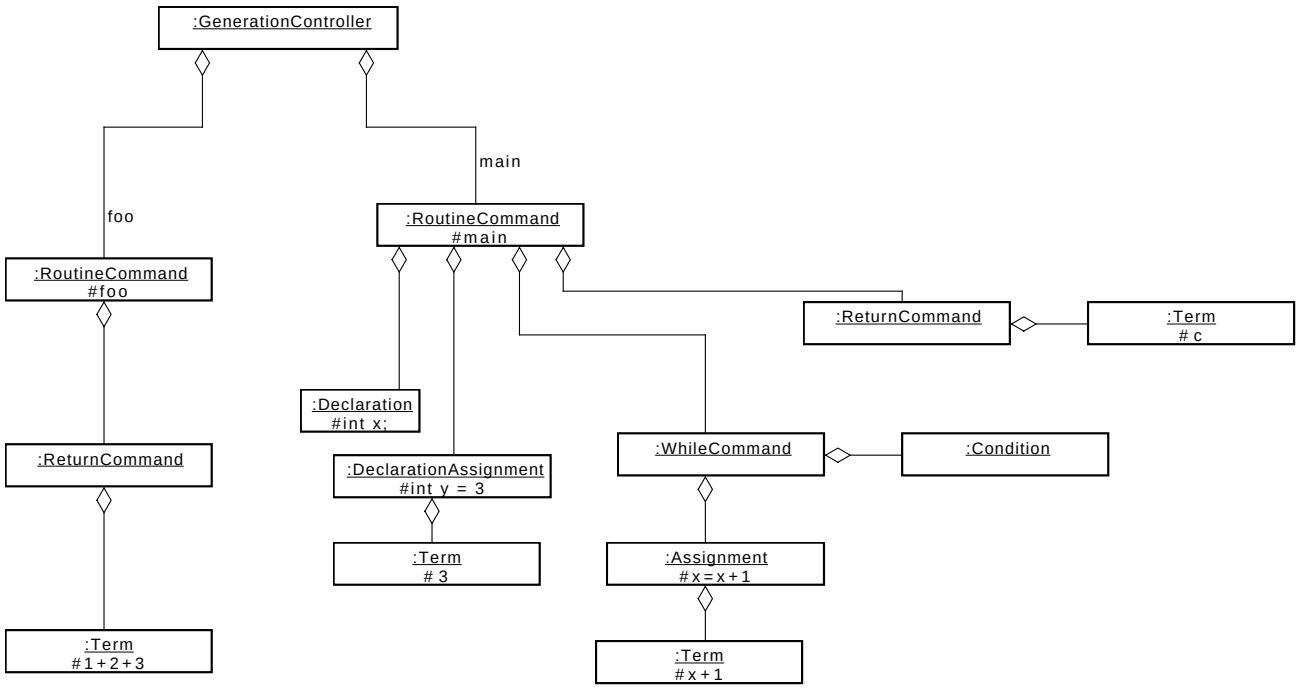


Abbildung 15: Objektstruktur nach der Commanderzeugung

führung der Commands folgt dann: Der *GenerationController* enthält einen Stack aus *Scope*-Objekten. In diesen Objekten ist alles über die zum aktuellen Ausführungszeit-

punkt vorhandenen Daten gespeichert, etwa die Datentypen oder Werte der Variablen. Zu Beginn legt der *GenerationController* einen „Urscope“ auf seinen Stack. Dieser enthält lediglich die Eingabevariablen. Dann wird die *run()*-Methode des Wurzelbefehls des Befehlsbaums der Mainroutine ausgeführt. Jeder Befehl ändert dann entprechend seiner Semantik die Werte im aktuell obersten Scope auf dem Stack. Dabei muss auch jeder Befehl eine entsprechende Typprüfung durchführen. Ein *IfCommand* beispielsweise prüft seine Bedingung und führt seine Kinderbefehle im Baum aus, falls die Bedingung wahr ist.
Einen Spezialfall stellen die *RoutineCall*-Commands dar. Ein solcher Befehl muss sich beim *GenerationController* den Wurzelbefehl der passenden Routine holen, diesem die Argumente in Form einer Liste von *Term*-Objekten übergeben, und diesen dann ausführen. Anschließend muss er sich vom *GenerationController* den Rückgabewert holen. Dieser Wurzelknoten ist ein Objekt der *Command*-Subklasse *RoutineCommmand*. Solche Commands müssen bei Ausführung zunächst die Argumente holen, dann einen neues *Scope*-Objekt auf den Stack legen, dann alle Kinder ausführen und danach den Scope wieder weglegen. Hierbei muss im Falle einer Funktion auch geprüft werden, ob ein Rückgabewert vorhanden ist.
Einen weiteren Spezialfall stellen *ReturnCommand*-Objekte dar. Sie müssen im *GenerationController* den Rückgabewert setzen und diesem mitteilen, dass die Routine beendet ist. Die darauffolgenden Befehle erkennen dies vor ihrerer Ausfühurung und führen sich nicht weiter aus.
Insgesamt gibt jeder Befehl nach der *run()*-Methode eine Liste von *TraceState*-Objekten zurück. Die gesamte Liste verpackt der *GenerationController* zu einem *Trace*-Objekt und gibt einen *TraceIterator* zurück.

# 6 Abhängigkeitseinteilung mit Blick auf die Implementierung

## 6.1 Abhängikeiten

Das Paketdiagramm besagt, dass FileHandler und DebugLogic nicht voneinader abhängen. Beide werden von der Control benutzt und müssen somit korrekt implementiert sein, bevor die Control richtig funktionieren kann. Weiter führt die Control Methoden der GUIFacade aus, ist aber nicht von derer Abhängig, da diese nicht relevant ist für eine funktionierende Control. Die GUI hingegen ist stark von der Control abhängig und kann ohne diese nicht korrekte Daten anzeigen.

## 6.2 Implementierung

Die Implementierung der Hauptpakete kann durch den von Fassaden geprägten Entwurf und dem MVC-Konzept gleichzeitig geschehen. Es ist jedoch sinnvoll das Exception Paket als erstes zu implementieren, um später keine Stellen im Quelltext suchen zu müssen, an denen Fehler auftreten können. Weiter sollten die Unterpakete der DebugLogic entsprechend der Benutztrelation implementiert werden, also zuerst AntlrParser, Interpreter und zum Schluss der Debugger. Dieses Problem besteht bei den Hauptpaketen nicht, da für die Kommunikation ausschließlich Fassaden benutzt werden. Diese können zur gleichzeitigen Implementierung der Pakete als Stummel pseudoerstellt werden, d.h. die Methoden sind leer und enthalten keine bzw. nur sehr wenig Funktionalität.
Der FileHandler stellt wiederum eine Ausnahme dar, da er mehrere Klassen zur Repräsentation und Interaktion zum Dateisystem bereitstellt. Hierbei müssen die Klassen ConfigurationFile, PropertiesFile und LanguageFile gleichzeitig mit der FileHandlerFacade Klasse einsatzbereit sein.

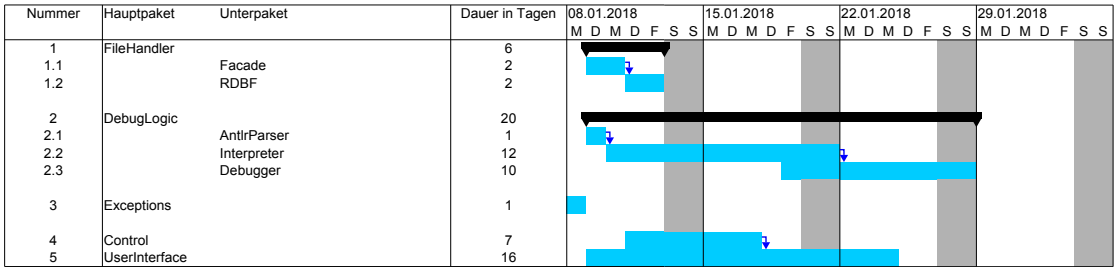


Abbildung 16: Gantt Diagramm: Zeitplanung der Implementierung

# 7 Formale Spezifikation von WLang und Speicherformaten

Startnichtterminal in den Grammatiken ist immer die Variable $r$.

## 7.1 Konntextfreie Antlr-Grammatik für WLang-Syntax

```
grammar Wlang;
r: programm;

programm: routine* mainRoutine;
routineHead: returntype = TYPE id = ID '(' args=arglist? ')' #FunctionHead
| 'void' id =ID '('args=arglist?')' #ProcedureHead
```

```
;

mainHead: returntype = TYPE 'main' '(' args=arglist? ')' #MainFunctionHead
| 'void' 'main' '('args=arglist?')' #MainProcedureHead
;

arglist: argument ',' arglist | argument;
argument: type=TYPE id=ID;
filledArglist: filledArgument ',' filledArglist | filledArgument;
filledArgument: term;
routine: routineHead block;
mainRoutine: mainHead block;

//Statements

statements : statement statements #CompStatement
| statement #SingleStatement
;
statement:  ifState
| ifelseState
| whileState
| assignment
| arrayDeclaration
| arrayDeclareAssign
| arrayElementAssign
| declaration
| funcCall ';'
| returnState;

funcCall: functionname = ID '(' args=filledArglist? ')'
  |functionname = 'main' '(' args=filledArglist? ')'
;


block: '{'statements'}';
assignment: declareAssign
| pureAssign
;

arrayDeclaration: type = TYPE dims id = ID ';';
arrayDeclareAssign: type = TYPE dims id = ID ASSIGN '{'filledArglist'};';
arrayElementAssign: arrayAccess ASSIGN value = term';';
dims: '['term']' #oneDims
| '['term']''['term']' #twoDims
```

```
| '['term']''['term']''['term']' #threeDims
;

pureAssign: id = ID ASSIGN value = term ';';
declareAssign: type = TYPE id = ID ASSIGN value = term ';';
declaration: type = TYPE id = ID ';';
returnState: 'return' returnvalue = term ';';

//Kontrollstrukturen
ifState: 'if' '(' condition ')' block
| 'if' '(' condition ')' statement
;
ifelseState: ifState 'else' block
| ifState 'else' statement
;
whileState: 'while' '(' condition ')' block
| 'while' '(' condition ')' statement
;
//Bedingungen
condition: ID #IdCondition
| arrayAccess #ArrayAccessCondition
| comparison #ComparisonCondition
| BOOLEANLITERAL #ConstantCondition
| '('condition')' '||' '('condition')' #OrCondition
| '('condition')' '&&' '('condition')' #AndCondition
| '!'condition #NotCondition
;

comparison: left=term '<' right=term #LessComp
   | left=term '<=' right=term #LessEqualComp
   | left=term '>' right=term #MoreComp
   | left=term '>=' right=term #MoreEqualComp
   | left=term '==' right=term #EqualComp
   | left=term '!=' right=term #NotEqualComp
;

term : '-' inner = term #NegativeTerm
    | left = term '/' right = term #Division
| left = term '*' right = term #Multiplication
| left = term '-' right = term #Subtraction
| left = term '+' right = term #Addition
| left = term '%' right = term #Modulo
|'('term')' #Brackets
| FLOATLITERAL #FloatLiteral
```

```
| INTLITERAL #IntLiteral
| LONGLITERAL #LongLiteral
| DOUBLELITERAL #DoubleLiteral
| ID #ID
| CHARLITERAL #CharLiteral
| funcCall #FunctionCallInTerm
| arrayAccess #ArrayAccessInTerm
;

arrayAccess: id = ID '['index=term']' #OneDimArrayAccess
| id = ID '['firstIndex=term']' '['secondIndex=term']' #TwoDimArrayAccess
| id = ID '['firstIndex=term']' '['secondtIndex=term']' '['thirdIndex=term']' #ThreeDimArr
;

//LITERALE bzw TOKENS

COMPOPERATOR:  '<'|'>'|'<='|'>='|'=='|'!=';
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT:   '/*' .*? '*/' -> skip;
LINE_COMMENT:   '//' ~[\r\n]* -> skip;
TYPE: 'float' | 'int' | 'char' | 'boolean' | 'double' | 'long';
ID : ([a-z]|[A-Z])+ ;
INTLITERAL: '-'? [1-9][0-9]* | '0';
FLOATLITERAL: ([1-9][0-9]*'.'[0-9]+ | '0') 'f';
CHARLITERAL: '\'' ~['\\\r\n] '\'';
BOOLEANLITERAL: 'true'| 'false';
NULLLITERAL: 'null';
LONGLITERAL: ([1-9][0-9]* | '0' )'L';
DOUBLELITERAL: [1-9][0-9]*'.'[0-9]+ | '0';
ASSIGN: '=';
```

## 7.2 Kontextfreie Antlr-Grammatik für Syntax von Watch-Expressions

```
grammar WatchExpressions;
r: we;

we: condition | term

condition: ID #IdCondition
| arrayAccess #ArrayAccessCondition
| comparison #ComparisonCondition
```

```
| BOOLEANLITERAL #ConstantCondition
| '('condition')' '||' '('condition')' #OrCondition
| '('condition')' '&&' '('condition')' #AndCondition
| '!'condition #NotCondition
;

//Bedingungen
comparison: left=term '<' right=term #LessComp
   | left=term '<=' right=term #LessEqualComp
   | left=term '>' right=term #MoreComp
   | left=term '>=' right=term #MoreEqualComp
   | left=term '==' right=term #EqualComp
   | left=term '!=' right=term #NotEqualComp
;


term : '-' inner = term #NegativeTerm
    | left = term '/' right = term #Division
| left = term '*' right = term #Multiplication
| left = term '-' right = term #Subtraction
| left = term '+' right = term #Addition
| left = term '%' right = term #Modulo
|'('inner = term')' #Brackets
| FLOATLITERAL #FloatLiteral
| INTLITERAL #IntLiteral
| LONGLITERAL #LongLiteral
| DOUBLELITERAL #DoubleLiteral
| ID #ID
| CHARLITERAL #CharLiteral
| arrayAccess #ArrayAccessInTerm
;

arrayAccess: id = ID '['index=term']' #OneDimArrayAccess
| id = ID '['firstIndex=term']' '['secondIndex=term']' #TwoDimArrayAccess
| id = ID '['firstIndex=term']' '['secondtIndex=term']' '['thirdIndex=term']' #ThreeDimArr
;

//LITERALE bzw TOKENS

COMPOPERATOR:  '<'|'>'|'<='|'>='|'==';
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT:   '/*' .*? '*/' -> skip;
LINE_COMMENT:   '//' ~[\r\n]* -> skip;
TYPE: 'float' | 'int' | 'char' | 'boolean';
```

```
ID : [A-Z]'.'([a-z]|[A-Z])+ ;
INTLITERAL: [1-9][0-9]* | '0';
FLOATLITERAL: ([1-9][0-9]*'.'[0-9]+ | '0') 'f';
CHARLITERAL: '\'' ~['\\\r\n] '\'';
BOOLEANLITERAL: 'true'| 'false';
NULLLITERAL: 'null';
LONGLITERAL: ([1-9][0-9]* | '0' )'L';
DOUBLELITERAL: [1-9][0-9]*'.'[0-9]+ | '0';
ASSIGN: '=';
```

## 7.3 Kontextfreie Antlr-Grammatik für Syntax vonbedingten Breakpoints

```
grammar Assertions;
r: condition;

condition: ID #IdCondition
| arrayAccess #ArrayAccessCondition
| comparison #ComparisonCondition
| BOOLEANLITERAL #ConstantCondition
| '('condition')' '||' '('condition')' #OrCondition
| '('condition')' '&&' '('condition')' #AndCondition
| '!'condition #NotCondition
;

//Bedingungen
comparison: left=term '<' right=term #LessComp
   | left=term '<=' right=term #LessEqualComp
   | left=term '>' right=term #MoreComp
   | left=term '>=' right=term #MoreEqualComp
   | left=term '==' right=term #EqualComp
   | left=term '!=' right=term #NotEqualComp
;


term : '-' inner = term #NegativeTerm
    | left = term '/' right = term #Division
| left = term '*' right = term #Multiplication
| left = term '-' right = term #Subtraction
| left = term '+' right = term #Addition
| left = term '%' right = term #Modulo
```

```
|'('inner = term')' #Brackets
    | INTLITERAL #IntLiteral
| FLOATLITERAL #FloatLiteral
| LONGLITERAL #LongLiteral
| DOUBLELITERAL #DoubleLiteral
| ID #ID
| CHARLITERAL #CharLiteral
| arrayAccess #ArrayAccessInTerm
;

arrayAccess: id = ID '['index=term']' #OneDimArrayAccess
| id = ID '['firstIndex=term']' '['secondIndex=term']' #TwoDimArrayAccess
| id = ID '['firstIndex=term']' '['secondtIndex=term']' '['thirdIndex=term']' #ThreeDimArr
;

//LITERALE bzw TOKENS

COMPOPERATOR:  '<'|'>'|'<='|'>='|'==';
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT:   '/*' .*? '*/' -> skip;
LINE_COMMENT:   '//' ~[\r\n]* -> skip;
TYPE: 'float' | 'int' | 'char' | 'boolean';
ID : [A-Z]'.'([a-z]|[A-Z])+ ;
INTLITERAL: [1-9][0-9]* | '0';
FLOATLITERAL: ([1-9][0-9]*'.'[0-9]+ | '0') 'f';
CHARLITERAL: '\'' ~['\\\r\n] '\'';
BOOLEANLITERAL: 'true'| 'false';
NULLLITERAL: 'null';
LONGLITERAL: ([1-9][0-9]* | '0' )'L';
DOUBLELITERAL: [1-9][0-9]*'.'[0-9]+ | '0';
ASSIGN: '=';
```

# 8 Änderung zum Pflichtenheft

Änderungen zum Pflichtenheft, z.B. gekürzte Wunschkriterien.

# 9 Anhang

UML-Klassendiagramm Vollständiges großformatiges Klassendiagramm im Anhang. Ausschnitte/Teile können bereits vorher verwendet werden, um Teilkomponenten zu beschreiben. Assoziationen zwischen Klas- sen dabei bitte mit entsprechenden Pfeilen darstellen, statt nur durch Feldtypen.