

# Praxis der Softwareentwicklung: Entwicklung eines relationalen Debuggers

## Entwurfsdokument

Benedikt Wagner  
udpto@student.kit.edu

Etienne Brunner  
urmlp@student.kit.edu

Pascal Zwick  
uyqpk@student.kit.edu

Chiara Staudenmaier  
uzhtd@student.kit.edu

Joana Plewnia  
uhfpm@student.kit.edu

Ulla Scheler  
ujuhe@student.kit.edu

Betreuer: Mihai Herda, Michael Kirsten

10. Dezember 2017

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Paketeinteilung</b>	<b>2</b>
2.1	Übersicht . . . . .	2
2.2	User Interface . . . . .	3
2.3	Control . . . . .	3
2.4	File Handler . . . . .	4
2.5	Debug Logic . . . . .	4
2.5.1	Debugger . . . . .	5
2.5.2	Antlr Parser . . . . .	6
<b>3</b>	<b>Beschreibung wichtiger Klassen</b>	<b>6</b>
3.1	Klassen im Paket „User Interface“ . . . . .	7
3.2	Klassen im Paket „Control“ . . . . .	7
3.2.1	Unterpaket Debugger . . . . .	7
3.2.2	Unterpaket Interpreter . . . . .	7
3.3	Klassen im Paket „DebugLogic“ . . . . .	7
3.4	Klassen im Paket „FileHandler“ . . . . .	7
3.4.1	Unterpaket FileHandler.Facade . . . . .	7
3.4.2	Unterpaket FileHandler.RDBF . . . . .	10
3.5	Klassen im Paket „Exceptions“ . . . . .	10
<b>4</b>	<b>Verwendete Design Patterns</b>	<b>10</b>
4.1	Patterns im Paket User Interface . . . . .	10
4.2	Patterns im Paket Control . . . . .	11
4.3	Patterns im Paket Debug Logic . . . . .	11
4.4	Patterns im Paket File Handler . . . . .	11
<b>5</b>	<b>Charakteristische Abläufe</b>	<b>12</b>
5.1	Erster Programmaufruf . . . . .	12
5.2	Konfigurationsdatei laden . . . . .	13
5.3	Konfigurationsdatei speichern . . . . .	14
5.4	AF10: Hinzufügen von Programmen . . . . .	15
5.5	AF20: Ändern von Programmen . . . . .	16
5.6	AF30: Setzen von Breakpoints . . . . .	17
5.7	AF40: Hinzufügen von Watch-Expressions . . . . .	18
5.8	AF50: Programme debuggen . . . . .	19
5.9	Erzeugung abstrakter Strukturen im Subpaket Interpreter . . . . .	20
5.9.1	Erzeugung einer abstrakten Repräsentation für Watch-Expressions und bedingte Breakpoints . . . . .	20
5.9.2	Erzeugung des Traces . . . . .	22

<b>6</b>	<b>Abhängigkeitseinteilung mit Blick auf die Implementierung</b>	<b>23</b>
6.1	Abhängigkeiten . . . . .	23
6.2	Implementierung . . . . .	24
<b>7</b>	<b>Formale Spezifikation von WLang und Speicherformaten</b>	<b>25</b>
<b>8</b>	<b>Änderung zum Pflichtenheft</b>	<b>25</b>
<b>9</b>	<b>Anhang</b>	<b>25</b>

# 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 *DDebugger*, 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 Kapselungsprinzip oder das Prinzip der losen Kopplung berücksichtigt. Wie genau die Einhaltung der Prinzipien sichergestellt wird, wird an den jeweiligen Stellen im Dokument erwähnt.

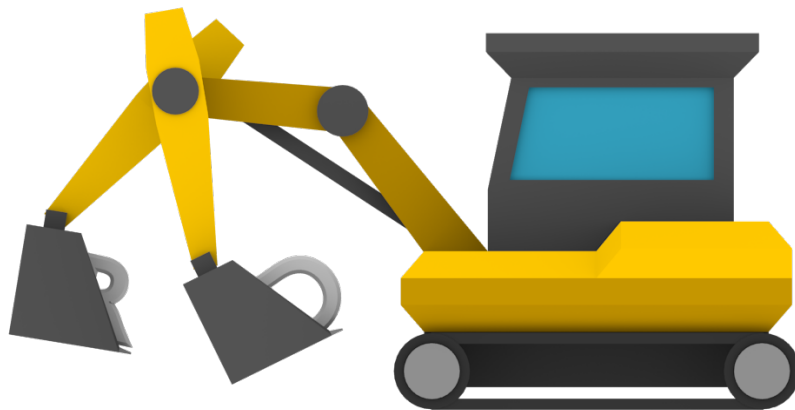


Abbildung 1: Produktlogo

## 2 Paketeinteilung

### 2.1 Übersicht

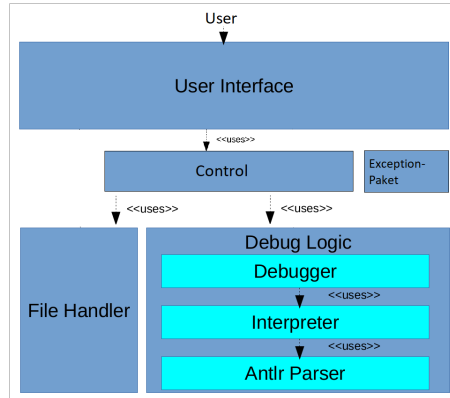


Abbildung 2: Architekturdiagramm

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.

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

**Benutzrelation** 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.

**Benutzrelation** 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* geparsen 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*.

**Benutztrrelation** 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 Benutztrrelation steht.

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

## 3 Beschreibung wichtiger Klassen

Detaillierte Beschreibung aller Klassen. Das beinhaltet (JavaDoc) Beschreibungen zu allen Methoden, 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.

—

### 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 existiert, 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 eingestellte Obergrenze für Funktionsaufrufe wieder.

## LanguageFile

- Klassenbeschreibung:  
Stellt eine Übersetzung der GUI zu einer bestimmten 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 verschiedenen 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

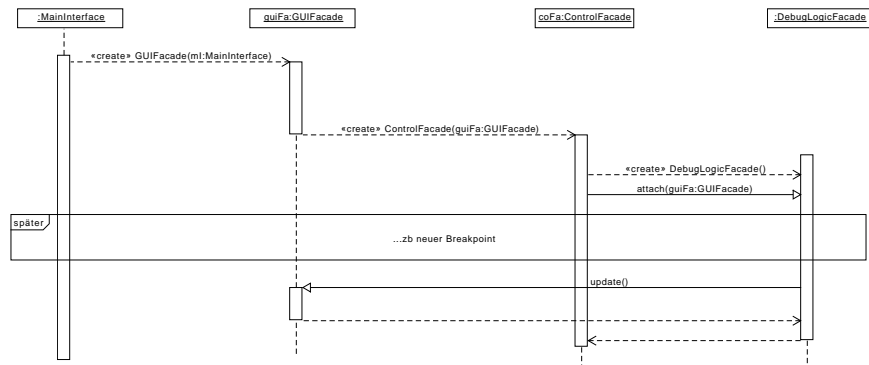


Abbildung 3: Sequenzdiagramm: Erster Programmaufruf

Wird das Produkt gestartet, erstellt die Main-Methode des MainInterface die GUIFacade und übergibt sich selbst. 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

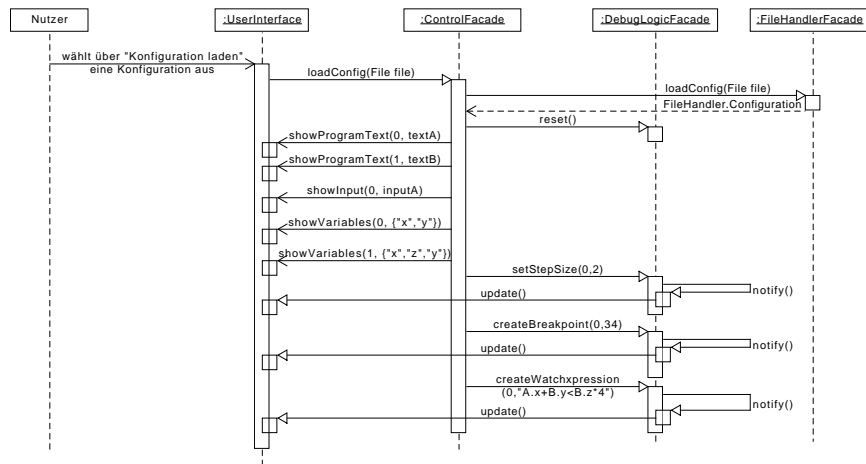


Abbildung 4: Sequenzdiagramm: Laden einer Konfigurationsdatei

Wählt der Benutzer über den Menueintrag „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 Methoden 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

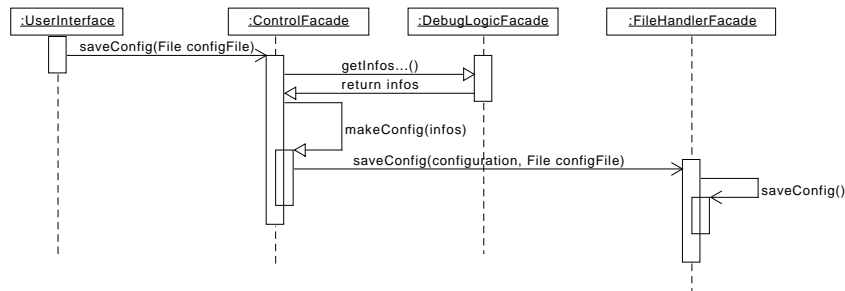


Abbildung 5: Sequenzdiagramm: Speichern einer Konfigurationsdatei

Möchte der Benutzer eine Konfigurationsdatei speichern, 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

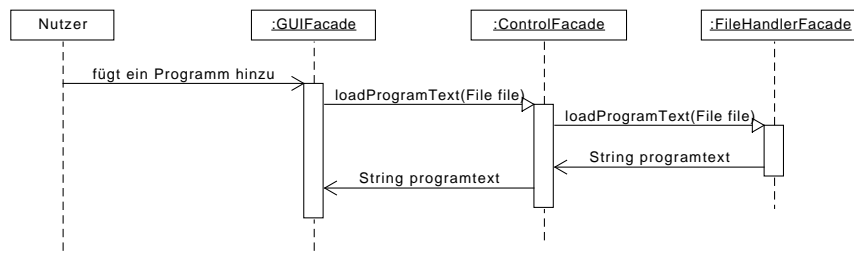


Abbildung 6: Sequenzdiagramm: Hinzufügen von Programmen

AF10 ist falsch

## 5.5 AF20: Ändern von Programmen

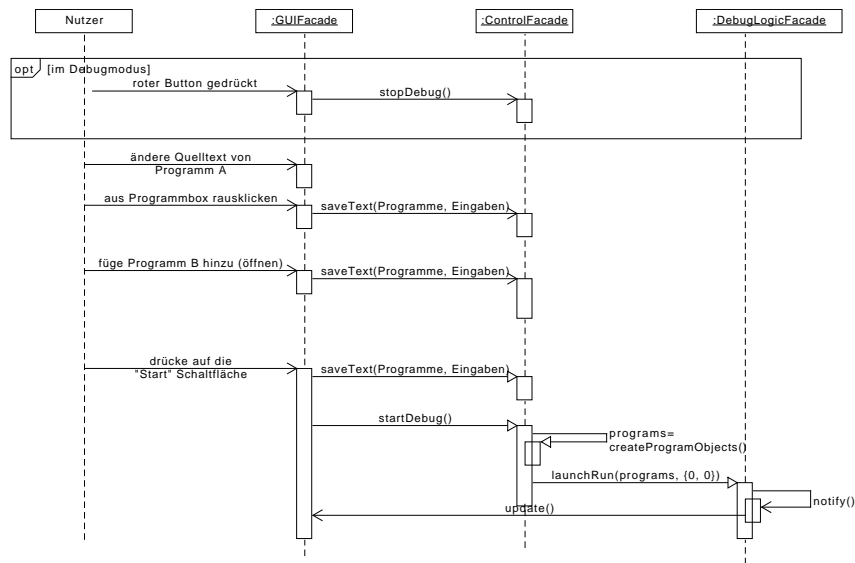


Abbildung 7: Sequenzdiagramm: Ändern von Programmen

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 Eingabevariablen 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

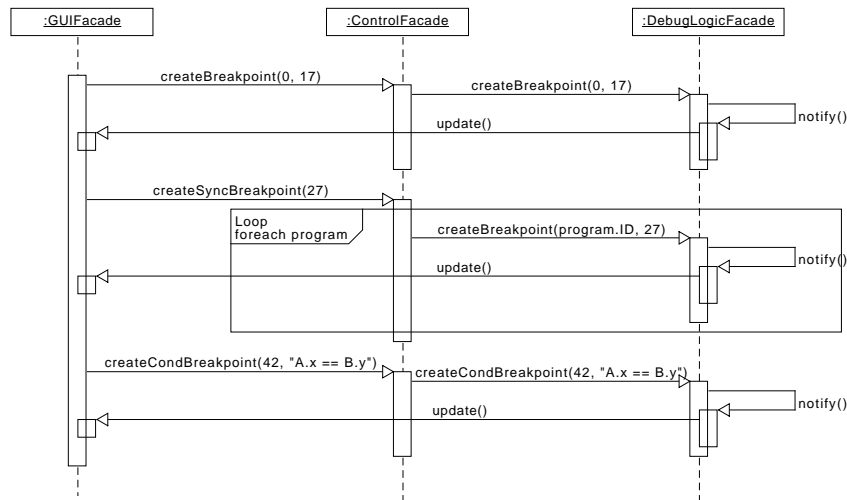


Abbildung 8: Sequenzdiagramm: Setzen von Breakpoints

Setzt der Benutzer einen Breakpoint in eine Zeile in einem Programm, reicht das Main-Interface 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

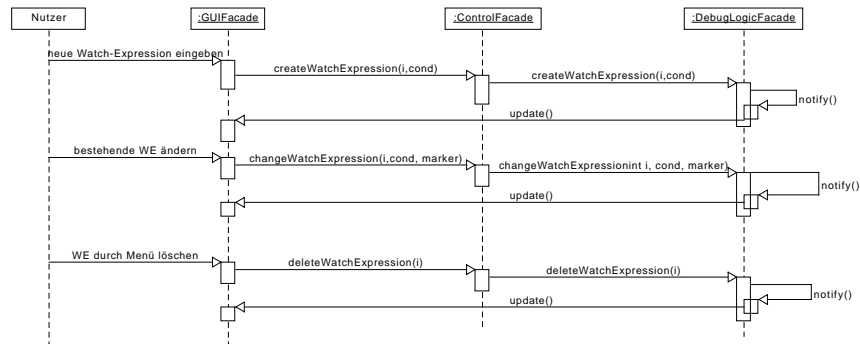


Abbildung 9: Sequenzdiagramm: Hinzufügen von Watch-Expressions

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

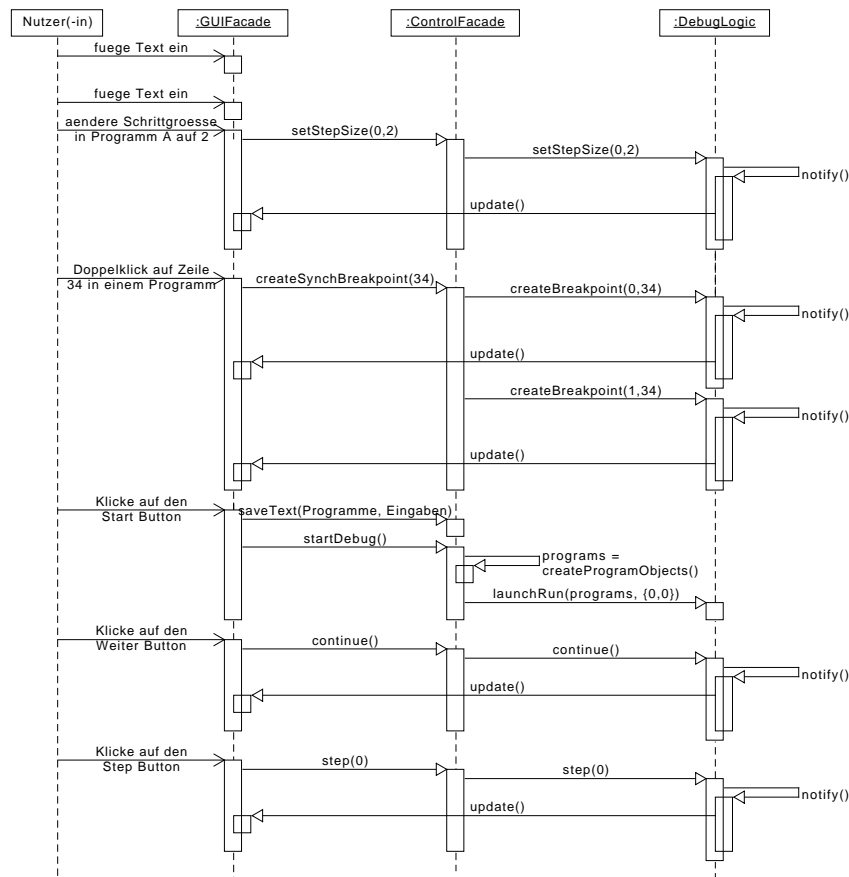


Abbildung 10: Sequenzdiagramm: Debuggen von Programmen

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 hingegen muss aus einer Bedingung bestehen. Der Zusammenhang von Bedingungen und Termen soll zunächst besprochen werden. In Dia-

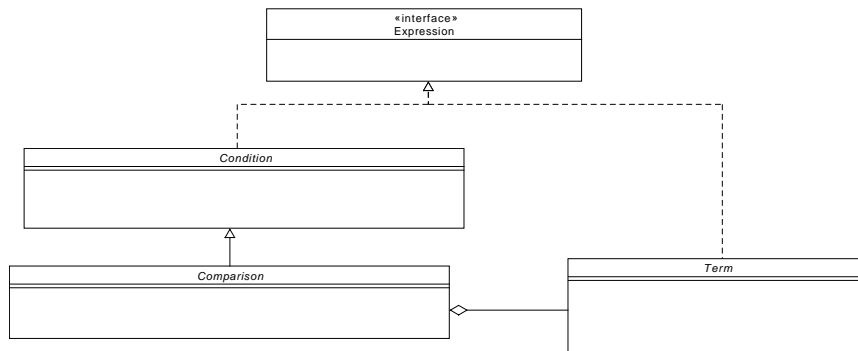


Abbildung 11: Terme und Bedingungen

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 stets 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

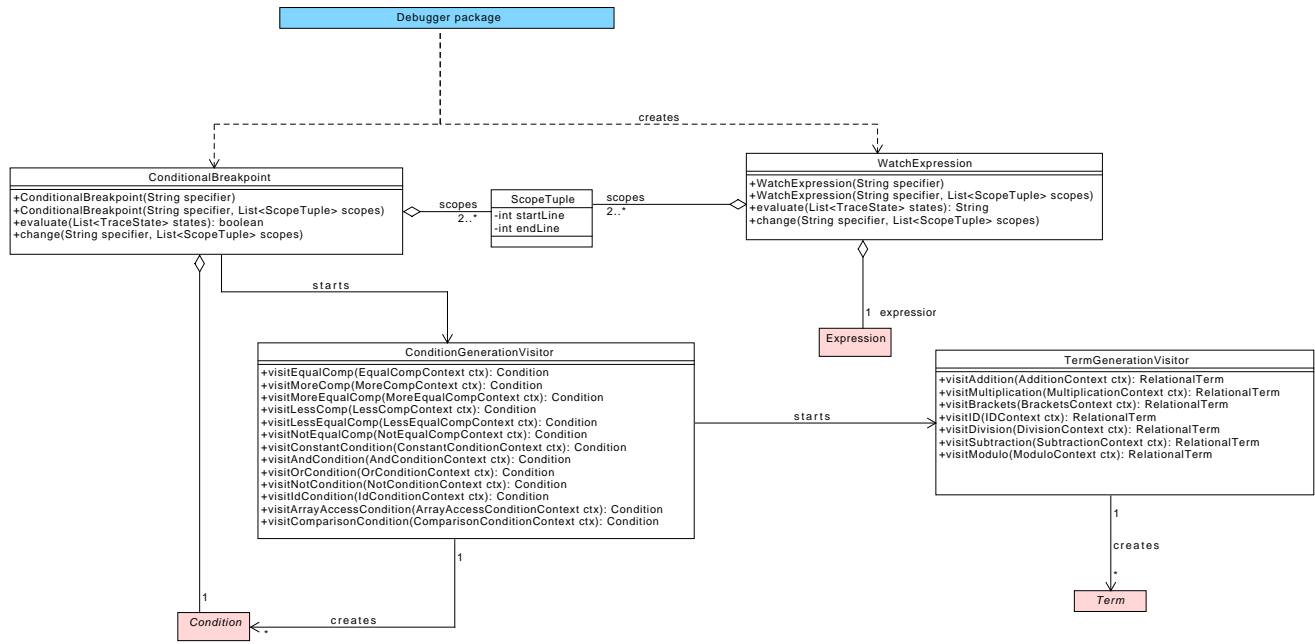


Abbildung 12: Watch-Expressions und Bedingte Breakpoints

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

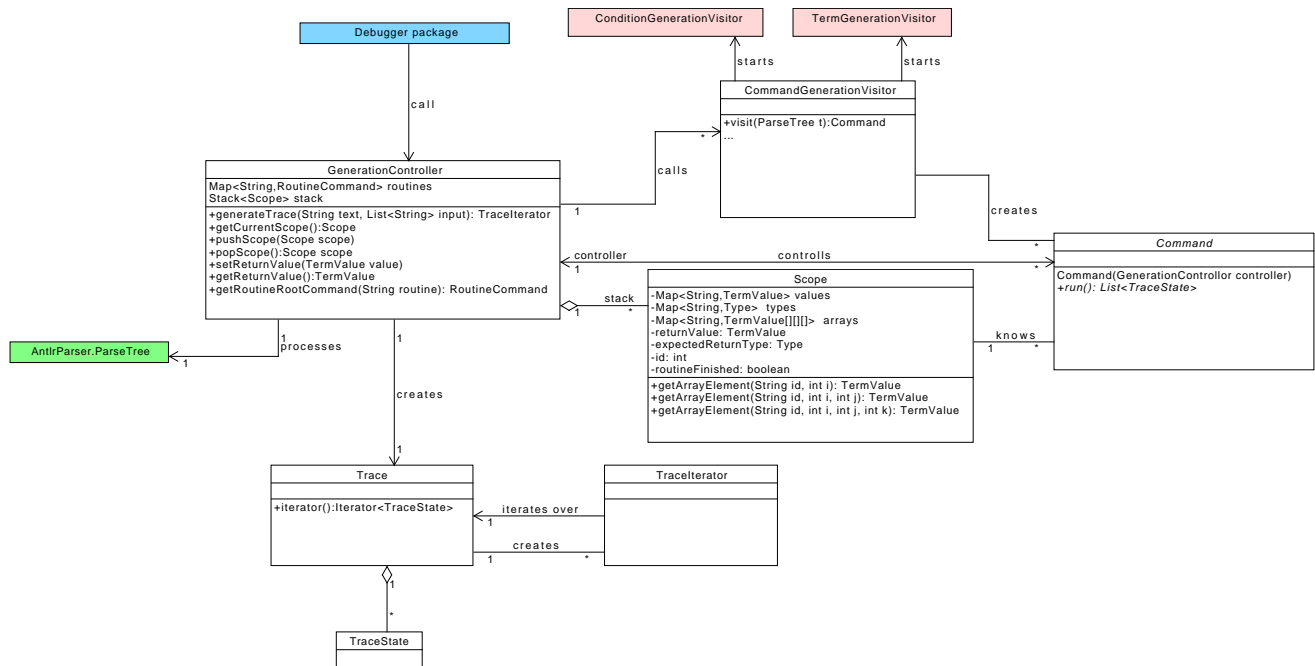


Abbildung 13: An der Tracegenerierung beteiligte Klassen

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 Tracerzeugung. 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

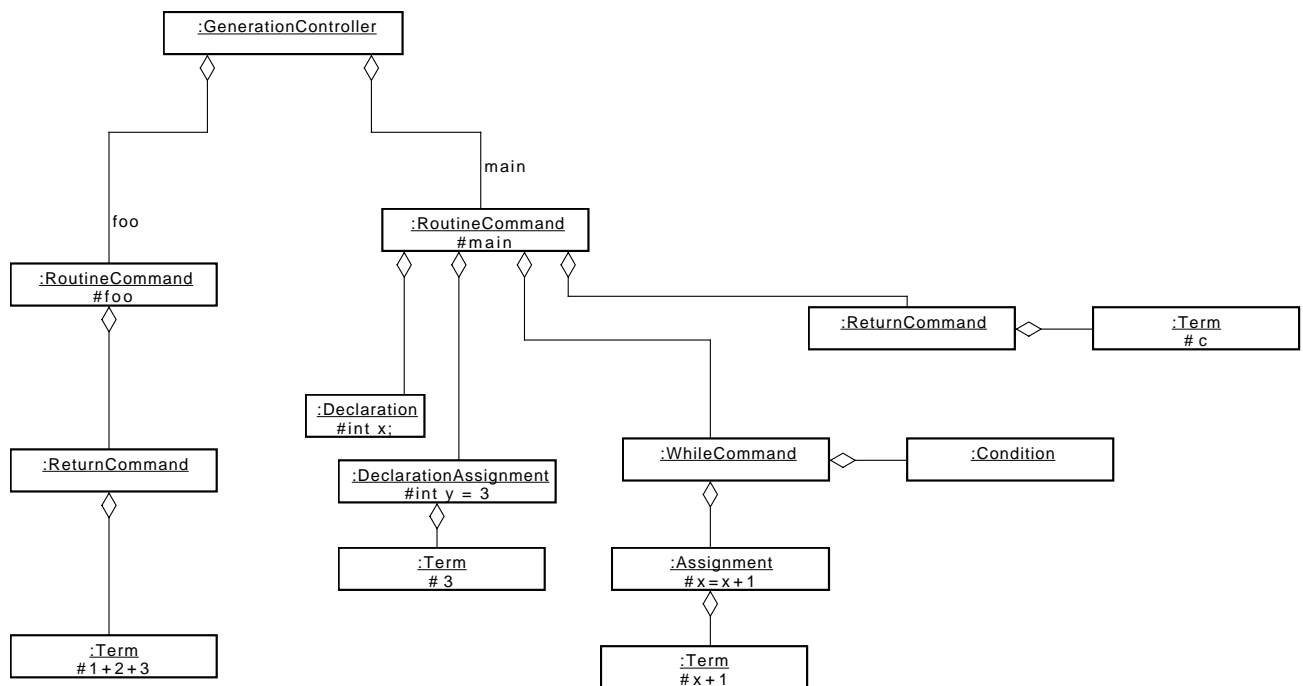


Abbildung 14: Beispiel für einen Ableitungsbaum

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.

## 6 Abhängigkeitseinteilung mit Blick auf die Implementierung

### 6.1 Abhängigkeiten

Das Paketdiagramm besagt, dass *FileHandler* und *DebugLogic* nicht voneinander 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.

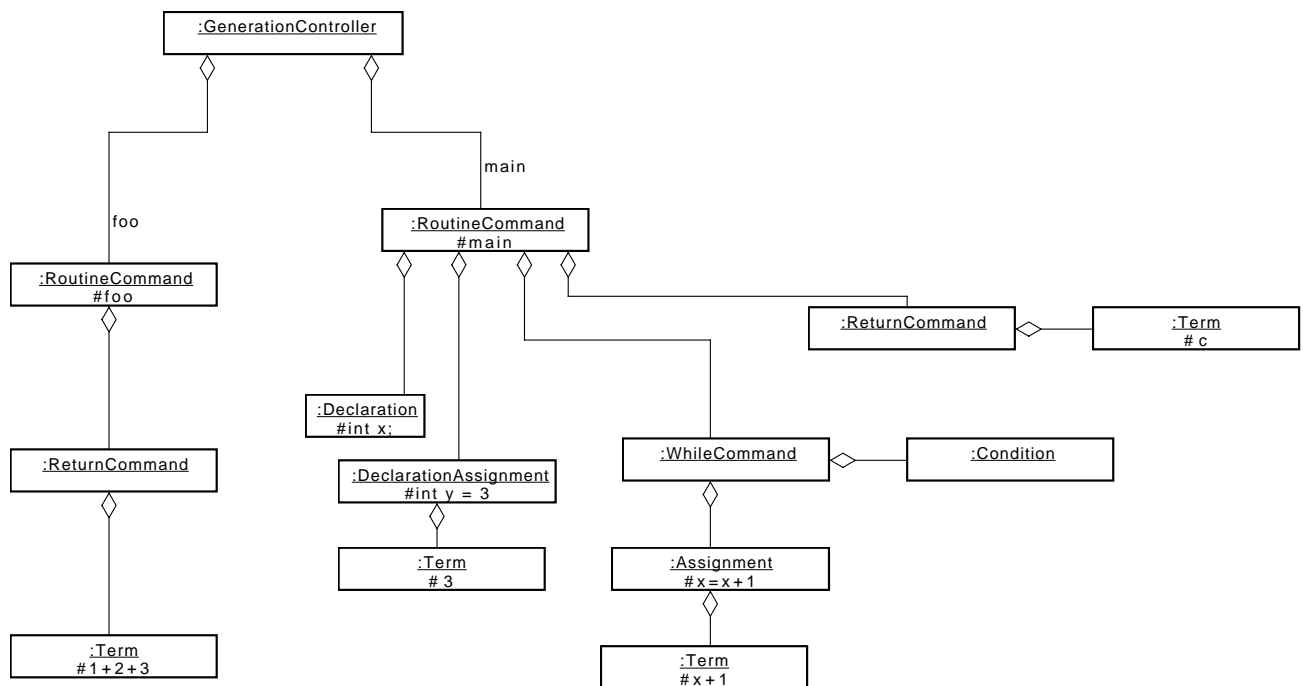


Abbildung 15: Objektstruktur nach der Commanderzeugung

## 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.

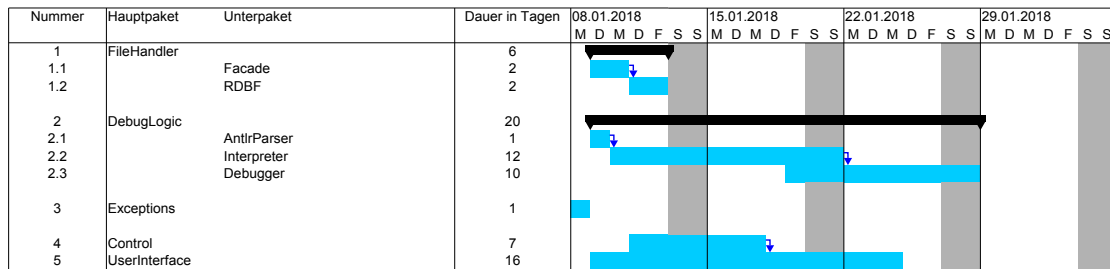


Abbildung 16: Gantt Diagramm: Zeitplanung der Implementierung

## 7 Formale Spezifikation von WLang und Speicherformaten

Speicherformate, Sprachdefinition(formal)

## 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 Klassen dabei bitte mit entsprechenden Pfeilen darstellen, statt nur durch Feldtypen.