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

17. Januar 2018

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*, das im Pflichtenheft definiert wurde. Aufgabe des *DIbuggers* ist es, dem Nutzer zu helfen, mehrere Programme gleichzeitig zu debuggen und interaktiv zu analysieren.
Dabei wurde in den Anforderungskriterien unter anderem festgelgt, dass neben Konzepten eines herkömmlichen Debuggers wie Einzelschritten und Breakpoints, auch zusätzliche Konzepte enthalten sein sollen, die den Umgang mit zwei oder mehr Programmläufen erleichtern. Beispiel für ein solches Konzept sind die Watch-Expressions, bei denen der Nutzer Zusammenhänge zwischen Variablen aus verschiedenen Programmen untersuchen kann. So soll das Finden relationaler Eigenschaften unterstützt werden.
Dieses Dokument erläutert nun, wie diese Anforderungen softwaretechnisch umgesetzt werden sollen. Dazu wird zunächst ein Überblick über den Aufbau des gesamten Projektes gegeben (Kapitel 2 und 3) und dann die einzelnen Pakete, inklusive ihrer wichtigsten Klassen, erklärt (Kapitel 4, 5, 6 und 7). Für einen genaueren Einblick, wie die Pakete zusammenarbeiten, empfiehlt sich Kapitel 8 über charakteristische Abläufe. Kapitel 9 stellt den Zeitplan für die Implementierungsphase (*09.01.-02.02.2018*) vor. Die restlichen Kapitel befassen sich mit Spezifika des Entwurfs.

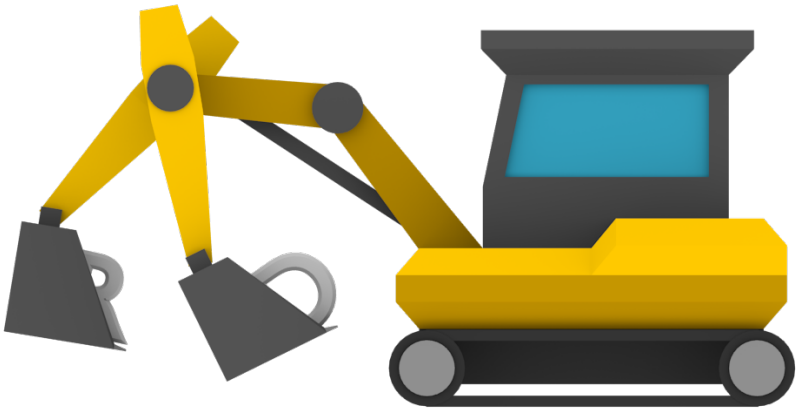

Abbildung 1: Produktlogo

2 Übersicht

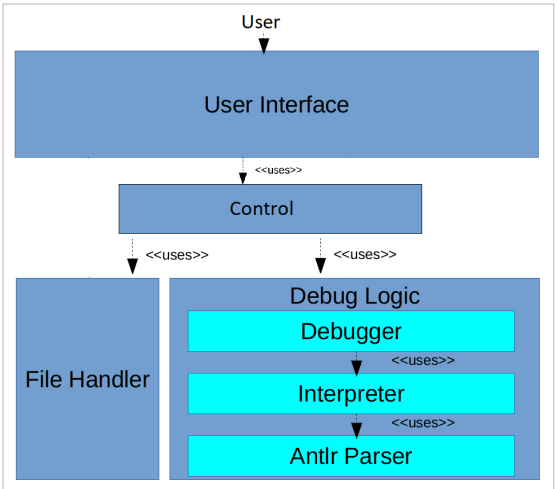

Abbildung 2: Architekturdiagramm

Das Produkt ist aufgeteilt in die vier Pakete *Control*, *UserInterface*, *FileHandler* und *DebugLogic*. Dabei hat jedes Paket seine eigene Aufgabe:

- Das *UserInterface* beinhaltet den Code für die Benutzeroberfläche, also alles, was der Nutzer vom Produkt sieht.
- Die *Control* nimmt die Benutzereingaben entgegen und steuert die Interaktion zwischen dem *UserInterface* und der *DebugLogic*.
- Die *DebugLogic* enthält die Anwendungslogik.
- Der *FileHandler* ist dazu da, Daten zu speichern und zu lesen.

Möchte der Nutzer beispielsweise mit dem Debuggen beginnen und drückt den entsprechenden Knopf auf der Benutzeroberfläche, setzt die *Control* diesen Befehl um, indem sie die notwendigen Schritte der *DebugLogic* ausführt. Will der Nutzer seine Ergebnisse später speichern, drückt er wieder einen Knopf auf der Benutzeroberfläche und der Befehl wird wieder vom *UserInterface* an die *Control* weitergegeben, die dieses Mal zum Speichern den *FileHandler* aufruft. In Abbildung 2 sind die Pakete und die Benutztrelation dargestellt.
Aus der Abbildung wird ebenfalls ersichtlich, dass die *DebugLogic* aus den Unterpaketen *Debugger*, *Interpreter* und *AntlrParser* besteht. Deren Funktionalität wird in Kapitel 7 dargelegt.

3 Global genutzte Entwurfsmuster

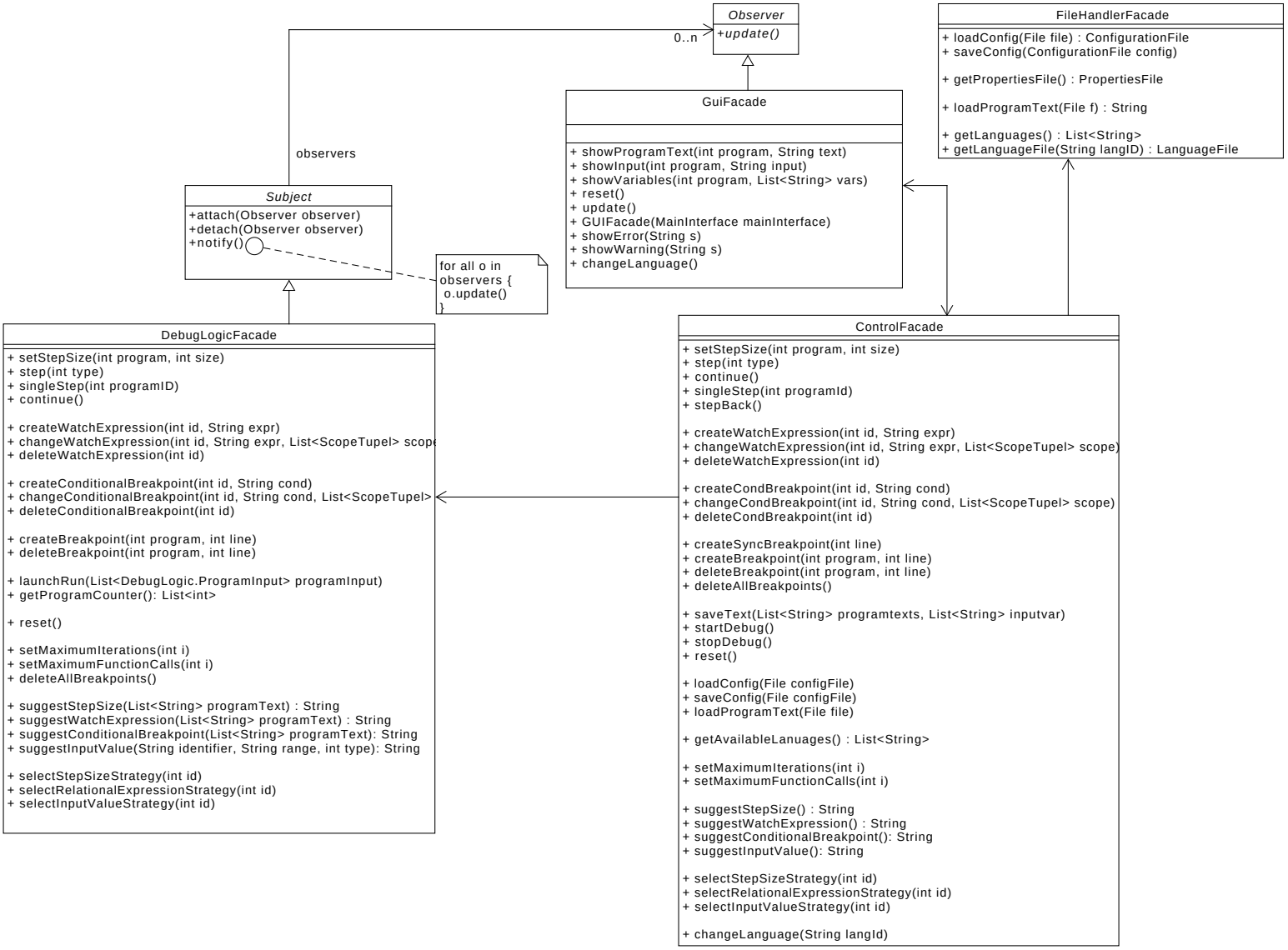

Abbildung 3: Die Schnittstellen der Pakete durch Fassaden dargestellt

Das MVC-Pattern Im Entwurf 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 (siehe Abbildung 3). Die genauen Schnittstellen sind also durch diese Klassen definiert.

Das Observerpattern im MVC Im Rahmen einer MVC-Architektur, ist es notwendig, dass die Viewkomponente eine Zustandsänderung des Modells sofort erfährt und darauf reagieren kann, indem sie sich die von ihr zur Anzeige benötigten Daten holt. Da es der Erweiterbarkeit wegen auch möglich sein soll, einmal mehrere verschiedene dieser beobachtenden Viewkomponenten ein Modell beobachten zu lassen, empfiehlt sich das Entwurfsmuster „Observer“. Die von diesem Pattern zur Verfügung gestellte 1-zu-n Abhängigkeit passt damit hervorragend zur vorliegenden Problemstellung: Aus Gründen der Wiederverwendbarkeit darf das Modell nicht festsetzen, welche und wie viele solcher Beobachter es hat. Die allgemein definierten Schnittstellen *Observer* und *Subject* ermöglichen dies.

Das Fassadenpattern Durch die Einteilung des Systems in einzelne Subsysteme in Form von Paketen, ist es von Nöten, Schnittstellen dieser Subsysteme zu definieren. Diese sollten fest sein und sich bei Änderung der internen Struktur eines Subsystems nicht ändern, sodass das Geheimnisprinzip eingehalten wird. Um zusätzlich die Abhängigkeiten der Pakete zwischeneinander im Sinne einer losen Kopplung gering zu halten, bietet sich das Entwurfsmuster „Facade“ an. Jegliche Kommunikation zwischen den in Abbildung 3 beschriebenen Paketen findet also über Fassaden statt. Das ermöglicht, die Schnittstellen des Paketes an einer Stelle zu definieren, und festgelegte Eintrittspunkte für die Kommunikation zwischen den Paketen zu bieten.

4 Das Paket „User Interface“

4.1 Übersicht

Das Paket *UserInterface* stellt die Möglichkeit zur Kommunikation des Nutzers mit dem Produkt dar. Hierbei dient dieses Paket als View Teil des MVC-Konzepts.
Es wird eine Fassade angeboten, welche es ermöglicht, Variablen, Programmtexte und Eingaben anzuzeigen. Dieses Paket nutzt die Fassade des Paketes *Control* und deren angebotene Schnittstellen, um an Informationen zu gelangen, die der Benutzer anfordert, und die Aktionen des Benutzers zur Ausführung weiterzugeben.
Für die Implementierung der Benutzeroberfläche wird die Bibliothek *javax.swing* benutzt. Durch die Nutzung von Swing besteht der Grundaufbau des *UserInterfaces* aus einem Kompositum.

4.2 Wichtige Elemente des User Interface

GUIFacade 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 *UserInterface* einfach ausgetauscht werden können.
Die Fassade bietet Methoden an, um Programmtexte und Eingabevariablen, sowie Fehlermeldungen oder Warnungen anzeigen zu können, und bietet die Möglichkeit, das *UserInterface* zurückzusetzen.
Außerdem wird die *GUIFacade* beim Start des Produkts im Rahmen des Beobachtermusters bei MVC (siehe Kapitel 3) bei der *DebugLogic* als Beobachter angemeldet und besitzt eine *update()*-Methode, über welche die Änderung der anzuzeigenden Daten gesteuert wird.

MainInterface Das *MainInterface* bildet die Grundlage der Benutzeroberfläche. Diese Klasse enthält ein Liste an *ProgramPanel*s , sowie ein *CommandPanel* und je ein *WatchExpression-* bzw. *CondBreakpointPanel*. Sie ist verantwortlich für das Anzeigen von Menüs und diesen vier Panelarten, sowie für das Benachrichtigen des Benutzers mithilfe von *DebuggerPopUps*.

ProgramPanel 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.

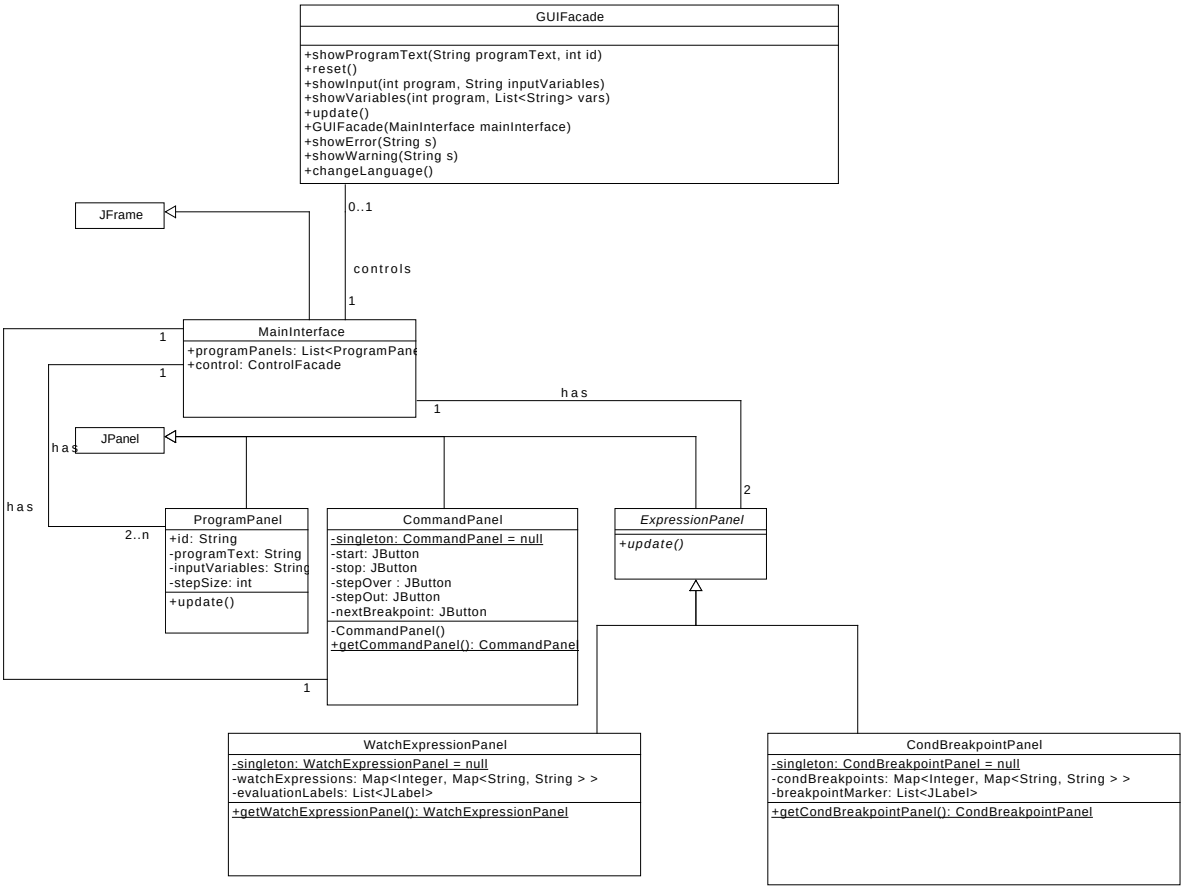

Abbildung 4: Wichtige Anzeigeelemente des *UserInterface*

Als Teil des Beobachter-Entwurfsmusters besitzt ein *ProgramPanel* eine update()-Methode, um bei Änderung der anzuzeigenden Daten benachrichtigt werden zu können.

CommandPanel Ein *CommandPanel* ist der Abschnitt, welcher die Buttons zur Kontrolle des Debugvorgangs anzeigt. Da ein solcher Kommandoblock nur ein mal benötigt wird, ist dieser als Singleton implementiert.

ExpressionPanel 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 (*WatchExpressionPanel* und *CondBreakpointPanel*). Diese werden als Singletons realisiert, um einen globalen Zugriffspunkt auf die Instanzen zu erhalten, und um die alleinige Existenz eines Exemplars pro Klasse zu Sichern.

Als Teil des Beobachter-Entwurfsmusters besitzt ein ExpressionPanel eine *update()*-Methode, um bei Änderung der anzuzeigenden Daten benachrichtigt werden zu können.

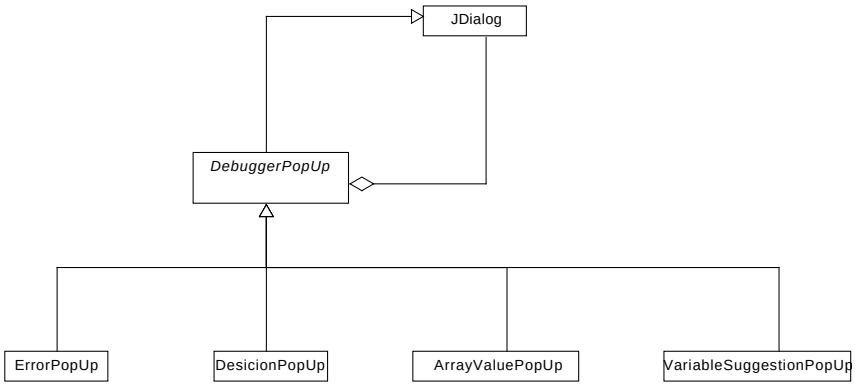

Abbildung 5: Realisierung des Entwurfsmusters Dekorierer in *DebuggerPopUp*

DebuggerPopUp Ein *DebuggerPopUp* ist ein JDialog, welcher auf dieses Produkt ausgelegt ist. Es handelt sich um eine abstrakte Klasse, welche die nicht abstrakten Unterklassen *ErrorPopUp*, *WarningPopUp*, *ArrayValuePopUp* und *VariableSuggestionPopUp* hat. Diese Unterklassen, welche von *DebuggerPopUp* erben, sind darauf ausgelegt Fehlermeldungen, Warnungen oder Variablenwerte anzuzeigen.
Die Klasse *DebuggerPopUp* stellt, wie in Abbildung 5 zu Sehen, ein Dekorierer Entwurfsmuster mit JDialog dar.

5 Das Paket „Control“

5.1 Übersicht

Das Paket *Control* entspricht dem Kontroll-Teil des Architekturstils MVC.
Es dient der Entgegennahme von Benutzerinteraktionen auf dem *UserInterface* und der Steuerung der Interaktion zwischen den Subsystemen *UserInterface* und *DebugLogic*. Schaltflächen, sowie Eingabefelder werden im Paket *UserInterface* mit Präsentationskomponenten wie dem Variableninspektor zusammengefasst.
Die vom Paket bereitgestellten Methoden sind über eine Fassade aufrufbar.
Control benutzt das Paket *Debuglogic*, um Zustandsveränderungen des Datenmodells auszulösen. 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 das Durchführen eines Einzelschrittes. Zum Ausführen von Speicher- und Ladeaufträgen benutzt *Control* das Paket *FileHandler*.

5.2 Wichtige Elemente der Control

Um die Funktionalität des Paketes aufzuteilen, wurden vier Klassen entworfen, die jeweils unterschiedliche Zuständigkeit besitzen.

ControlFacade *ControlFacade* bietet eine Schnittstelle zum Paket *Control*.

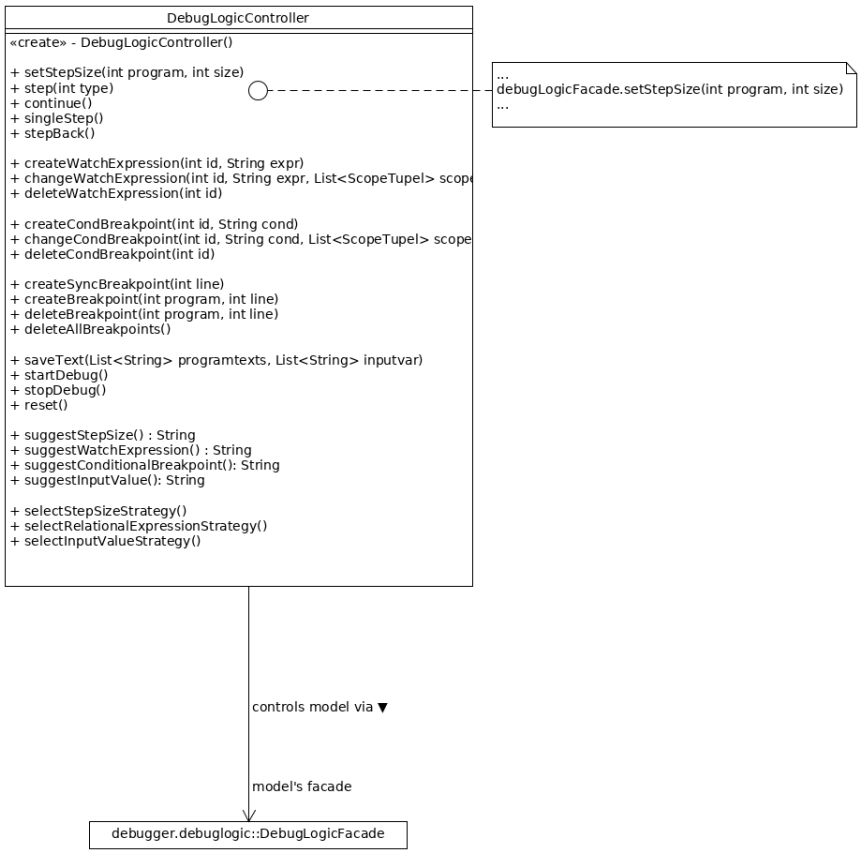

Abbildung 6: Die Klasse DebugLogicController

DebugLogicController *DebugLogicController* ist für die Modifikation der Daten des Modells und das Aufrufen von Debugmechanismen zuständig.

Über die bereitgestellten Methoden wird „DebugLogic“ dazu aufgefordert, Schritte durchzuführen, WatchExpressions oder Haltepunkte zu erstellen und zu entfernen oder Vorschläge zu generieren. Außerdem werden die Zustände „Debugmodus“ und „Editiermodus“ in *Control* implementiert, folglich bietet diese Klasse zwei Methoden zum Wechseln des aktuellen Modus.

ExceptionHandler *ExceptionHandler* dient zur Behandlung von beim Aufruf der Methoden aus DebugLogic oder FileHandler ausgelösten Ausnahmen.
Insbesondere kann die Präsentation dazu aufgefordert werden, Benutzer der Anwendung entsprechend über eine Fehlersituation zu informieren.

FileHandlerInteractor *FileHandlerInteractor* ist verantwortlich für das Speichern und Laden von Konfigurationsdateien, sowie das Verwalten einer Sprachdatei, die Anzeigetexte für die Präsentation des *UserInterface* enthält.
Dementsprechend besitzt die Klasse Methoden, durch welche das Wiederherstellen gespeicherter Sitzungen, oder das Laden eines Programmtextes ermöglicht wird. Um dies zu ermöglichen, werden sowohl das Modell modifiziert, als auch die Präsentation (direkt) aufgefordert die gebotene Darstellung zu verändern.

6 Das Paket „File Handler“

6.1 Übersicht

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 der *FileHandler* eine Konfigurations- oder Sprachdatei, welche auf dem Dateisystem gespeichert ist, in eine virtuelle Datei um. Diese besteht aus einer Klassenstruktur, welche äquivalent zur Definition des Speicherformats (siehe Anhang 12) ist, also aus Zuweisungen und Blöcken. Weiter erzeugt der *FileHandler* Objekte der Konfigurations-, Sprach- und Einstellungsdateien und kann diese nach außen weitergeben. Einstellungsdateien werden mithilfe von Java's Properties Klasse realisiert. Da der FileHandler an unterster Stelle der Benutztrelation steht, benötigt er keine Schnittstellen anderer Programmpakete, lediglich werden mindestens jeweils eine Klasse zum Lesen und Schreiben von Konfigurations-, Einstellungs- und Sprachdateien benötigt. Dafür werden Klassen für das interne Relational Debugger Format (RDBF) beziehungsweise Java Properties bereitgestellt. Jedoch kann das Paket um andere Formate, wie „XML“ oder „Json“, erweitert werden. Weiter werden eine Fassadenklasse und drei Klassen für das Speichern von Produkteinstellungen, Sprachen (Übersetzungen der GUI) und Konfigurationen angeboten und nach außen weitergegeben.

6.2 Wichtige Elemente des File Handlers

6.2.1 Unterpaket FileHandler.Facade

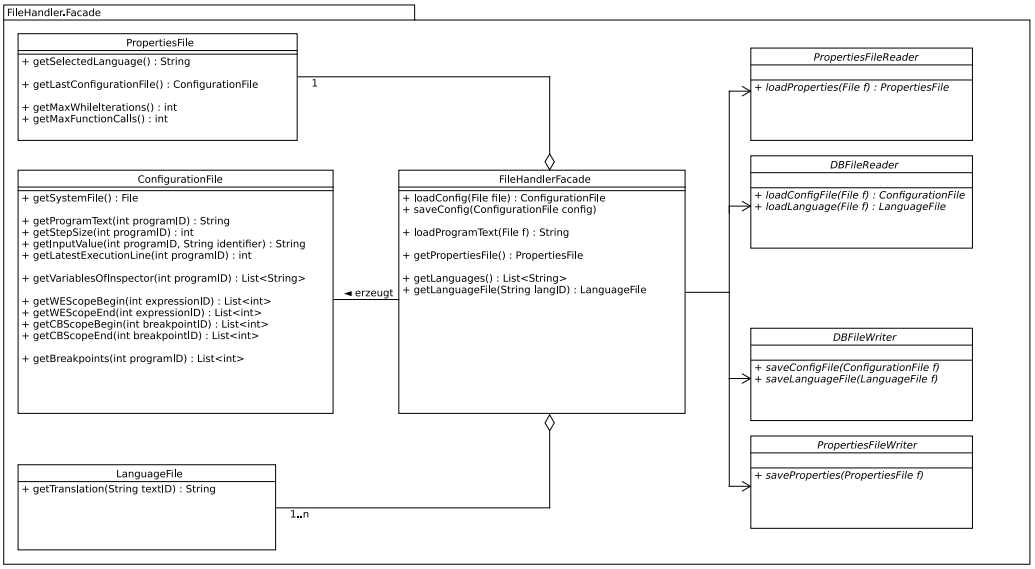

Abbildung 7: Die Schnittstellen Fassade des File Handlers

FileHandlerFacade Diese Klasse speichert alle verfügbaren Sprachen und hilft bei der Erzeugung von Konfigurations- und Einstellungsdateien. Die Klasse beinhaltet alle aktuellen Sprachdateien. Sie bietet Methoden zum Lesen und Speichern von Konfigurationsdateien an. Dabei können auch Textdateien, die einen Programmtext enthalten gelesen werden. Weiter kann die aktuelle Einstellungsdatei abgerufen werden, sowie alle verfügbaren Sprachen und eine bestimmte Sprachdatei.

ConfigurationFile Diese Klasse speichert eine Konfiguration des Debuggers. Dabei enthält diese Objekte für den Speicherort der Konfiguration, sowie Programmtexte, Schrittgrößen und Eingabewerte. Weiter wird der Status des Debuggers gespeichert, also die derzeitigen Programmzähler, Watch-Expressions, Breakpoints (einfach und bedingt), sowie der von die von der GUI abhängigen Konfiguration der Variableninspektors. Methoden zum Lesen und Schreiben dieser Attribute werden zur Verfügung gestellt. Dabei wird darauf geachtet, dass nicht zuviel von der internen Struktur der Klasse nach außen gegeben wird.

PropertiesFile Diese Klasse speichert die Einstellungen des Debuggers. Darunter fallen Daten wie die ausgewählte Sprache, obere Grenzen für Iterationen und die zuletzt verwendete Konfigurationsdatei, falls diese existiert. Methoden für diese Attribute werden durch einfache Getter realisiert.

LanguageFile Diese Klasse stellt die Übersetzung in eine bestimmte Sprache zu den von einem User Interface angezeigten Texten bereit. Hierbei zählen nicht nur Anzeigeelemente, sondern auch geworfene Fehlermeldungen der unteren Schichten. Somit wird eine Methode zum transformieren einer Identität in einen Text bereitgestellt.

FileReader / Writer Klassen Zum Lesen und Schreiben der Konfigurations-, Einstellungs- und Sprachdateien werden vier abstrakte Klassen *DB(DIbugger)FileReader*, *PropertiesFileReader*, *DBFileWriter* und *PropertiesFileWriter* angeboten. Diese stehen jeweils für die Basis einer Strategie und bieten zusammen abstrakte Methoden zum Lesen und Schreiben der oben genannten Dateien an.

6.2.2 Unterpaket FileHandler.RDBF

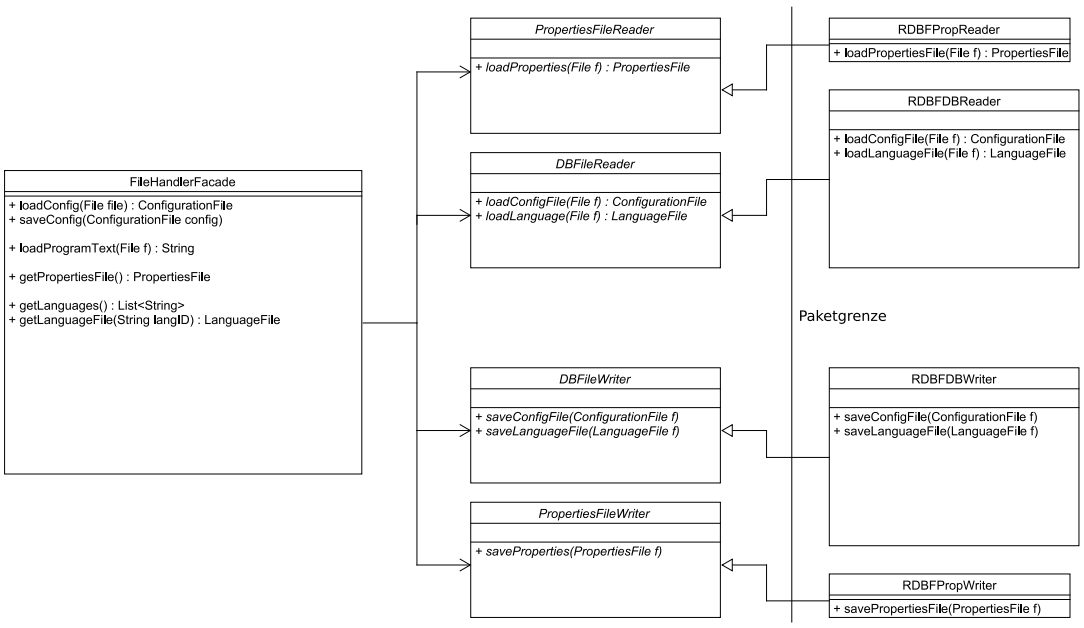

Abbildung 8: Das Strategie Entwurfsmuster im FileHandler

Dieses Paket bietet als untergeordnete konkrete Strategie für das Lesen und Schreiben der Konfigurations-, Einstellungs, und Sprachdateien formatiert im Relational Debugger

Format (RDBF) Klassenstrukturen und Methoden an. Das Speicherformat RDBF kann im Anhang 12.3 eingesehen werden.

Konkrete Strategien zu den oben gennanten FileReader / Writer Klassen Als konkerete Implementierungen der DBFileReader, -Writer und PropertiesFileReader, -Writer Klassen werden vier Unterklassen angeboten. Nennenswert tragen sie den selben Klassennamen wie obige, jedoch mit Präfix RDBF. Die Klassen stellen jeweils eine Implementierung zum Lesen und Schreiben von Konfigurations- und Sprachdateien im RDBF Format, sowie Einstellungsdateien im Java Properties Format zur Verfügung. Dabei stellen sie eine konkrete Strategie dar, um somit Austauschbar und Erweiterbar für weitere Strategien wie XML und Json zu sein.

RDBFReader Diese Klasse stellt Funktionalität zum Lesen von Dateien im RDBF Format bereit. Dabei wird eine Methode zum Einlesen einer RDBF Datei bereitgestellt, die ein RDBFFile zurückgibt.

RDBFWriter Diese Klasse stellt Funktionalität zum Speichern von RDBFFile Objekten im RDBF Format bereit. Es wird nur eine Methode angeboten, welche eine virtuelle RDBF Datei auf das Dateisystem des Nutzers schreibt.

RDBFParser Diese Klasse existiert als Singleton und übernimmt die Funktionalität, eingelesene Zeilen zu analysieren und zu interpretieren, um diese dann final zu Objekten zusammenzufassen. Sie bietet Methoden zum herausfinden des Typs einer Zeile (Zuweisung, Block, Textblock), sowie eines Zuweisungswertes an. Damit können weitere Methoden benutzt werden, welche aus einer Zuweisung die Variable und den Wert herausfinden, bei einem Block den Blocknamen und den Wert einer Zuweisung in einen primitiven Datentyp von JAVA umwandelt.

RDBFAdditions Abstrakte Klasse zum repräsentieren von RDBFFile und RDBFBlock, da diese sich nur in wenigen Punkten unterscheiden. Hierfür werden eine Liste an Blöcken (RDBFBlock) und Daten (RDBFData) gespeichert und Getter-Methoden bereitgestellt. Diese können mithilfe von Blocknamen und Variablennamen Teillisten der obigen Attribute erstellen und zurückgeben.

RDBFFile Repräsentiert eine RDBF Datei als schnell zugreifbare Objektstruktur und stützt sich dabei auf die Implementierung von RDBFAdditions. Sie speichert zusätzlich nur den Dateipfad der darzustellenden Datei.

RDBFBlock Stellt einen in der RDBF Grammatik definierten Block dar. Dabei werden die Methoden von RDBFAdditions geerbt. Die Klasse speichert zusätzlich zu beiden Listen nur den Namen des Blocks.

RDBFData Repräsentiert eine Zuweisung der RDBF Grammatik als Datenstruktur. Hierbei enthält diese einen Variablennamen, Wert und zusätzlich den Typ des Wertes. Einfache Getter und Setter Methoden für die Attribute existieren.

6.2.3 Unterpaket FileHandler.Exceptions

Interface: FileHandlerException Dieses Interface sorgt dafür, dass alle Klassen im Unterpaket Exceptions des FileHandlers die benötigten Methoden für eine *FileHanderException* haben.
Alle Klassen in diesem Unterpaket implementieren die Schnittstelle *FileHandlerException*. Es existieren die Klassen LanguageNotFounrException, ParseBlockException, ParseAssignmentException und InvalidLineTypeException. Die LanguageNotFoundException tritt auf, falls eine nicht existente Sprache angefragt wird. Alle anderern Exceptions behandeln das Auftreten von Fehlern während dem Auslesen von einzelnen Zeilen. Dabei werden Methodenaufrufe auf falsche Zeilen, sowie syntaktisch falsche zeilen berügsichtigt.

7 Das Paket „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 im Anhang 12 gegebenen Grammatiken.
Darauf aufbauend in der mittleren Schicht findet sich das Subpaket *DebugLogic.Interpreter*, das die Aufgabe hat, diese Ableitungsbäume durch Interpretieren in eine abstrakte und leicht handhabbare Form zu bringen. In der obersten Schicht ist das Subpaket *DebugLogic.Debugger* angesiedelt. Dieses nutzt die abstrakten Repräsentationen und führt den eigentlichen Debugprozess darauf aus.

7.1 Debugger

7.1.1 Übersicht

Der Debugger nutzt die vom Subpaket *DebugLogic.Interpreter* erzeugten Informationen, um Watch-Expressions und bedingte Breakpoints auszuwerten, sowie die üblichen Debugmechanismen zu steuern. Als oberste Schicht des Paketes *DebugLogic* stellt dieses Subpaket die gleichen Schnittstellen wie die DebugLogic bereit. Diese können in Kapitel 3 der entsprechenden Fassadenklasse entnommen werden.
Um die üblichen Debugmechanismen wie Schritte und Weiter durchführen zu können, nutzt dieses Subpaket den vom Subpaket *DebugLogic.Interpreter* bereitgestellten TraceIterator. Um WatchExpressions und bedingte Breakpoints auszuwerten und zu repräsentieren, nutzt dieses Subpaket die vom Subpaket *DebugLogic.Interpreter* bereitgestellte abstrakte Repräsentationen. Die Klassenstruktur ist in Abbildung 9 dargestellt. Die Aktionen des Paketes sind von Anfragen gesteuert, das bedeutet, es geschieht nichts, ohne dass von aussen über die Klasse *DebugLogicFacade* eine entprechende Anweisung als Methodenaufruf kommt. Deshalb hat diese Fassade auch für alle Debugmechanismen entsprechende Methoden, zum Beispiel zur Ausführung von Schritten in den Programmen.
Bei jeder Änderung des Zustandes des Debuggers wird nach dem Beobachtermuster die Methode *notify()* der Fassade aufgerufen und so alle Beobachter benachrichtigt. Je nach Methodenaufruf in der *DebugLogicFacade* wird die Anfrage an die *DebugControl* oder an eine konkrete Ausprägung eines Vorschlagsinterfaces (zum Beispiel *StepsizeSuggestion*) weitergeleitet. Da man das Verfahren, einen solchen Vorschlag zu generieren, beliebig komplex gestalten kann, sollen die verwendeten Algorithmen austauschbar und variabel gehalten werden. Dafür bietet sich das Entwurfsmuster „Strategie“ besonders an. Im Entwurf wird es durch die Interfaces *StepSizeSuggestion, RelationalSuggestion* und *InputValueSuggestion* realisiert.

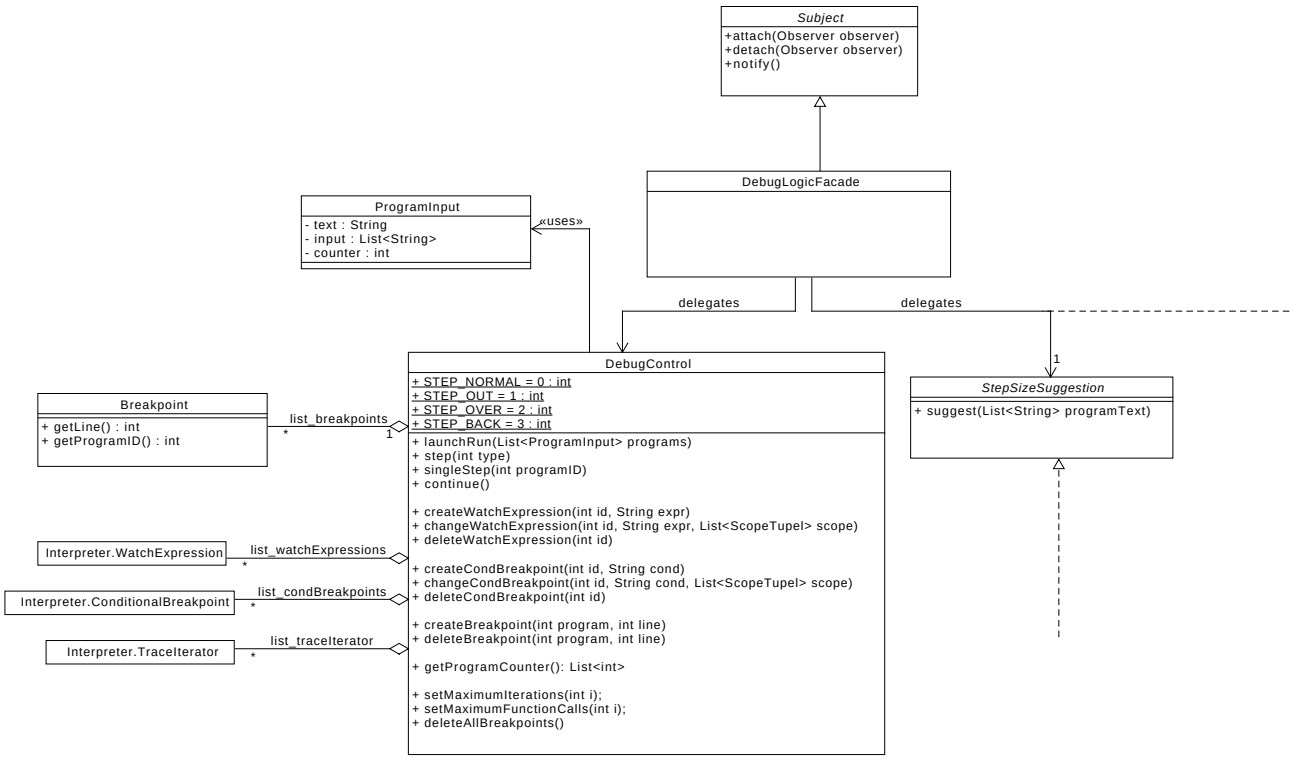

Abbildung 9: Die Klassen des Paketes *DebugLogic.Debugger*

Die konkreten Debugmechanismen werden in der Klasse *DebugControl* gesteuert. So verwaltet diese etwa die vom Paket *DebugLogic.Interpreter* bereitgestellten *WatchExpression*-Instanzen oder die *TraceIterator*-Objekte, um so Schritte auszuführen.

7.1.2 Wichtige Elemente des Debuggers

DebugControl Die Klasse *DebugControl* koordiniert den Aufruf der Debugmechanismen, wie Start, Schritte, Einzelschritte und Weiter, sowie das Erstellen, Ändern und Löschen von (bedingten) Breakpoints und Watch-Expressions.

ProgramInput Kapselt die Informationen zu einem zu debuggenden Programm. Dazu gehören Programmtext, Eingabevariablen und Identifikationsnummer.

Breakpoint Kapselt die Informationen eines Breakpoints in einer Zeile eines Programms.

StepsizeSuggestion Stellt das Grundgerüst einer Klasse dar (abstrakte Klasse), welche die Größe der Steps anhand der Programmtexte vorschlägt. Durch die Nutzung eines Strategiemusters wird das Erweitern des Produkts durch weitere Arten Vorschläge zu generieren erleichtert.

InputValueSuggestion Stellt das Grundgerüst einer Klasse dar (abstrakte Klasse), welche eine Variable eines bestimmtem Typs innerhalb eines Bereichs vorschlägt. Auch hier wird die Erweiterbarkeit des Produkts durch die Nutzung eines Strategiemusters erleichtert.

RelationalSuggestion Stellt das Grundgerüst einer Klasse dar (abstrakte Klasse), welche bedingte Breakpoints oder WatchExpressions vorschlägt. Hier wird ebenfalls durch Strategiemuster die Erweiterbarkeit des Produkts um weitere Arten der Vorschlagsgenerierung erleichtert.

7.2 Interpreter

7.2.1 Übersicht

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 diese und stellt sie abstrakt dar. Innerhalb dieses Paketes wird auch auf semantische Fehler geprüft, etwa das Fehlen eines return-Statements.
Dieses Unterpaket benutzt das Unterpaket *DebugLogic.AntlrParser*, um damit aus den reinen Zeichenketten einen Syntaxbaum gemäß der im Anhang 12 gegebenen Grammatik für die Sprache WLang erzeugen zu lassen. Der Vorgang der Erzeugung des oben erwähnten Pfades wird genauer in 23 erklärt.

7.2.2 Wichtige Elemente des Interpreters

GenerationController Steuert die Erzeugung eines kompletten Programmverlaufs (Trace), siehe Kapitel 8.

CommandGenerationVisitor Verwendet das Visitor Entwurfsmuster, um über einen Ableitungsbaum nach der in Kapitel 10 gegebenen WLang-Grammatik zu gehen und dabei eine Menge von Commands zu erzeugen.
Um die Term- und Befehlsstrukturen zu erzeugen, wird hier das Entwurfsmuster *Visitor* verwendet. Zum einen ermöglicht es, die von Antlr generierten *ParseTree*-Objekte mit den von Antlr vorgegebenen Schnittstellen zu durchlaufen. Zum anderen bietet dieses Pattern die Möglichkeit, beliebige Operationen auf den Ableitungsbäumen auszuführen, ohne die von Antlr generierten Klassenstrukturen zu verändern. Die Nutzung dieses Patterns ist daher beim Umgang mit Antlr beinahe unumgänglich.

Scope Kapselt aktuelle Variablenbelegung innerhalb eines Funktionsaufrufs.

Command Stellt einen ausführbaren Befehl dar, der die aktuellen Variablenzustände gemäß seiner Semantik verändern kann.
Für diese Klasse und ihre Unterklassen wird ein „Kompositum“ verwendet. Ein Auszug davon ist in Abbildung 10 zu sehen. Zunächst gibt es elementare Befehle, wie etwa eine einfache Zuweisung. Es gibt jedoch auch Befehle, wie etwa while- und if- Befehle, die mehrere Unterbefehle haben, und entscheiden müssen, ob und wie oft diese ausgeführt werden. Sie stellen also in diesem Fall Befehlskompositionen dar. Ein großer Vorteil dieser Modellierung besteht in der Nutzung der Methode *run()*. Der Aufrufer eines Befehls muss nicht wissen, ob es sich um einen elementaren Zuweisungs- oder Deklarationsbefehl oder um einen komplexen zusammengesetzten Befehl wie etwa eine Schleife handelt. Ihn interessiert nur die während der Ausführung des Befehls angenommenen Zustände. Die Befehlsfunktionalität wird so weggekapselt.
Einen Befehl als Objekt zu kapseln, um diesen in Warteschlangen zu fügen, aufzubewahren oder Empfänger damit zu parametrisieren ist der Zweck des Entwurfsmusters „Command“, das hier auch verwendet ist. Da die Befehle zunächst alle erzeugt und dann später ausgeführt werden. Die Ausführung der Befehle ist entsprechend komplex, da dabei etwa Typprüfungen stattfinden müssen (Näheres dazu ist in Kapitel 8 zu finden), weswegen es sich auch empfiehlt, diese Arbeit zu kapseln.

Trace Stellt den kompletten Programmverlauf eines WLang-Programms dar.

TraceState Stellt einen Zustand während des Programmverlaufs eines WLang-Programms dar.

Der Traceiterator Wir betrachten Abbildung 11. Grundkonzept des Debuggens ist die folgende Idee: Zunächst wird die komplette Ausführung eines Programmlaufes berech-

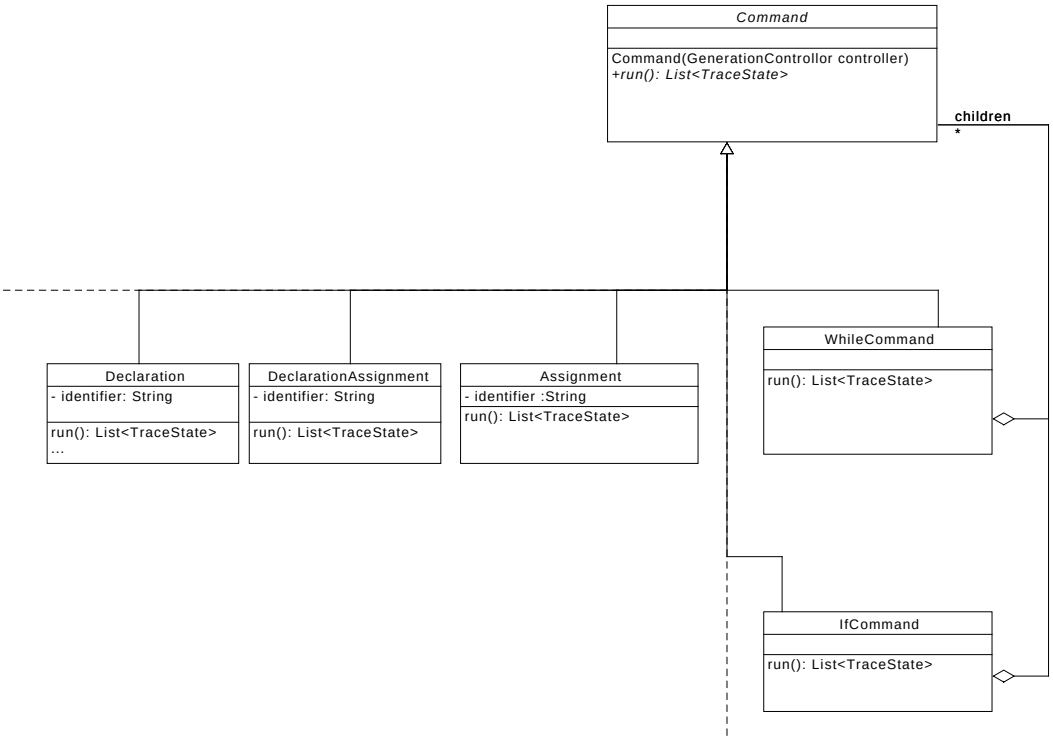

Abbildung 10: Die Klasse *Command* als Kompositum

net und dabei nach jedem ausgeführten Befehl der Zustand der Variablen gespeichert. Die Aggregation all dieser als *TraceState*-Instanzen dargestellten Zustände ist ein sogenanntes *Trace*-Objekt. Um es dem Paket *DebugLogic.Debugger* zu ermöglichen, ohne Kenntnis der konkreten Darstellung dieses Traces über die *TraceState*-Objekte zu iterieren, wird lediglich ein *TraceIterator*-Objekt nach außen gegeben. Um dabei die volle von Java angebotene Funktionalität zu nutzen, werden die entsprechenden Java Schnittstellen implementiert.

Term Stellt eine Abstrakte Repräsentation eines arithmetischen bzw. logischen Terms dar.
Die Struktur eines Termes kann wie folgt beschrieben werden:
Konstanten und Variablen sind Terme. Sind A und B Terme und σ ein zweistelliger Operator, dann ist auch $A\sigma B$ ein Term. Ist ∇ ein einstelliger Operator, dann ist auch ∇A ein Term. Diese induktiv aufgebaute Struktur der Terme lässt sich in naheliegender Weise in eine Klassenstruktur übertragen, die in Abbildung 12 dargestellt ist. Hier wird das Entwurfsmuster „Kompositum“ verwendet. Der Vorteil besteht in der einfachen und einheitlichen Nutzung der Funktion *evaluate()*. Diese gibt eine Instanz vom Typ *Term-*

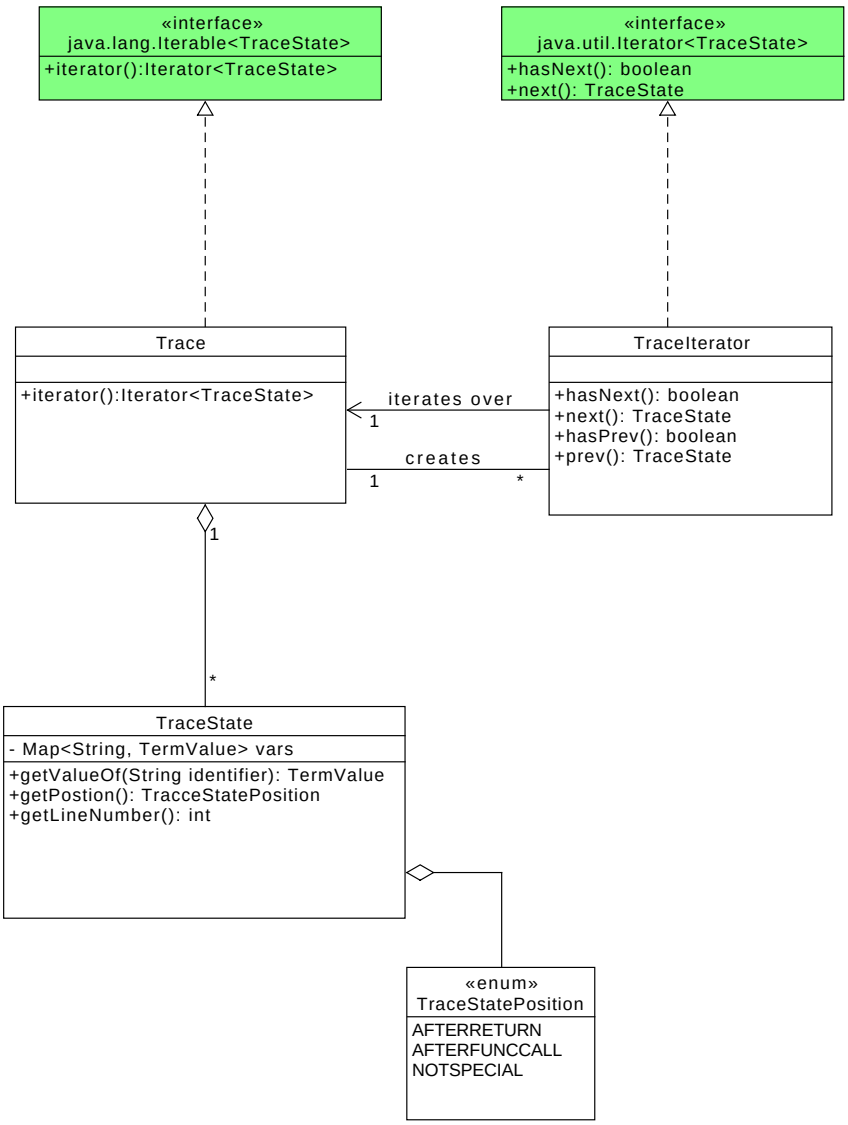

Abbildung 11: Der Traceiterator

Value zurück. Die vollständige Funktionalität des vorliegenden Typsystems ist in den Implementierungen dieser Schnittstelle gekapselt. Einen Spezialfall von Termen stellen *BinaryCondition* und *Comparison* dar. Diese werten sich per Definition zu booleschen Werten aus. Das Liskovsche Substitutionsprinzip erlaubt Kovarianz in den Ausgabeparametern, weswegen es möglich ist, diesen Subklassen von *Term* in den *evaluate()*-Methoden den Rückgabetyp *BooleanValue* zu geben, eine spezielle Implementierung des *TermValue*-Interfaces.

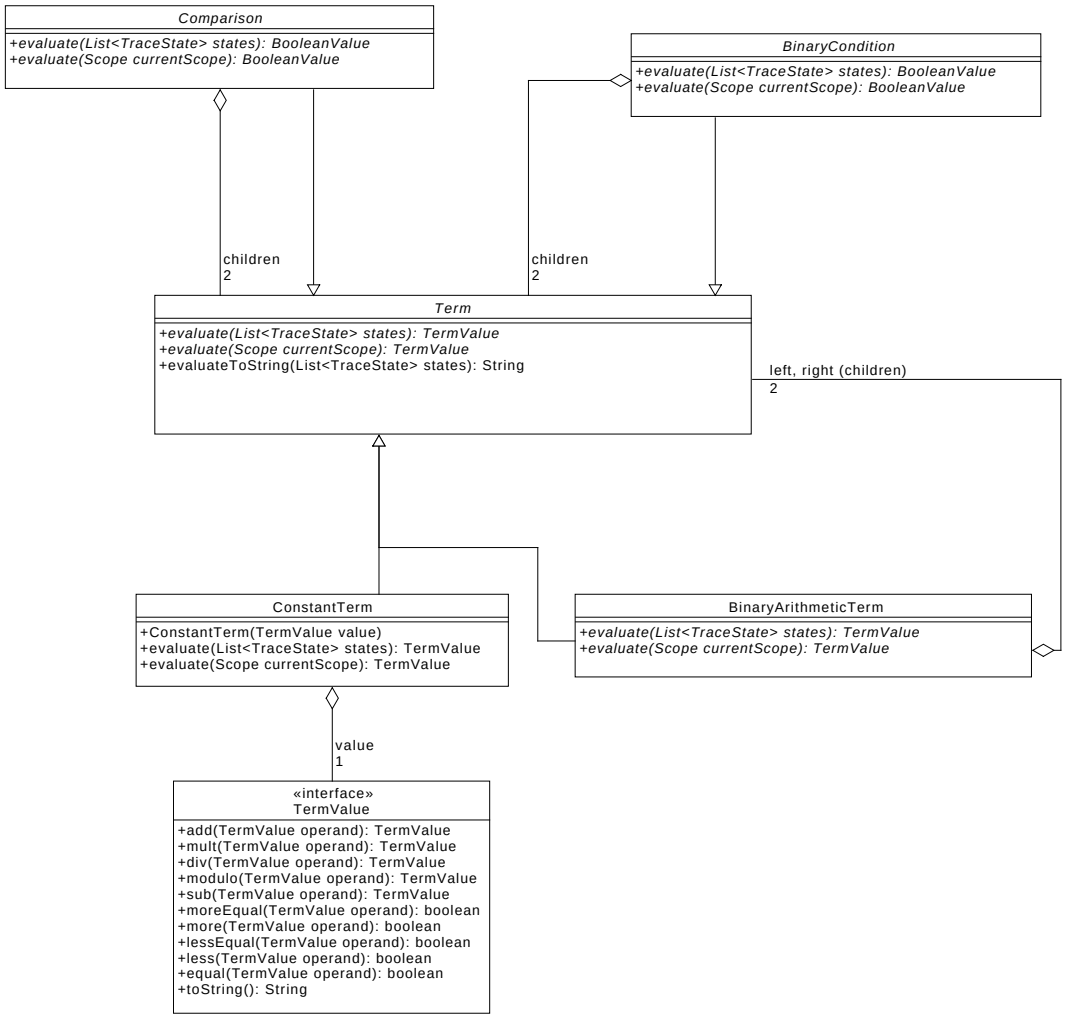

Abbildung 12: Das Termkompositum im Entwurf

WatchExpression Diese Klasse repräsentiert eine Watch-Expression, welche ein Ausdruck über den Variablen verschiedener Programme ist, der sich innerhalb seines Gültigkeitsbereiches zu einem bestimmten Wert auswertet.

ConditionalBreakpoint Ein Conditional Breakpoint ist ein Ausdruck über den Variablen verschiedener Programme, der sich innerhalb seines Gültigkeitsbereiches zu einem booleschen Wert auswertet.

7.3 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.
Dieses Unterpaket parst die Eingaben des Nutzers (d.h. sowohl Programmtexte als auch Variablen und Ausdrücke für bedingte Breakpoints und Watch-Expressions) gemäß der im Anhang (Kapitel 12) gegebenen Grammatik. Genauer wird die Verwendung des Antlr Parsers in Kapitel 10 beschrieben.

7.4 Exceptions

Dieses Unterpaket enthält Klassen, die für das Produkt spezifisch entwickelte Exceptions darstellen. Um das Handling dieser Exceptions kümmert sich die *Control*.
Jeder Interpreter muss einiges an Fehlerüberprüfung durchführen. So müssen hier vor allem semantische Fehler in den vom Nutzer eingegebenen Programmen entdeckt werden. Dazu zählen das Fehlen eines Return-Statements (*MissingReturnCallException*) oder eine im Typsystem nicht erlaubte Zuweisung (*WrongTypeAssignmentException*), wie etwa `boolean x = 9.3f`. Auch Funktionsaufrufe mit falschen Argumenten (*WrongArgumentExceptions*) oder nicht vorhandener Funktionen (*RoutineNotFoundException*) werden hier abgefangen.

8 Charakteristische Abläufe

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

8.1 Erster Programmaufruf

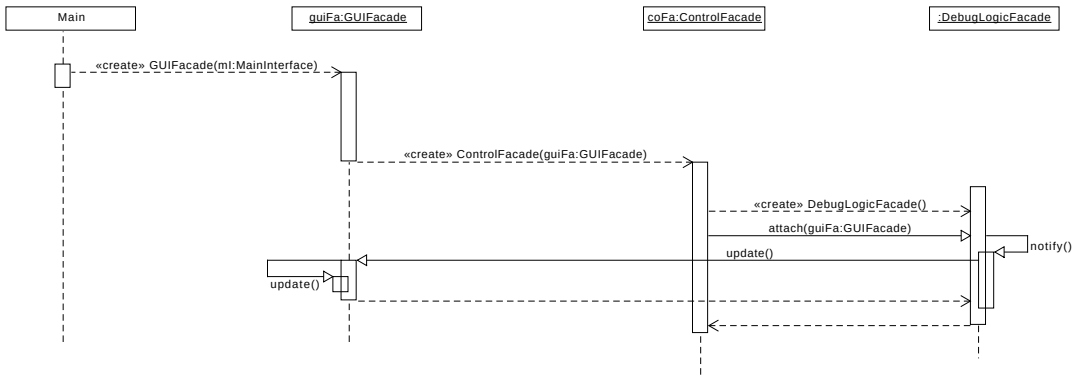

Abbildung 13: Sequenzdiagramm: Erster Programmaufruf

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.

8.2 Konfigurationsdatei laden

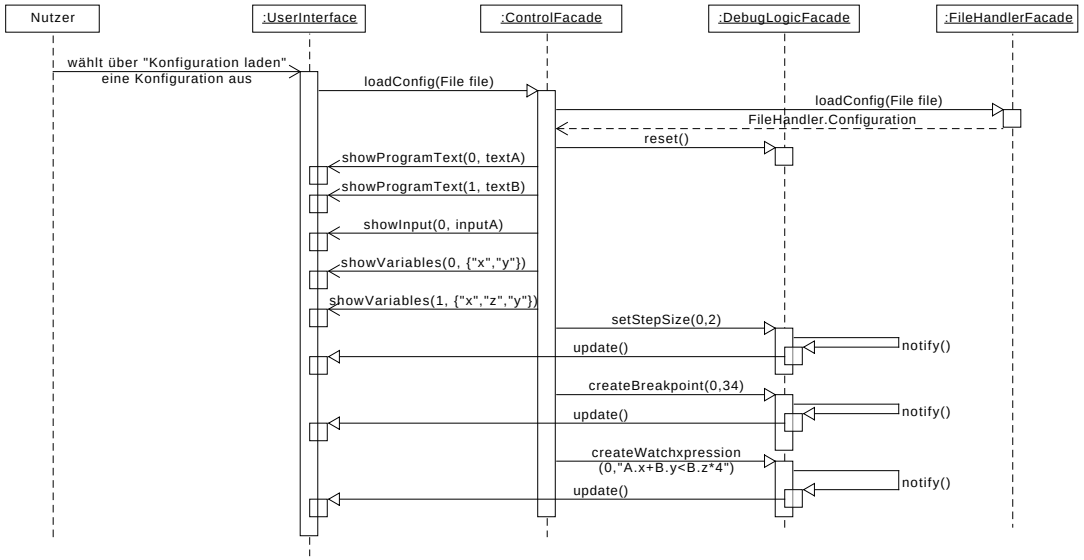

Abbildung 14: Sequenzdiagramm: Laden einer Konfigurationsdatei

Wählt der Benutzer über den Menüeintrag „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 anzuzeigenden 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.

8.3 FA140: Konfigurationsdatei speichern

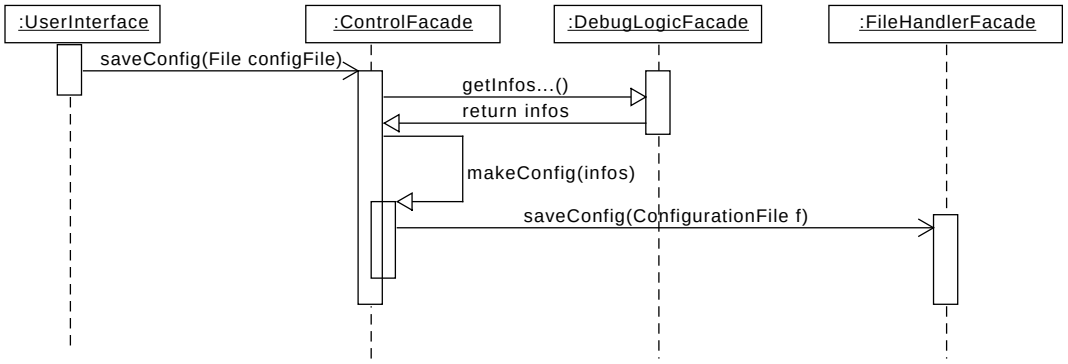

Abbildung 15: Sequenzdigramm: Speichern einer Konfigurationsdatei

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.

8.4 AF10: Hinzufügen von Programmen

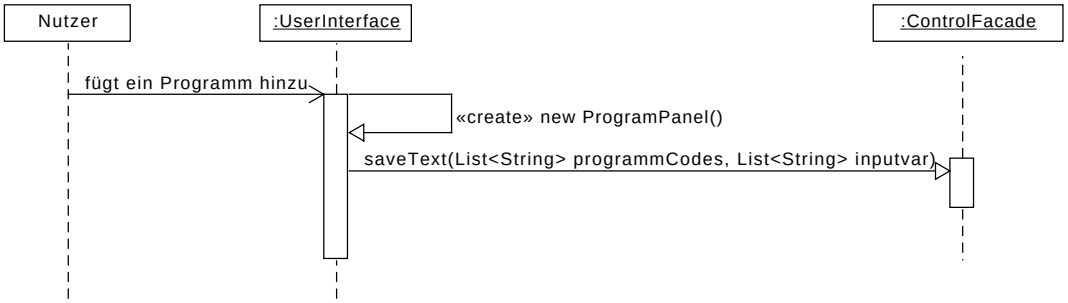

Abbildung 16: Sequenzdiagramm: Hinzufügen von Programmen durch den Menüeintrag

Fügt der Benutzer über den Menüeintrag ein neues Programm hinzu, erstellt das *User-Interface* ein neues ProgramPanel. Dies beschreibt die Funktionale Anforderung FA 170. Fügt er ein neues Programm hinzu, indem er in ein bereits vorhandenes ProgramPanel seinen Programmcode kopiert (FA 160) oder schreibt (FA 150), gibt das *UserInterface* die Informationen an die *Control* weiter. Diese Weitergabe von Informationen geschieht nach jedem Einfügen und Ändern von Programmtexten oder Eingabevariablen. Zum Start des Debugmodus gibt die *Control* alle diese Informationen an die *DebugLogic* weiter.

8.5 AF20: Ändern von Programmen

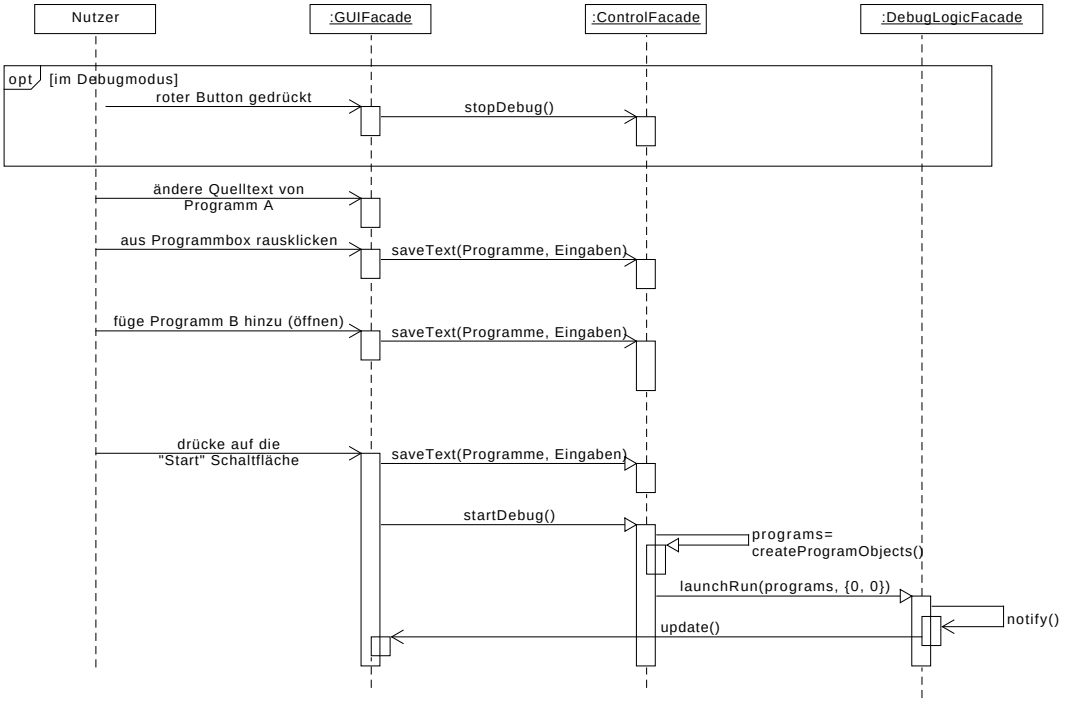

Abbildung 17: Sequenzdiagramm: Ändern von Programmen

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 Methode erstellt aus den gespeicherten Informationen Programm-Instanzen und gibt diese an die *DebugLogicFacade* weiter und startet damit den Debug-Lauf.

8.6 AF30: Setzen von Breakpoints

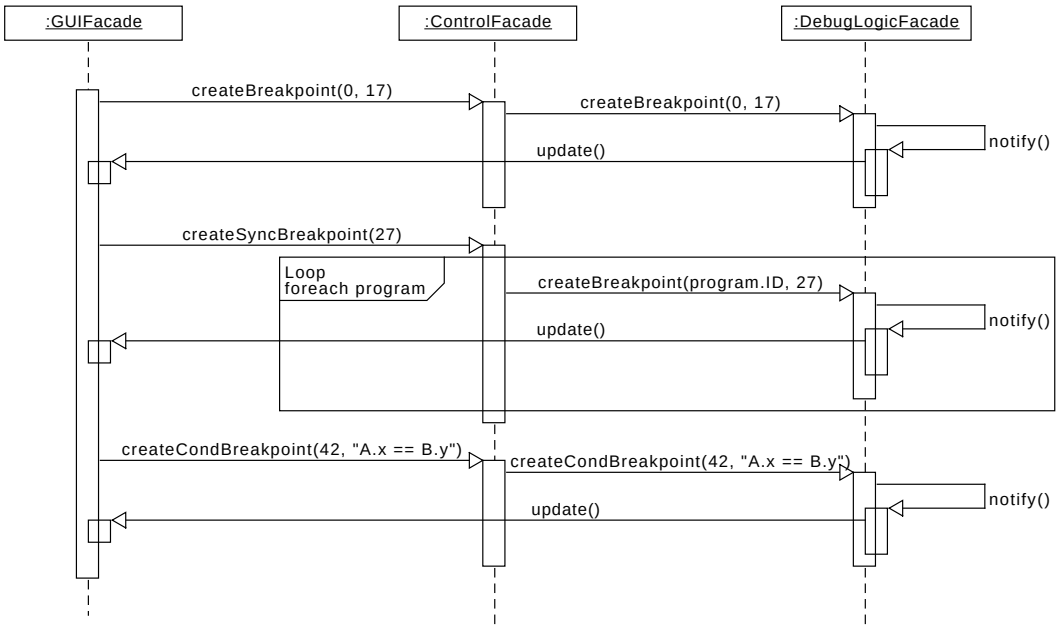

Abbildung 18: Sequenzdiagramm: Setzen von Breakpoints

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. Setzt 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.

8.7 AF40: Hinzufügen von Watch-Expressions

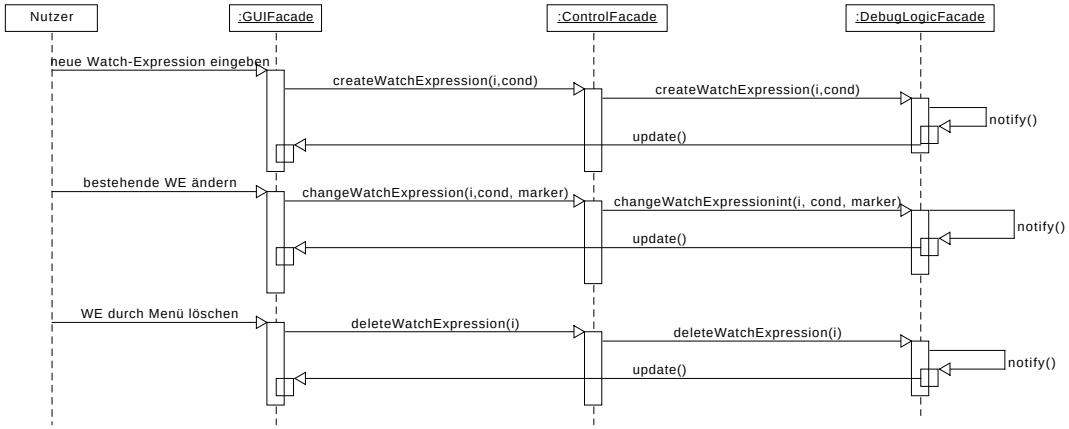

Abbildung 19: Sequenzdiagramm: Hinzufügen von Watch-Expressions

Die folgenden Vorgänge sind für Watch-Expressions (FA 110) und bedingte Breakpoints (FA 90) 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, z.B. indem er die Bereichsbindung angibt (FA 120, bzw. FA 100 für bedingte Breakpoints), 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.

8.8 Generieren von Vorschlägen

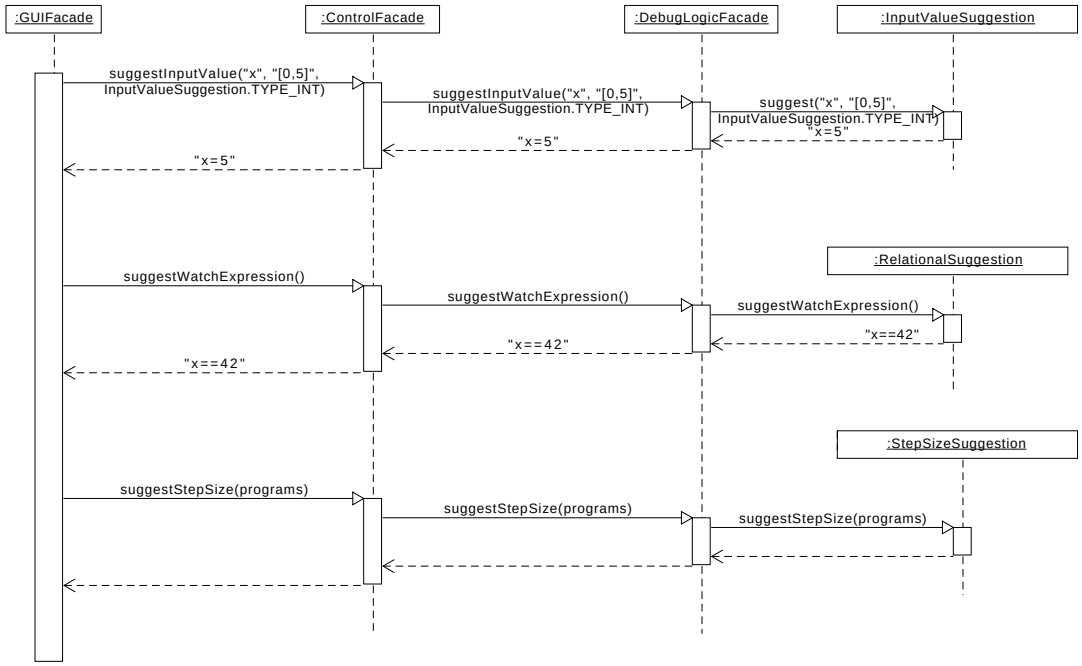

Abbildung 20: Sequenzdiagramm: Generierung von Vorschlägen für Watch-Expressions, bedingte Breakpoints, Schrittgrößen und Eingaben

Möchte der Benutzer sich eine Watch-Expression vorschlagen lassen (FA 270), so wird nach Anklicken des dazugehörigen Buttons der Befehl über die *ControlFacade* zur *DebugLogicControl* weitergeleitet. Dort wird dieser an eine Vorschlagsstrategie (z.B. *EquivalenceSuggestionStrategy*) gesendet und dort verarbeitet. Nachdem ein Vorschlag generiert wurde wird dieser als Zeichenkette, welche die finale Watch-Expression Bedingung repräsentiert, an die *GUIFacade* zurückgegeben. Danach fügt die GUI wie in AF40 eine Watch-Expression mit der erhaltenen Bedingung hinzu.
Bedingte Breakpoints lassen sich auf die gleiche Weise generieren (FA 280).
Schrittgrößen werden ähnlich vorgeschlagen (FA 300), hierbei werden der *Control* die Programme übergeben, welche diese an die *DebugLogic* weitergibt. Dies ist nötig, damit die Schrittgröße anhand der Programmcodes errechnet werden kann.
Eingaben können ebenfalls ähnlich vorgeschlagen werden (FA 220), jedoch muss der Nutzer hier angeben, für welche Variable, für welchen Variablentyp und in welchem Intervall er einen Vorschlag erhalten möchte.

8.9 AF50: Programme debuggen

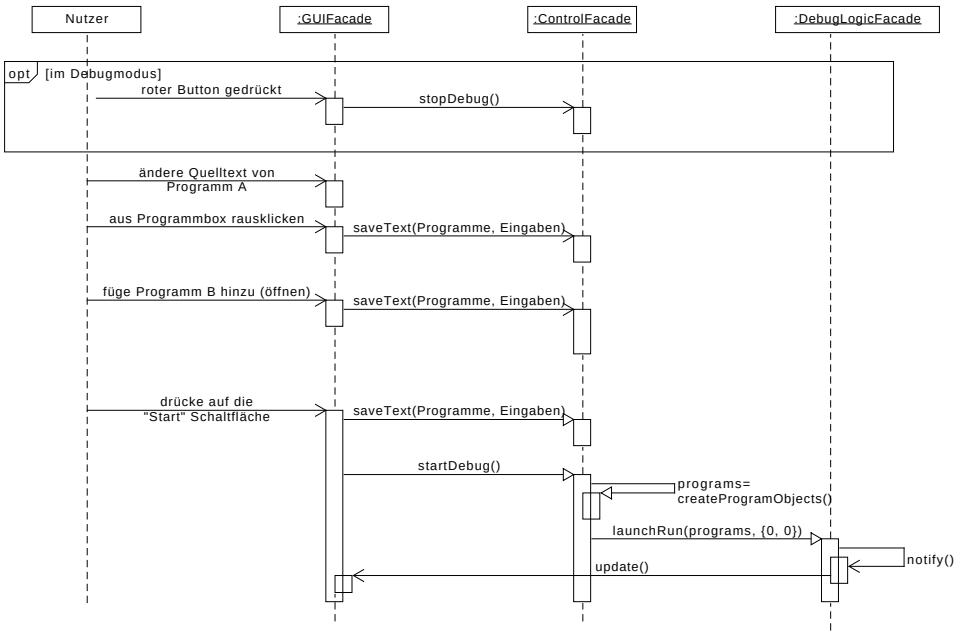

Abbildung 21: Sequenzdiagramm: Debuggen von Programmen

Dieses Sequenzdiagramm fasst die Schritte des Nutzers bei einem Debug-Lauf zusammen (FA 10). Hierbei werden vor Start des DIbuggers Programme hinzugefügt, die Schrittgröße geändert und Breakpoints hinzugefügt. Sobald der Benutzer auf Start klickt, werden seine Eingaben final gespeichert und die *Control* startet den Debug-Lauf der *DebugLogic*. Wenn der Benutzer durch Weiter (FA 130), Schritt (FA 20)oder Einzelschritt (FA 30) durch den Debug-Lauf 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.

8.10 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.

8.10.1 Erzeugung einer abstrakten Repräsentation für Watch-Expressions und bedingte Breakpoints

Watch-Expressions und bedingte Breakpoints bestehen immer aus einem vom Nutzer spezifizierten Ausdruck über den Variablen der vorkommenden Programme. Syntaktisch handelt es sich bei diesem Ausdruck einfach um einen Term. Semantisch muss aber bei bedingten Breakpoints sichergestellt sein, dass es sich um einen Term handelt, der sich zu einem Wahrheitswert auswertet. Zusätzlich enthalten Watch-Expressions und bedingte Breakpoints einen Gültigkeitsbereich in Form von mehreren *ScopeTuple*-Instanzen.
In den hier abgebildeten Diagrammen sind der Übersichtlichkeit halber nur die für die Erklärung wichtigsten Klassen meist ohne Methoden oder Attribute zu sehen. Wir betrachten das Diagramm 22.

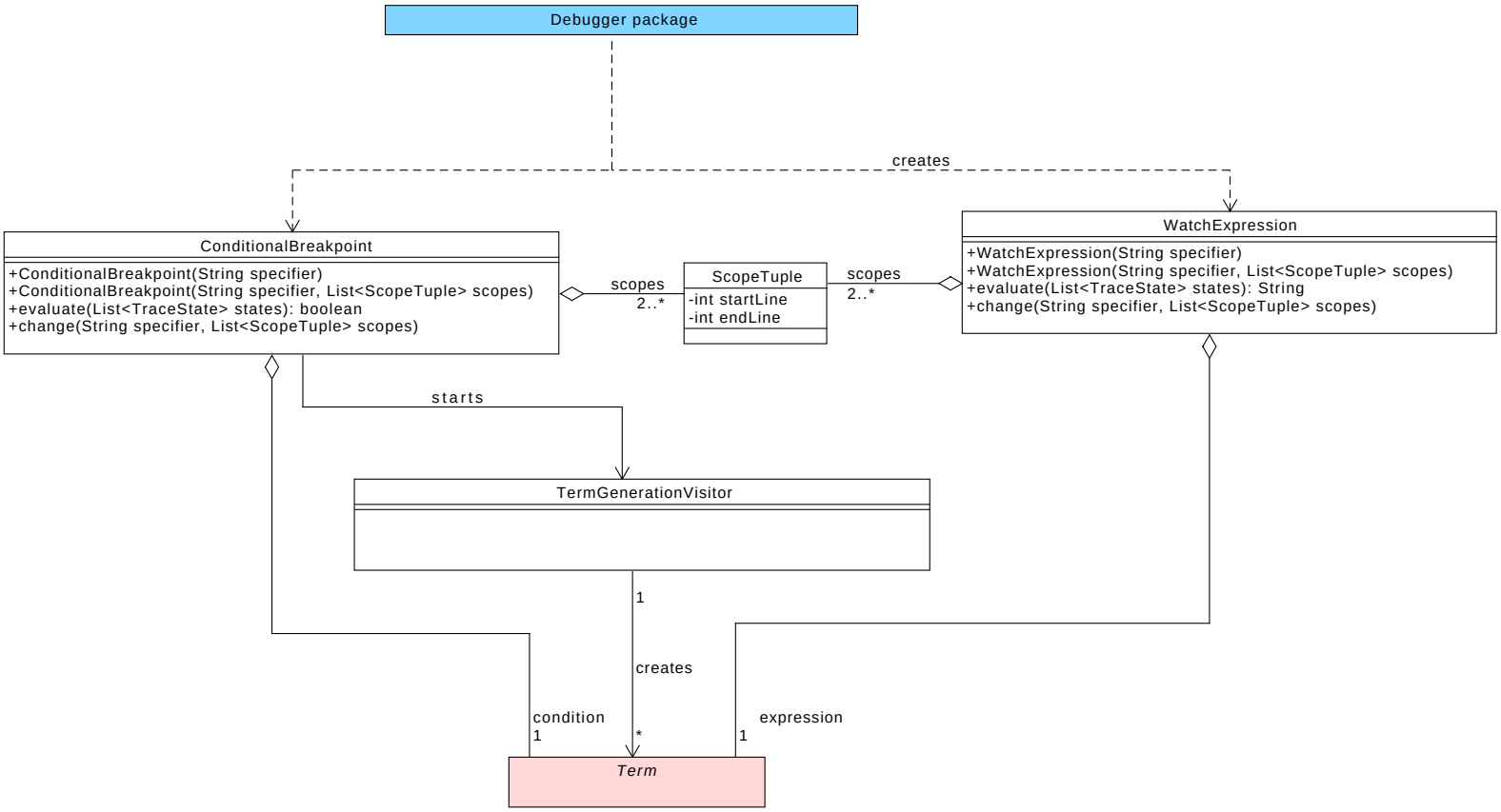

Abbildung 22: Watch-Expressions und Bedingte Breakpoints

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 *TermGenerationVisitor* gestartet, der diesen Syntaxbaum abläuft und dabei eine *Condition* erzeugt. Beim Erzeugen einer WatchExpression passiert das Gleiche. Die Gültigkeitsbereiche können entweder bei Erzeugung direkt mitübergeben werden oder erst später durch Aufruf der *change()*-Methode hinzugefügt werden. Standardmäßig ist der Gültigkeitsbereich maximal.

8.10.2 Erzeugung des Traces

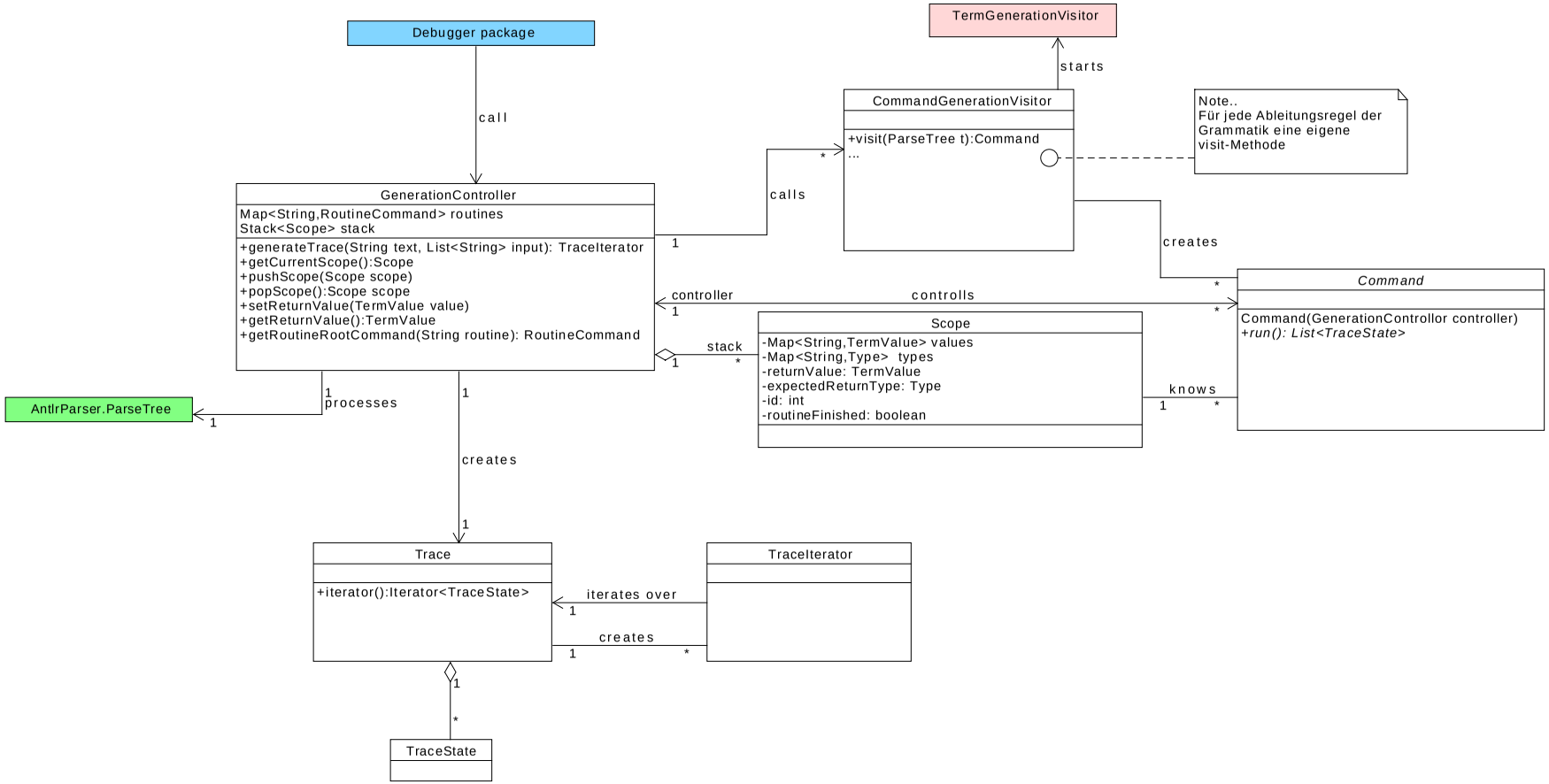

Abbildung 23: An der Tracegenerierung beteiligte Klassen

Gegeben sei 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 Abbildung 23 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 Abbildung 24 zu sehen.

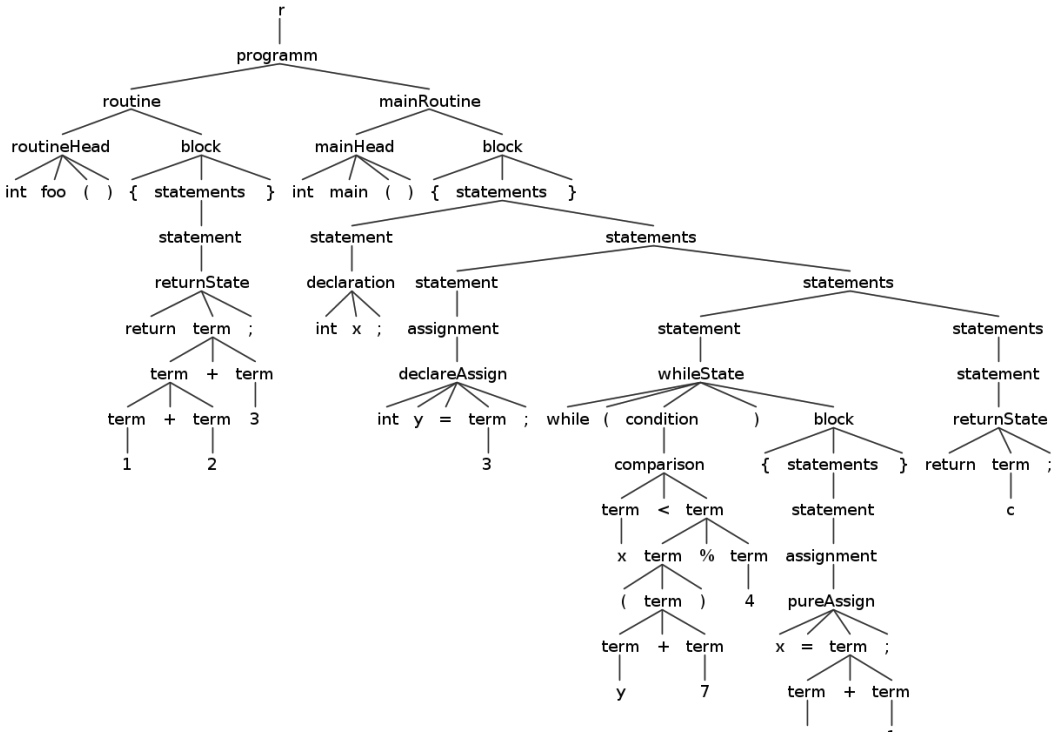

Abbildung 24: Beispiel für einen Ableitungsbaum

Über diesen Baum läuft der *CommandGenerationVisitor* und erzeugt für jede Routine einen Baum aus Befehlen (*Command*-Kompositum). Dabei verwendet er einen weiteren Visitor, die Terme in Form der in Kapitel 12 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 (Abbildung 25) aus. Die

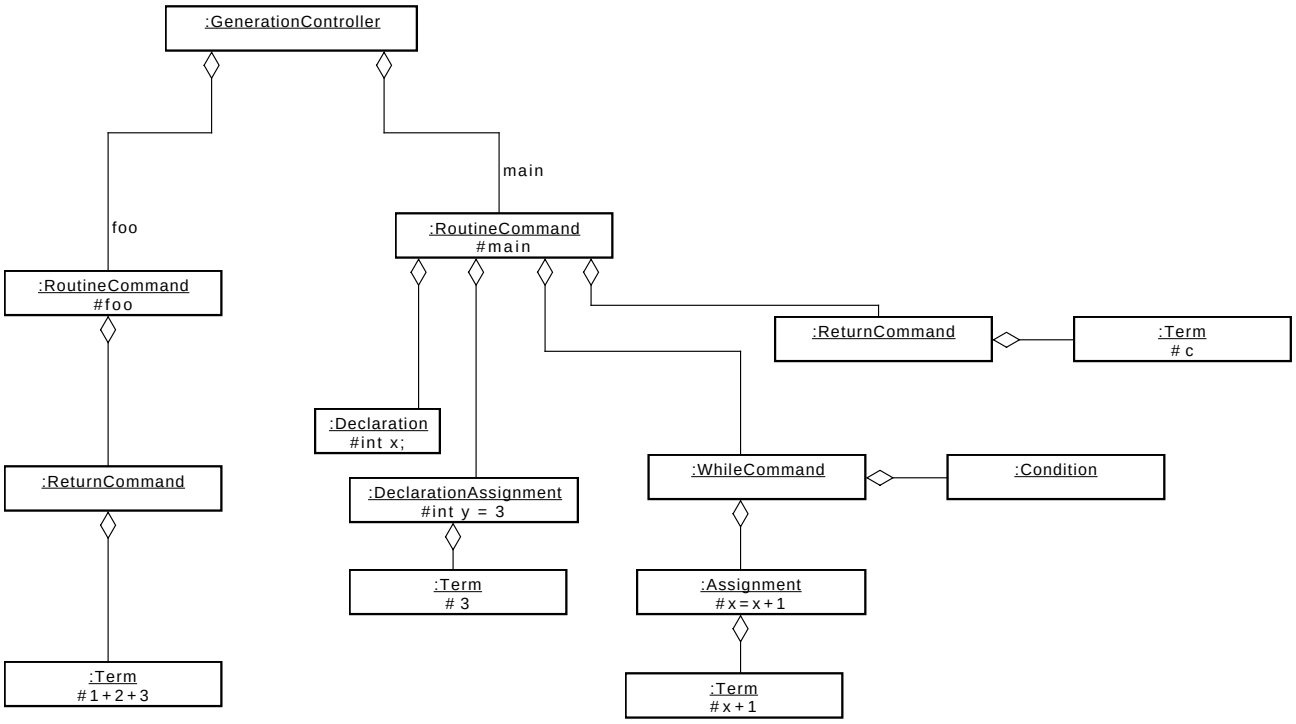

Abbildung 25: Objektstruktur nach der Commanderzeugung

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

führen. 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 Ausführung und führen sich nicht weiter aus.
Insgesamt gibt jeder Befehl nach der *run()*-Methode eine Liste von *TraceState*-Objekten zurück. So gibt ein *IfCommand*-Objekt also eine leere Liste zurück, falls die Bedingung nicht erfüllt ist, und die Konkatenation aller TraceStatelisten seiner Kinder sonst. Ein *WhileCommand*-Objekt gibt dementsprechend die Listen seiner Kinder in wiederholter Form zurück. Die gesamte Liste verpackt der *GenerationController* zu einem *Trace*-Objekt und gibt einen *TraceIterator* zurück.

9 Abhängigkeitseinteilung mit Blick auf die Implementierung

9.1 Abhängigkeiten

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 keine korrekten Daten anzeigen.

9.2 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.

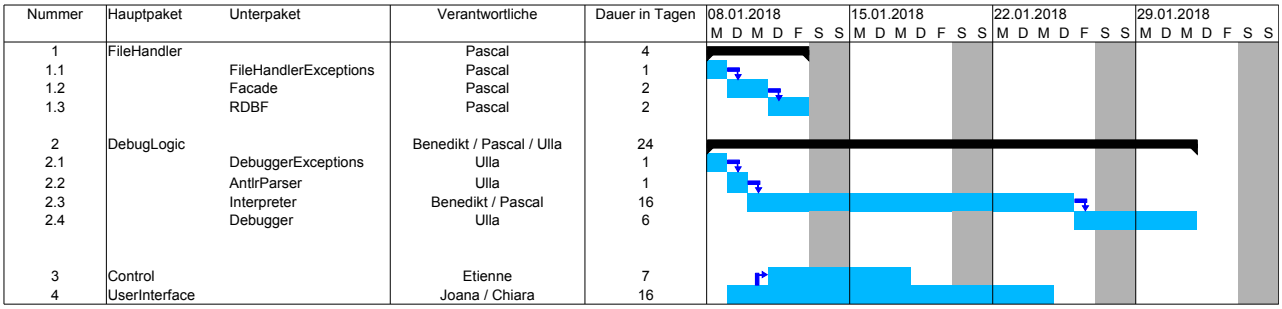

Abbildung 26: Gantt-Diagramm: Zeitplanung der Implementierung

Das obige Gantt-Diagramm beschreibt die vorläufige Zeitplanung der Implementierungs-

phase. Hierbei muss beachtet werden, dass die jeweiligen Verantwortlichen nur für die korrekte Implementierung des Pakets zuständig sind. Jedoch werden auch weitere Personen an diesen Paketen arbeiten. Geplant ist, dass nach fertiger Implementierung eines Paketes die jeweiligen Personen zu einem anderen, noch zu entwickelnden, Paket wechseln und dieses weiter implementieren, sowie Modultests dafür schreiben. Wichtig ist außerdem, dass Tests nicht nur von Personen geschrieben werden, die die zu testenden Methoden und Klassen implementiert haben, sondern auch von anderen. Dadurch können versehentliche Implementierungsfehler besser aufgedeckt und behoben werden.

10 Programmsprache, Antlr und Speicherformat

10.1 Wlang

Wlang ist eine prozedurale, imperative Sprache, welche weder objektorientiert, noch einrückungsbasiert ist.
Sie unterstützt die primitiven Datentypen int, long, float, double, char und boolean mit den selben Genauigkeiten wie Java. Außerdem sind bis zu dreidimensionale Arrays dieser Datentypen möglich.
Desweiteren unterstützt Wlang Kontrollstrukturen wie Funktionen und Prozeduren, die bedingten Ausführungen if, else if und else und while-Schleifen.
Es muss in jedem Programm eine main-Routine vorhanden sein, von der die Ausführung ausgeht. Routinen, die aufgerufen werden, müssen darüber stehen. Kommentare können sowohl zeilenweise (//) als auch zeilenübergreifend (/*Kommentar*/) benutzt werden. Sämtliche Variablen müssen innerhalb einer Routine stehen. Auch Rekursion ist möglich.
Für die Definition der Kontextfreien Antlr-Grammatik siehe Anhang, in den Unterkapiteln 12.1 und 12.2.

10.2 Verwendung von Antlr

Wie bereits im zweiten und dritten Kapitel beschrieben, benutzt der DIbugger im Paket *DebugLogic.AntlrParser* extern erzeugte Java-Klassen, um die textuelle Nutzer-Eingabe zu einem ablaufbaren *Parse Tree* umzuwandeln. Diese Klassen werden von dem frei verfügbaren Programm Antlr© (Version 4) erzeugt. Ein *Parse Tree* (manchmal auch *Syntax Tree* genannt) ist dabei ein Baum mit einer eindeutigen Wurzel, der die syntaktische Struktur eines Textes – in diesem Fall der Nutzereingabe – mit Hilfe einer kontextfreien Grammatik darstellt. Die Spezifikation der Grammatik findet sich im Anhang, in den Unterkapiteln 12.1 und 12.2.

Auf Basis der Grammatik generiert Antlr mehrere Java-Klassen, wie einen Lexer, einen Parser, sowie einen Visitor und ein Visitor-Interface. Die Funktionalität dieser Klassen wird im Unterpaket *DebugLogic.Interpreter* genutzt und erweitert.

10.2.1 Vorteile der Verwendung eines Parser-Generators

Die Verwendung eines Parser-Generators wie Antlr im Allgemeinen hat mehrere Vorteile gegenüber einem "von Hand" geschriebenen Parser: Offensichtlich spart es Zeit und Ressourcen, die in die Entwicklung des eigentlichen Produktes gesteckt werden können. Nicht zu vernachlässigen ist auch, dass Änderungen an der Grammatik zunächst nur in der Grammatik selbst gemacht werden müssen und nicht sofort Änderungen in mehreren, möglicherweise sehr großen Klassen nach sich ziehen. Zuletzt ermöglichen Parser-Generatoren wie Antlr, dass man ein Projekt auf einem Parser aufsetzt, der bereits von einer großen Gruppe an Nutzern getestet wurde, und so die Fehleranfälligkeit reduziert.

10.2.2 Anforderungen an den verwendeten Parser-Generator

Mindestanforderungen an einen Parser-Generator für dieses Projekt waren Java-Kompatibilität, Kostenfreiheit und eine offene Lizenz, die mit der vom DIbugger verwendeten Lizenz kompatibel ist.

Antlr kann Java-Code produzieren, ist kostenlos und unter BSD-Lizenz veröffentlicht. Obwohl es auch andere Parser-Generator gibt, die diese Anforderungen erfüllen, spricht für Antlr, dass es auf allen Betriebssystemen läuft, in eine Entwicklungsumgebung eingebunden werden kann, eine aktuelle Version mit gutem Support bereitstellt und viele Online-Ressourcen dazu verfügbar sind. Diese Faktoren erleichtern dabei nicht nur die gegenwärtige Entwicklung des Produkts sondern auch zukünftige Erweiterungen.

10.3 Relational Debugger File (RDBF) Speicherformat

RDBF ist ein nicht einrückungsbasiertes, imperatives Speicherformat, welches das Speichern von komplexen Blockstrukturen erlaubt. Eine Blockstruktur besteht aus einem Blocknamen und Rumpf, welcher in sich wieder mehrere Blöcke und Daten hat. Daten bestehen aus einem Namen, Wert und Typ, welcher aber erst von einem Parser herausgefunden werden muss. Es werden die Datentypen String, int, long, float, double und boolean unterstützt. Weiter gibt es einen spezialisierten TextBlock, der nur einen Fließtext zum Speichern enthält. Kommentare können durch das Kennzeichnen am Anfang einer Zeile mit „//“ erzeugt werden. Das Speicherformat ist durch eine Kontextfreie Grammatik definiert. Diese kann im Anhang unter 12.3 eingesehen werden.

11 Änderung zum Pflichtenheft

Erweiterte Syntax für Watch-Expressions und bedinge Breakpoints In Kapitel 12.8 des Pflichtenheftes wurde die Syntax der Watch-Expressions und bedingter Breakpoints durch eine kontextfreie Grammatik definiert, welche zu den Funktionalen Anforderungen FA90 und FA110 gehört. In dieser Grammatik war kein bedingter Breakpoint der Form $A\&\&B$ bzw. $A||B$ oder $!A$ ableitbar. Diese Einschränkung ergab sich, während des Entwurfs, als nicht notwendig und wird demnach aufgehoben. Eine genaue Definition der Watch-Expressions und bedingten Breakpoints in Form einer Antlr-Grammatik ist im Anhang 12 zu finden. Insgesamt können diese auf syntaktischer Ebene aus beliebigen Termen bestehen. Dies ändert jedoch nichts an den Funktionalen Anforderungen FA90 und FA110, sondern nur an deren Umsetzung.

12 Anhang

Startnichtterminal in den Grammatiken ist immer die Variable r.

12.1 Kontextfreie 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')'#BracketCondition
| condition '&&' condition #AndCondition
```

```
| condition '||' condition #OrCondition
| '!'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: '=';
```

12.2 Kontextfreie Antlr-Grammatik für Syntax von bedingten Breakpoints und Watch-Expressions

```
grammar Terms;
r: generalTerm;

generalTerm: condition | term;

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

12.3 Kontextfreie Grammatik RDBF Format

```
R : ((STATEMENT | BLOCK | BLOCK_TEXT | COMMENT) NEWLINE)*;

BLOCK : VARIABLE '{' NEWLINE
((STATEMENT | BLOCK | COMMENT) NEWLINE)*
NEWLINE '}';
BLOCK_TEXT : VARIABLE '{' NEWLINE 'def_blockLen=' I_VALUE NEWLINE TEXT NEWLINE '}';

STATEMENT : VARIABLE '=' VALUE;
```

```
COMMENT : '//' (WORD SPACE?)*;

VALUE : (I_VALUE | L_VALUE | F_VALUE | D_VALUE | B_VALUE | S_VALUE);
I_VALUE : NUMBER+;
L_VALUE : NUMBER+ 'L';
F_VALUE : NUMBER+ ('.' NUMBER*)? 'f';
D_VALUE : NUMBER+ ('.' NUMBER*)?;
B_VALUE : 'true' | 'false';
S_VALUE : '\'' (WORD SPACE?)* '\'';

TEXT : ((~[])* (SPACE | NEWLINE)?)*;
WORD : (LITERAL | NUMBER | SPEC_CHAR)+;
VARIABLE : (LITERAL | NUMBER | SPEC_CHAR_VAR)+;

LITERAL : [A-Z] | [a-z];
NUMBER : [0-9];
SPEC_CHAR : ~('\'' | NEWLINE);
SPEC_CHAR_VAR : '$' | '&' | '_';
NEWLINE : '\n';
SPACE : ' ';
```