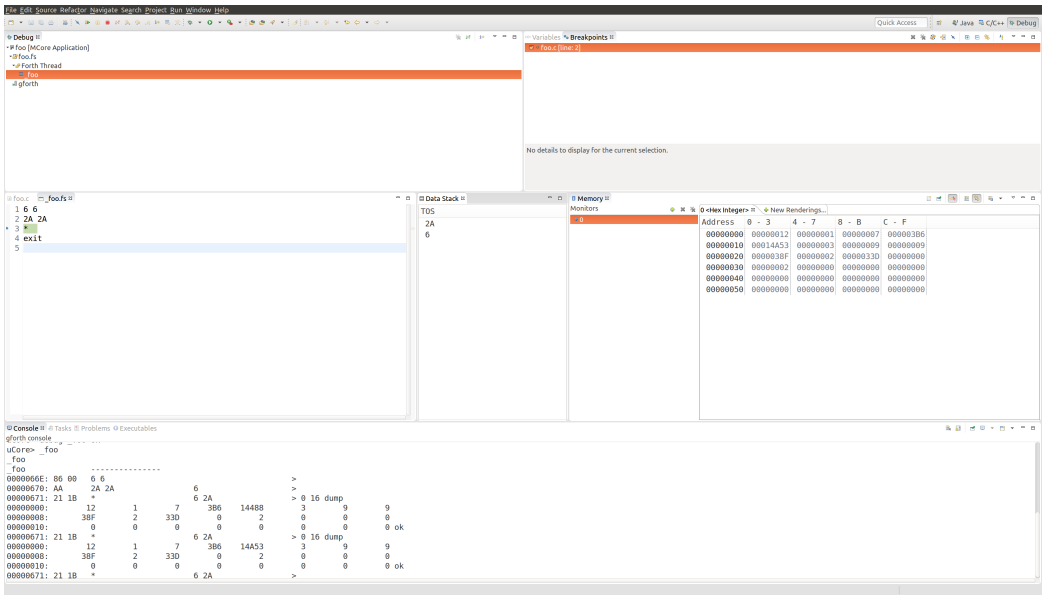
Bachelor-Thesis

Eclipse Entwicklungsumgebung für MicroCore

Benjamin Neukom

August 2015



Betreuer: Carlo Nicola

Inhaltsverzeichnis

In dieser Arbeit wird beschrieben, wie eine IDE für MicroCore mittels Eclipse implementiert werden kann.

1. Einleitung

Todo Copy from Forth MDT ZIEL!

2. Die Eclipse Platform

In diesem Kapitel wird gezeigt, was Eclipse für Möglichkeiten bietet, um eine moderne Entwicklungsumgebung zu implementieren. Diese verschiedenen Möglichkeiten werden Verglichen und die Vor- und Nachteile aufgezeigt. Es werden auch die wichtigsten Eclipse Features, welche für das Entwickeln von Eclipse Rich Client Platform (RCP) Applikationen benötigt werden, beschrieben.

2.1. Eclipse als Platform

Eclipse RCP bietet eine Basis um beliebige, (nicht zwingendermassen Entwicklungsumgebungen) Betriebssystem unabhängige Applikationen zu entwickeln. Es bietet Mechanismen, wie Plugins und Extension Points, um modulares programmieren zu unterstützen und vereinfachen. Auch bietet das Framework Features, wie das Konzept von Views, Editoren und vielen mehr, welche häufig in Applikationen gebraucht werden.

2.2. Plugins

Eclipse Applikationen nützen eine auf der OSGi Speizifikation basierten Runtime. Eine Komponente in dieser Runtime ist ein Plugin. Eine Eclipse RCP Applikation besteht aus einer Ansammlung dieser Plugins. Ein Eclipse Plugin kann Extension Points 2.3 anderer Plugins ansteuern und eigene Extension Points oder APIs anbieten.

2.3. Extension Points

Um Plugins erweitern zu können, bietet Eclipse das Konzept von Extension Points an. Über Extension Points können Plugins ein bestimmte Funktionalität anbieten, welches von anderen Plugins per XML deklarativ angesteuert werden kann.
So existiert zum Beispiel ein Plugin, welches einen Extension Point für Views definiert. Andere Plugins können nun über diesen Extension Point, deklarativ in einem XML File, eine neue View erstellen und diese konfigurieren. [1]

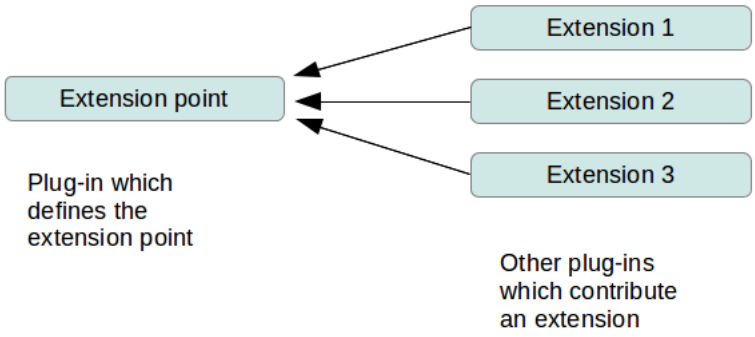


Abbildung 2.1.: Ein Plugin, welches einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.

Es ist auch möglich, eigene Extension Points zu definieren, falls ein Plugin für andere Entwickler offen stehen soll für Erweiterungen.

2.4. Eclipse basierte MCore Entwicklungsumgebung

Eclipse als Grundlage für eine Entwicklungsumgebung zu verwenden eignet sich besonders gut, da Eclipse schon einiges an Funktionalität für eine IDE zur Verfügung stellt und schon viele Entwicklungsumgebungen (PHP, C/C++, Pyhton, D unter anderem) als Eclipse RCP entwickelt wurden. Auch existieren einige Tools, auf welche ich noch genauer eingehen werde, wie Xtext und DLTK, welche das entwickeln einer Entwicklungsumgebung weiter vereinfachen.

2.4.1. JDT

Eine Möglichkeit die MCore Entwicklungsumgebung zu implementieren wäre die standard Features von dem Eclipse Java Development Tools (JDT) zu verwenden. Dies wäre sehr Aufwendig, da alle Features, wie Syntax Highlighting, Code Completion oder eine Outline neu implementiert werden müssten.

2.4.2. XText

XText ist ein Framework, welches es erleichtert, eine auf Eclipse basierte Entwicklungsumgebungen zu programmieren. Es ermöglicht auf schnelle Weise ein Grundgerüst einer IDE mit Features wie:

- Ein Editor mit Syntax Coloring
- Code Completion
- Compiler Integration

- Ein Java-basierter Debugger
- Eine Outline
- Indexing

zu generieren. [2] Es muss lediglich eine ANTLR [3] Grammatik für die Sprache definiert werden. Der grosse Nachteil ist, dass C, inklusive Preprozessor, zu parsen sehr schwierig ist und eine Entwicklungsumgebung somit auch mit Xtext nicht einfach zu implementieren ist.

2.4.3. Dynamic Language Toolkit Framework

Das Dynamic Language Toolkit (DLTK) ist ein weiteres Framework, welches ein Grundgerüst für eine Entwicklungsumgebungen generieren kann. Ursprünglicherweise war das Framework nur für dynamische Sprachen geignet, es kann aber auch für statische Sprachen verwendet werden. Die D Entwicklungsumgebung wurde mittels DLTK realisiert [4]. Es bestehen aber wieder dieselben Nachteile wie bei Xtext. Da C schwierig zu parsen ist, müsste trotz dem Framework noch viel selbst implementiert werden. Da D keinen Preprozessor besitzt, konnte für diese Entwicklungsumgebung das DLTK Framework verwendet werden.

2.4.4. Eclipse C/C++ Development Tools

Das Eclipse C/C++ Development Tools (CDT), ist eine Eclipse Distribution mit Unterstützung für C und C++. Das CDT bietet alle Features, welche man von einer Entwicklungsumgebung erwartet und stellt Extension Points zur Verfügung um diese für eine eigene Entwicklungsumgebung zu gebrauchen. So kann man mit relativ wenig Aufwand einen neuen Compiler in die Entwicklungsumgebung einbinden. Dies wurde auch schon mehrfach gebraucht, um verschiende C/C++ Compiler im Eclipse CDT zu integrieren. Auf die Einbindung des Compilers wird im Kapitel 3 genauer eingegangen.

2.4.5. Verwendung für MCore Eclipse

Ich habe mich dazu entschieden, das Eclipse CDT als Target Platform zu wählen. Somit können alle Features, welche das Eclipse CDT zur Verfügung stellt, gebraucht werden. Frameworks, welche ein Grundgerüst einer Entwicklungsumgebung generieren, funktionieren nur Teilweise für C und sind somit keine guten Alternativen.

3. Compiler Integration

In diesem Kapitel wird beschrieben, wie der Compiler in die Entwicklungsumgebung eingebunden wurde um ein C-File nach Forth zu übersetzten. Und welche Möglichkeiten mit dem CDT für das Editieren von C-Files zur Verfügung stehen.

3.1. Integration in CDT

Für die Integration in das CDT stehen einige notwendige Plugins und Extension zur Verfügung.

3.1.1. CDT Extension Points

3.1.2. Tool

3.1.3. Toolchain

3.2. Builder

TODO findet Errors im Path (falls Eclipse nicht richtig aufgesetzt wurde)

3.3. Error Parsing

3.4. C Programmierung im Eclipse

4. Programm Launch

In diesem Kapitel wird beschrieben, wie ein C-Programm aus der IDE gestartet werden kann und was dafür implementiert wurde.

4.1. Launch

4.1.1. Run

4.1.2. Debug

5. Forth Kommunikation

In diesem Kapitel wird beschrieben, wie die Entwicklungsumgebung mit dem Forth Prozess kommuniziert. Die Kommunikation mit dem Forth Prozess ist von zentraler Bedeutung, da viel der Funktionalität der Entwicklungsumgebung davon abhängt, dass die Kommunikation stabil läuft. Es wird gezeigt wie die Kommunikation designt, implementiert und getestet wurde.

5.1. Prozess Kommunikation

Um mit dem Prozess zu kommunizieren, gibt es grundsätzlich zwei Möglichkeiten. Die erste ist ein Machine Interface zu implementieren, oder direkt mit dem Prozess kommunizieren. In folgenden Kapiteln werden die beiden Möglichkeiten kurz besprochen.

5.1.1. GDB/MI-Commands

Eine Möglichkeit mit dem Prozess zu kommunizieren wäre, ein MachineInterface (MI) wie es für den GDB implementiert wurde, zu gebrauchen. GDB/MI ist ein Linien basiertes Maschinen orientiertes Text Interface zu dem GDB. Es wurde dazu entwickelt, damit der GDB als Debugger in ein grössers System einfacher einbindbar ist. [5] Eine MI ähnliches Interface wäre mit grossem Aufwand verbunden, da das Interface zuerst definiert werden müsste. Dafür könnte aber viel des CDT Debugging Mechanismus verwendet werden, da der CDT Debugger auch auf dem GDB/MI basiert. Auch ist der CDT Debugger sehr kompliziert [6] und es wäre ein grosser Einarbeitungsaufwand notwendig um den Forth Debugger darauf basieren zu können. [6]

5.1.2. Direkte Kommunikation mit dem Prozess

Eine weitere Möglichkeit wäre, direkt die Befehle an den Prozess senden und auf Antworten warten. Dafür müsste einige Klassen implementiert werden, um das Kommunizieren zu vereinheitlichen und vereinfachen.

Probleme bei der Kommunikation

Bei der Kommunikation mit dem Prozess können einige Probleme Auftreten. Es kann sein, dass der Prozess plötzlich keine Antworten mehr gibt. Auch weiss man nicht, wie lange es dauern wird, bis der Prozess Antwort gibt. Diese Probleme müssen mit den im vorigen genannten Klassen sauber gelöst werden.

5.2. Kommunikation

Ich habe mich dafür entschieden, die Kommunikaton nicht über ein MI ähnliches Interface zu implementieren. Der Aufwand ein solches Interface zu designen und die Einarbeitungszeit für den CDT Debugger wären zu gross. Die Kommunikation wird also direkt mit dem Prozess erfolgen. Das Design und die Implementation wird in folgenden Kapiteln besprochen.

5.3. API Design

In einem ersten Schritt wurde ein API designt, welches verwendet werden soll um die Kommunikation mit dem Prozess möglichst einfach zu halten.

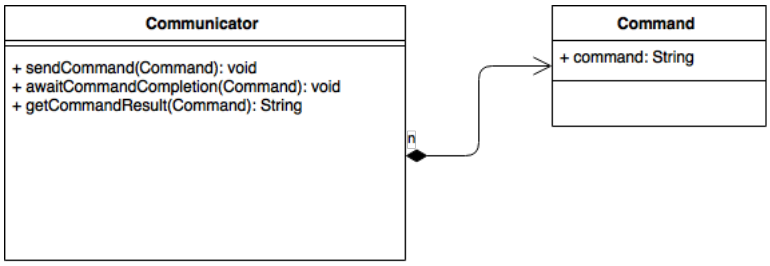


Abbildung 5.1.: Ein erstes Design des Prozess API. Die zentrale Kommunikation geschieht über die Communicator Klasse. Es können Commands abgesetzt werden und es kann auf Resultate gewartet werden.

Bei der Implementierung kamen dann einige Änderungen hinzu. Auf die endgültige Implementation werde ich im nächsten Kapitel genauer eingehen.

5.4. Implementierungs Details

TODO class diagramms TODO some source code of the central dispatch of the commands?

5.4.1. Klassenbeschreibung

5.4.2. Kommunikation mittels Commands

5.4.3. Forth Output Parsing

5.4.4. Await auf Resultate

6. Debugger

In diesem Kapitel wird beschrieben, wie der Debugger in Eclipse integriert wurde. Es wird aufgezeigt, was für Möglichkeiten existieren einen Debugger in Eclipse zu integrieren und welche implementiert wurden.

6.1. Breakpoints

Als erstes müssen für den Debugger Breakpoints für Forth gesetzt werden können. Dafür stehen zwei Möglichkeiten zur Verfügung auf welche ich kurz eingehen möchte.

6.1.1. Per Konsole

Die erste Möglichkeit ist, die Breakpoints per Konsole zu setzen. Das senden der Commands funktioniert mit den im Kapitel 5 beschriebenen Klassen. In der Konsole kann der Command

```
debug _function
```

abgesetzt werden, um einen Breakpoint zu setzen.
TODO Konsolen Screenshot

6.1.2. Im Source Code

Eine weitere Möglichkeit ist, Breakpoints im C-File zu setzen. Dies wurde so umgesetzt, dass der Breakpoint nur auf eine Funktionsdefinition gesetzt werden kann. Alle anderen Zeilen des C-Source Codes können nicht direkt auf den übersetzten Forth Code abgebildet werden und sind deshalb nicht erlaubt für die Breakpoints.
Eclipse CDT stellt den Abstract Synatx Tree (AST) des C-Files zur Verfügung. Mit Hilfe des AST kann überprüft werden, ob sich der Breakpoint wirklich auf einer Funktionsdefinition befindet.
TODO Screenshot

6.2. Konsolen basierter Debugger

Eine erste Implementation des Debuggers ist, den schon existierenden Forth Konsolen Debugger im Eclipse zu integrieren. Dieser kann mit dem im Kapitel 5 beschriebenen Prozess Kommunikationsmitteln angesteuert und in einer Eclipse Console View angezeigt werden.
TODO Screenshot

6.3. Forth Debugger

Frontend für Konsolen Debugger.

6.3.1. Neue Debugger Aktionen

Jump Action

Over Action

6.3.2. Stack View

User Interface

6.3.3. Memory View

User Interface

6.4. C Debugger

7. Entwicklungsumgebungs Einstellungen

Für die Entwicklungsumgebung wurde eine Eclipse Preference Page erstellt. In diesem Kapitel wird erklärt, was in dieser Preference Page für Einstellungen vorgenommen werden können und was diese für Auswirkungen haben.

7.1. Umbilical

Test

7.2. Loader

Test

8. Optimierungen

In diesem Kapitel wird beschrieben, was für Optimierungen für den Compiler vorgenommen wurden. Wie diese implementiert wurden und was für Resultate diese liefern.

8.1. Peephole Optimierung

Peephole Optimierungen ist eine Art von Optimierung, welche auf einer kleinen Sequenz von generiertem Code durchgeführt wird. Dieses Sequenz wird Peephole oder Window genannt. Code Generatoren generieren häufig Instruktionen, welche ohne Seiteneneffekte, entfernt werden können. Peephole Optimierungen können die grösse der Codes um 15-40 Prozent verkleinern und sind heute in allen gängigen Compilern implementiert. [7] Zu der Peephole Optimierung gehören unter anderen folgende Arten von Optimierungen:

- Constant Folding - Konstante Expressions auswerten
- Constant Propagation - Konstante Werte in Expressions substituieren
- Strength Reduction - Langsame Instruktionen mit äquivalenten schnellen Instruktionen ersetzen.
- Combine Operations - Mehrere Oprationen mit einer äquivalenten ersetzen
- Null Sequences - Unötige Operationen entfernen [8]

8.1.1. Beispiele

Constant Propagation

Folgende Instruktionen

```
1
2
swap
+
dup
```

können durch:

```
2
2
```

ersetzt werden. Die Instruktionen swap, + und dup können schon zur Kompilierzeit durchgeführt werden.

Combine Operations

Folgende zwei Instruktionen

```
rot
rot
```

können durch

```
-rot
```

ersetzt werden. Die zwei rot Instruktionen sind äquivalent zu einer -rot Instruktion.

8.1.2. Optimierungen

TODO Ablauf Diagram

8.1.3. Automatische Generierung von Peephole Optimierungen

8.1.4. Constant Propagation

Unter Constant Propagation versteht man, dass substituieren von Konstanten werden im Code. Dies kann zur Folge haben, dass mehrere Instruktionen schon zur Kompilierzeit ausgwertet werden können wie bei den Beispielen 8.1.1 zu sehen ist.

Die Constant Pro

8.1.5. Resultate und Tests

8.1.6. Mögliche Erweiterungen

SSA

A. Literaturverzeichnis

[1] Vogella. Eclipse extension points and extensions - tutorial. http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html, 2013.

[2] Xtext. https://eclipse.org/Xtext/, 2013.

[3] Antlr. http://www.antlr.org/.

[4] Bruno Medeiros. D development tools. https://github.com/bruno-medeiros/DDT.

[5] The gdb/mi interface. https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.

[6] Understanding the gnu debugger machine interface (gdb/mi). http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug2/.

[7] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. In *ACM Trans. Program. Lang. Syst.*, pages 191–202, 1980.

[8] Peephole optimization. https://en.wikipedia.org/wiki/Peephole_optimization.

B. Abbildungsverzeichnis

C. Tabellenverzeichnis

D. Ehrlichkeitserklärung

Hiermit bestätigen die Autoren, diese Arbeit ohne fremde Hilfe und unter Einhaltung der gebotenen Regeln erstellt zu haben.

Benjamin Neukom

Ort, Datum Unterschrift