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

4. Dezember 2017

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument dokumentiert die Ergebnisse der Entwurfsphase (*28.11.-22.12.2017*) im Rahmen des Moduls Praxis der Softwareentwicklung (PSE) am Lehrstuhl „Anwendungsorientierte formale Verifikation - Prof. Dr. Beckert“ am Karlsruher Institut für Technologie (KIT).
Hierbei handelt es sich um den Entwurf des Produkts *DIbugger*, welches im Pflichtenheft definiert wurde. Das Entwurfsdokument beschreibt die Paketeinteilung, Beschreibung der Klassen und Abläufe und genaue Spezifikation der Abhängigkeiten und Kernkomponenten.
Die Implementierung während der Implementierungsphase (*09.01.-02.02.2018*) wird anhand der Vorgaben in diesem Dokument durchgeführt.
Hierbei werden das Geheimnisprinzip und Lose Kopplung berücksichtigt, um für erhöhte Verständlichkeit zu sorgen.

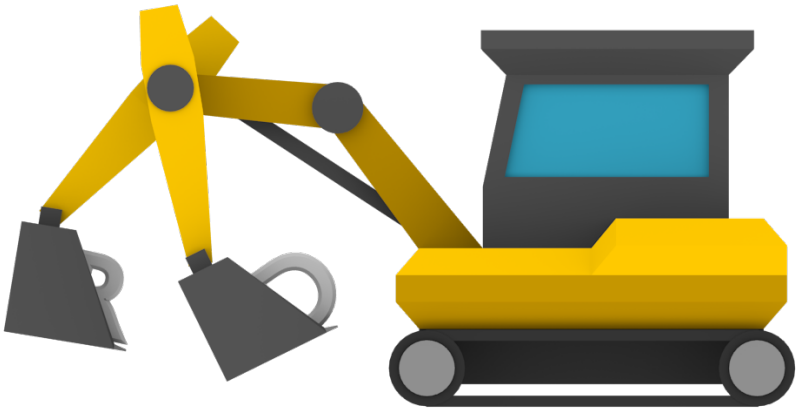

Abbildung 1: Produktlogo

2 Paketeinteilung

2.1 Übersicht

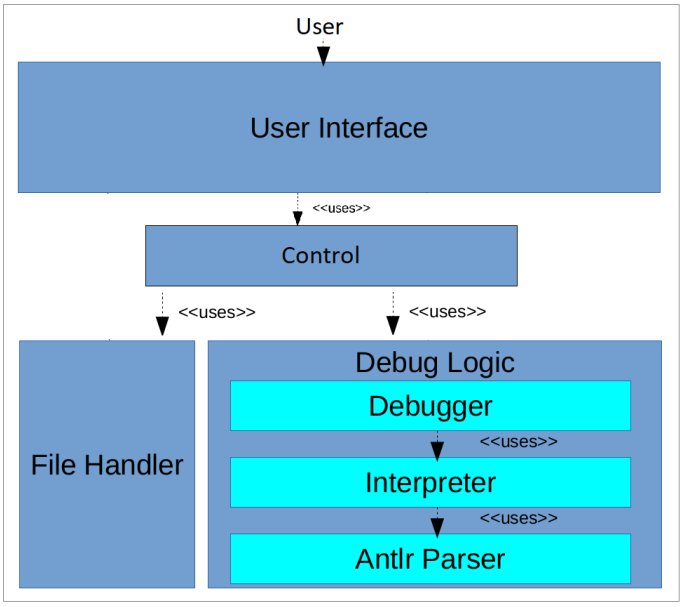

Abbildung 2: Architekturdiagramm

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

2.2 User Interface

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

Schnittstellen

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

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

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

2.3 Control

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

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

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

2.4 File Handler

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

Schnittstellen

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

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

Benutztrelation Der FileHandler hat keine Unterpakete und steht an unterster Stelle der Benutztrelation. 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 6 gegebenen Grammatiken.
Darauf aufbauend in der mittleren Schicht finden sich die Subpakete *DebugLogic.TraceGenerator* und *DebugLogic.RelationalExpressionGenerator*, die beide die Aufgabe haben, diese Ableitungsbäume durch interpretieren in eine abstrakte und leicht handhabbare Form zu bringen. Da beide Subpakete eine gemeinsame Schicht darstellen, findet hier auch ein hohes Maß an Kommunikation statt.

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

2.5.1 Debugger

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

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

Benutztrelation Das Subpaket benutzt die Subpakete *DebugLogic.TraceGenerator* und *DebugLogic.RelationalExpressionGenerator*. Um die üblichen Debugmechanismen wie Schritte und Weiter durchführen zu können, nutzt dieses Subpaket den vom Subpaket *DebugLogic.TraceGenerator* bereitgestellten Iterator. Um WatchExpressions und bedingte Breakpoints auszuwerten und zu repräsentieren, nutzt dieses Subpaket die vom Subpaket *DebugLogic.RelationalExpressionGenerator* bereitgestellte abstrakte Repräsentationen.

2.5.2 Trace Generator

Aufgaben Dieses Subpaket nimmt den Quelltext eines WLang-Programms entgegen und hat die Aufgabe, einen Pfad über den gesamten Programmfluss dessen zu erzeugen, sodass später darüber iteriert werden kann. Dazu gehört auch das Prüfen auf semantische Fehler, etwa das Fehlen eines return Statements.

Schnittstellen

- Angebotene Funktionalität:
  Stellt einen Iterator über den Ausführungspfad eines gegebenen Programmes zur Verfügung.

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

Benutztrelation Das Unterpaket benutzt das Unterpaket *DebugLogic.AntlrParser*, um damit aus den reinen Zeichenketten einen Syntaxbaum gemäß der in 6 gegebenen Grammatik für die Sprache WLang erzeugen zu lassen. Auf dieser Vorraussetzung baut die Arbeit des Subpakets auf.

2.5.3 Relational Expression Generator

Aufgaben Dieses Subpaket nimmt Zeichenketten, die Watch-Expressions und bedingte Breakpoints beschreiben, entgegen und hat die Aufgabe, diese zu interpretieren. Dazu wird in diesem Paket eine abstrakte Darstellung dieser erzeugt.

Schnittstellen

- Angebotene Funktionalität:
  Erzeugt aus gegebenen Zeichenketten für Watch-Expressions und bedingte Breakpoints eine abstrakte Repräsentation, sodass diese dann leicht ausgewertet werden kann.

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

Benutztrelation Das Unterpaket benutzt das Unterpaket *DebugLogic.AntlrParser*, um damit aus den reinen Zeichenketten einen Syntaxbaum gemäß der in 6 gegebenen Grammatik für Watch-Expressions und Bedingte Breakpoints erzeugen zu lassen. Auf dieser Vorraussetzung baut die Arbeit des Subpakets auf.

2.5.4 Antlr Parser

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 6 gegebenen Grammatik.

Schnittstellen

- Angebotene Funktionalität:
  Prüft die textbasierten Eingaben des Nutzers auf Übereinstimmung mit der gegebenen Grammatik und erzeugt aus der Eingabe einen ablaufbaren Syntaxbaum,

der dann vom Unterpaket *DebugLogic.RelationalExpressionGenerator* weiter ausgewertet werden kann.

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

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

3 Beschreibung der Klassen

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

3.1 Klassen in Paket 1

3.2 Klassen in Paket 2

...

4 Charakteristische Abläufe

Beschreibung von charakteristischen Abläufen anhand von Sequenzdiagrammen. Beispielsweise bieten sich Testszenarien aus dem Pflichtenheft hier an. Wir empfehlen Sequenzdiagramme möglichst früh zu erstellen, denn dabei werden die Schnittstellen zwischen Packages und Klas- sen klar. Auf Klassen oder Pakete in Beschreibung aller Klassen verweisen

5 Abhängigkeitseinteilung

Mit Blick auf den Implementierungsplan: Aufteilung in Klassen/Pakete, die unabhängig vonein- ander implementiert und getestet werden können.

6 Formale Spezifikation von Kernkomponenten

Speicherformate, Sprachdefinition(formal)

7 Änderung zum Pflichtenheft

Änderungen zum Pflichtenheft, z.B. gekürzte Wunschkriterien.

8 Anhang

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