Dokumentation Grinj

Table of Contents

Allgemeiner Teil

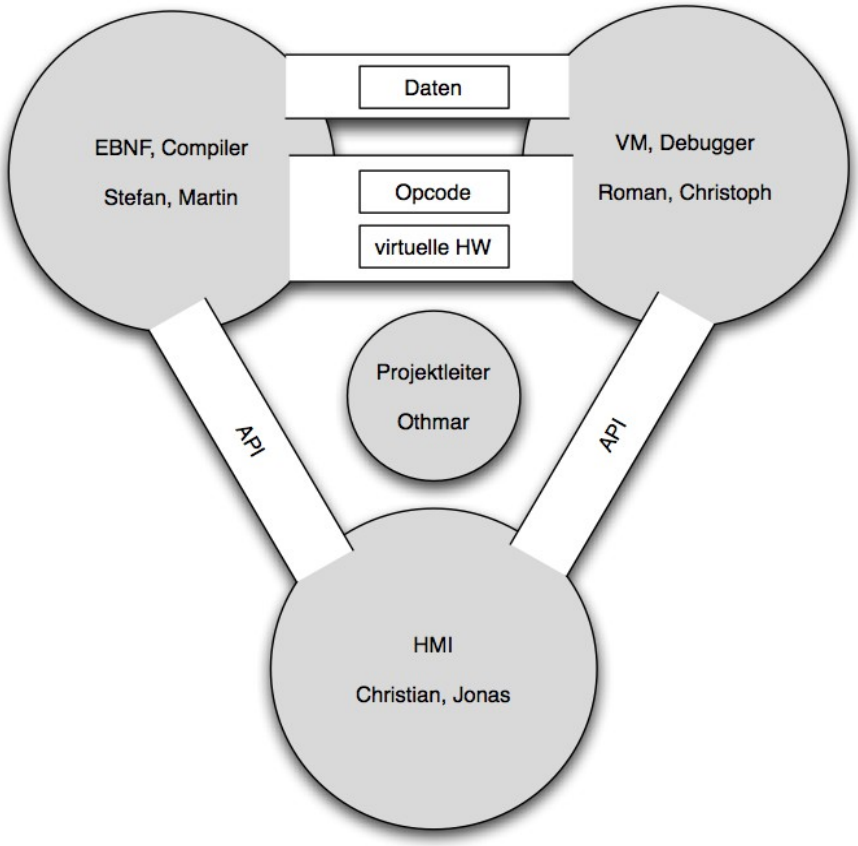
Name des Projekts

Der Projektname war Grinj und bedeutet Grinj is not Java.

Einteilung der Gruppen

Die sieben Projektmitglieder wurden auf die Teilgebiete Compiler, VM, HMI und Projektleiter aufgeteilt. Die Aufteilung ist in der untenstehenden Grafik ersichtlich.

Die sogenannten Special Teams (Compiler, VM, HMI) sind für die Teilgebiete zuständig und kennen diese speziell gut. Für die Koordination zwischen den einzelnen Special Teams und die Kommunikation mit Herrn Frei ist der Projektleiter verantwortlich.



RFC's und Minimalstandard

Die RFC's und der Minimalstandard wurden im cvs unt grinj/docs/rfc abgelegt. Ebenfalls wurde eine Kopie im Anhang angefügt.

Minimalstandard

Der Minimalstandard besteht aus den RFC_0001 bis RFC_0004.

RFC_0001

Der RFC_0001 wurde nach einem Gespräch mit Herrn Frei gelöscht. Hier wurde die Sprachdefinition von Brainf**k abgehandelt.

RFC_0002

Die Definition der Opcodes und ein Beispiel wird im RFC_0002 beschrieben.

Dieser RFC ist vorallem für die Kommunikation zwischen dem Compiler und der VM nötig, weil der Output der Compiler der Input der VM ist.

RFC_0003

Das Format des Compilats muss von beiden Seiten (Compiler und VM) implementiert werden. Der Parser der VM ist auf die Richtigkeit des Compilats angewiesen.

RFC_0004

Der Austausch der Debug – Informationen zwischen der VM und der GUI erfolgt via Socket. Im Debug – Protokoll werden die Status – Meldungen der VM beschrieben.

Arbeitsweise

Am Anfang des Projekt wurde eine Codeverwaltungssystem (wobei cvs bekannt war) und ein Zeitmanagementsystem gesucht.

Für cvs haben wir uns entschieden, weil es den meisten Projektmitgliedern ein Begriff war und unser Hoster(Stefan Zoller) das System im Griff hat.

Auf der Suche nach einem Zeit- und Aufgabenmanagementsystem stiessen wir auf XPWeb. Um die geleisteten Arbeiten zu erfassen, wurde kurzerhand eine Erweiterung von Stefan Zoller hinzugefügt.

cvs

Das content version system, kurz genannt cvs, verwaltete den Sourcecode. Die Versionsverwaltung wird vollständig übernommen.

Wird gleichzeitig an der selben Datei gearbeitet, so entsteht bei späterem einchecken ein Konflikt, welcher der Entwickler löst und danach die neuste Version auf den Server lädt.

Ohne Codeverwaltungssytem wäre das Projekt zum scheitern verurteilt gewesen.

XPWeb

Um die Verwaltung der Aufgaben und das Zeitmanagement zu realisieren, wurde XPWeb ausgewählt. Die Verwaltung und Erfassung der Aufgaben ist wirklich sehr einfach.

Für die Zweiterfassung hat Stefan Zoller eine Erweiterung zum XPWeb geschrieben.

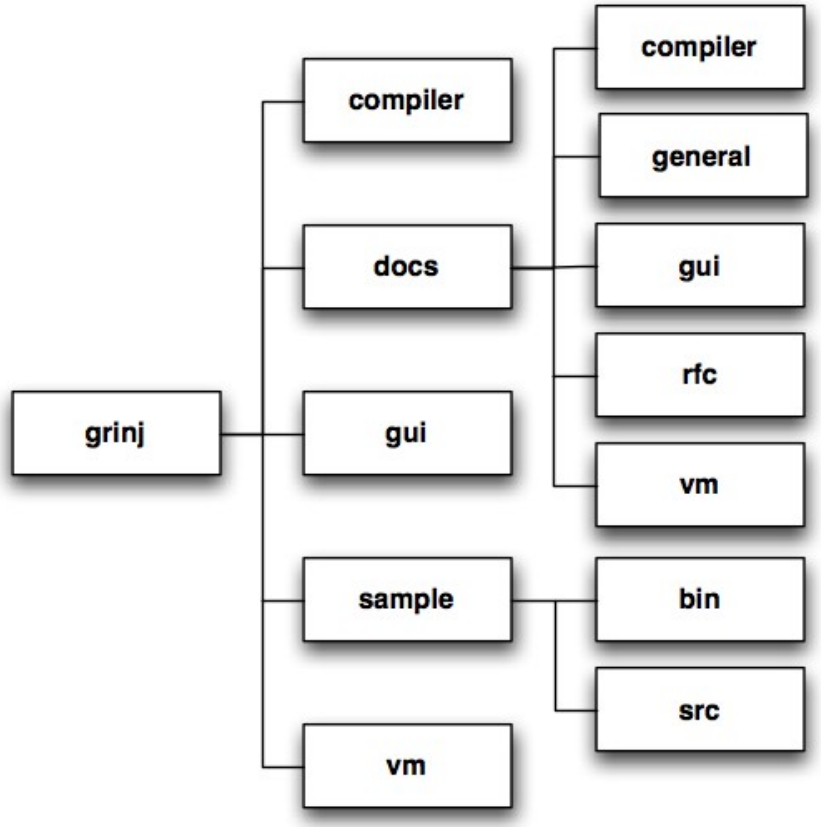
Aufbau

Ordnerstruktur

Jedes Special Team hat im grinj einen Ordner erhalten, in welchem Sie die Teilprojekte entwickelten.

Zusätzlich wurde der Dokumentationsordner (docs) beim Start des Projektes kreiert, wo ebenfalls jedes Special Team einen Ort erhielt für die Doku. Der allgemeine Teil der Doku, wie Montagssitzungen, Gruppeneinteilung Präsentation wurde unter docs/general abgelegt.

Die Projektdefinitionen, wie Schnittstellen, Debuginfos, Dateiformat werden in rfc abgelegt.



Der Grinj-Compiler

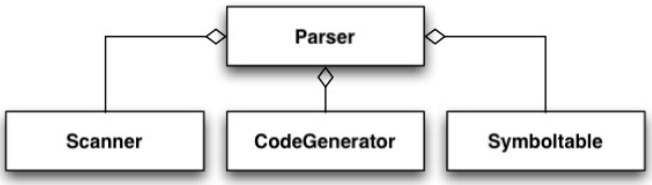
Einleitung

Das Erstellen eines Compilers ist grundsätzlich keine triviale Angelegenheit. Dazu aber zuallererst eine gute Nachricht: Der von uns verwendete "Compiler-Compiler" CocoR erledigt einen Grossteil der Arbeit automatisch. Allerdings erstellt CocoR nicht alle Dateien, die für einen Compiler nötig ist. Es werden zwei Grundklassen "Parser" und "Scanner" erstellt, die in die eigene Applikation eingebunden werden können; dazu gehören anschliessend aber natürlich noch weitere Klassen, die manuell erstellt werden müssen.

Zu CocoR selbst haben wir einige hervorragende Dokumentationen gefunden; Sie liegen im Dokumentations-Ordner bereit. Das erste Dokument "CocoR-UserManual" beinhaltet das Benutzerhandbuch der Coco-R-Programmierer, das zweite File "CocoR-Manual2" enthält ein ganzes Buch zum Thema Compiler- und VM-Bau. Die Autoren verwenden im Buch durchweg CocoR als Generator sowie C++ als Programmiersprache, deswegen stellt das PDF eine sehr brauchbare Einführung und Anleitung für das Programm dar.

"The Big Picture"

Das UML-Diagramm zeigt die Minimal-Anforderung eines Compilers, der mit CocoR erstellt worden ist:



Wir haben eine `main()`-Funktion, die das Parser-Objekt einbindet; in das Objekt `Parser` selbst wird ein `Scanner`, eine `SymbolTable` und ein `CodeGenerator` gepackt, die die Arbeit erledigen.
Dazu kommen dann noch einige kleinere Klassen wie `Errors` (wird in `Parser.h` von CocoR deklariert), `BitArray`, `Token` und `Buffer` (diese Klassen werden von CocoR in `Scanner.h` deklariert), `Obj` (wird in `SymTab.h` deklariert) sowie ein enum `Op` (wird in `CodeGen.h` verwendet).

Von `main()` aus werden eigentlich nur drei Dinge erledigt. Zuerst werden alle benötigten Objekte erstellt. Danach wird die `parse()`-Methode des Parsers aufgerufen, und falls hier keine Fehler aufgetreten sind, wird mittels des `CodeGen`s die gewünschte Code-Form ausgegeben. In unserem Fall wird dem Compiler beim Aufruf erstens mitgeteilt, ob der Output in Binärcode oder in der eigenen Assembler-Sprache (als ASCII-Text) erfolgen soll, ob die Debug-Informationen mitgeschrieben werden sollen, und ausserdem bekommt das

Programm noch Angaben über das Sourcecode-File sowie den Dateinamen für den Output mit.

`Parser` verwendet die an CocoR mitgelieferte Sprachbeschreibung, um die Syntax des Quelltextes zu durchforsten und mit Hilfe von `Scanner` eine Baumstruktur aufzubauen, die in einer verketteten Liste von Opcode-Objekten in `CodeGen` gespeichert wird. Werden dabei Fehler entdeckt (semantisch oder syntaktisch), gibt der Parser oder der Scanner - je nach Art des Fehlers - eine entsprechende Meldung direkt an den Fehlerkanal aus und stoppt den Programmablauf.
Geht alles gut - d.h., ist der Parse-Prozess korrekt durchlaufen und das Programm codiert - wird es durch das `CodeGen`-Objekt ausgegeben.
Die Ausgabe erfolgt dabei einfach durch ein letztes Durchlaufen der verketteten Liste der Baum-Einträge und eine Dekodierung der enthaltenen Werte durch Opcodes, die in der `CodeGen`-Klasse angegeben worden sind.

CocoR

CocoR kann als Sourcecode aus dem Internet heruntergeladen werden und ist in verschiedenen Geschmacksrichtungen erhältlich. Wir möchten am Schluss des Kompilier-Vorgangs C++-Code bekommen, also wählen wir die C++-Variante des Tools, kompilieren es und kopieren das erhaltene Binary in einen Ordner, der in der `PATH`-Variable angegeben ist.

Sprachdefinition als EBNF-File

CocoR benötigt eine Sprachdefinition zur Steuerung. Zuerst muss also eine EBNF-Beschreibung der gewüschten Sprache erstellt werden; unsere GRINJ-EBNF sieht wie folgt aus:

```
grinj =       "program" ident "{" {VarDecl} {ProcDecl} "}".
VarDecl =     Type ident {"," ident} ";".
ProcDecl =    "void" ident "(" ")" "{" {VarDecl | Stat} "}".
Type =        "int" | "bool" | "string".

Stat =        ident ("=" Expr ";" | "(" ")" ";")
            | "if" "(" Expr ")" Stat ["else" Stat]
            | "while "(" Expr ")" Stat
            | "read" ident ";"
            | "write" Expr ";"
            | "writec" Expr ";"
            | "{" {Stat | VarDecl} "}".

Expr =        SimExpr [RelOp SimExpr].
SimExpr =     Term {AddOp Term}.
Term =        Factor {MulOp Factor}.
Factor =      (ident | "true" | "false" | number | "-" Factor
               | string).

AddOp =       "+" | "-".
MulOp =       "*" | "/".
RelOp =       "==" | "<" | ">".
```

Die ATG-Datei

Mit dem EBNF-File alleine gibt sich CocoR noch nicht zufrieden. Damit das Tool die korrekten Klassen Parser und Scanner bereitstellen kann, muss in einer weiteren Datei - dem ATG-File - zusätzliche Information zu den einzelnen Sprachelementen gegeben werden.

Unser ATG-File ist ziemlich gross, wir verzichten deshalb hier auf eine komplette Abbildung. Sie finden die Datei ausgedruckt im Anhang oder ansonsten im src-Ordner des Compiler-Projekts.

Auf den ersten ca. 10 Zeilen werden Klassen-Variablen für die zu erstellende Klasse `Parser` deklariert. Diese Zeilen werden genau so in die Header-Datei der Parser-Klasse übernommen.

Anschliessend werden im Abschnitt „`CHARACTERS`“ einige Variablen für CocoR definiert: Klassen von Zeichen („`digits`“, „`letter`“ ...), woraus besteht ein `Token`, welche Zeichen werden zur Kennzeichnung von Kommentaren im Sourcecode verwendet und welche Zeichen soll der Compiler im Sourcecode ignorieren.

Unter dem Titel „`Productions`“ folgen nun die genaueren Definitionen der Sprach-Konstrukte. Man erkennt gut auf der linke Seite des Files die ursprüngliche EBNF-Form; rechts (jeweils zwischen Klammern mit Punkten abgesetzt) steht Code, den CocoR an der

entsprechenden Stelle in die Klasse `Parser` einfügen soll.

Abändern der Sprachdefinition

Um neue Features zur Sprache hinzuzufügen, müssen verschiedene Dinge beachtet werden.
Zum Ersten ist es unvermeidbar, dass eine klare EBNF-Formulierung der neuen Features existiert.
Diese EBNF-Formulierung muss anschliessend auch ins ATG-File eingepasst werden, zusammen mit Code, den der Parser beim Auftreten der Konstrukte auszuführen hat.
Unter Umständen reicht das aber nicht - je nach Art der Änderung sind tiefgreifende Struktur-Modifikationen nötig, beispielsweise neue Klassen, die geschrieben und dann auch eingebunden werden müssen, um neue Datenstrukturen zu halten, oder neue Opcodes, die dann nicht nur im ATG-File verankert, sondern auch im `CodeGen`-Objekt angepasst werden müssen.
Unser Code ist so konzipiert, dass ein erneutes Verwenden von CocoR mit den von uns mitgelieferten Sprachbeschreibungs-Dateien je eine Klasse `Parser` und `Scanner` generiert, die eins zu eins ins Grinj-Projekt einkompiliert werden können. Dazu haben wir unter anderem auch die Datei `Parser.frame`, die von CocoR als Vorlage für die `Parser`-Klasse verwendet wird, so abgeändert, dass unsere Header-Files automatisch eingebunden werden.

Abändern der Definition - zwei Beispiele

Hinzufügen einer direkten Variablen-Deklaration

In der ursprünglichen Beispiel-Sprache `Taste` können Variablen nicht gleichzeitig deklariert und gleich mit Werten versehen werden. Folgendes Beispiel ist in Taste ungültig:

```
int i = 7;
```

Wir haben uns dieses Feature aber für grinj gewünscht und deshalb eine kleine Änderung an EBNF- und ATG-File vorgenommen, damit der von CocoR generierte Code diese Syntax erlaubt.

Im originalen Taste-EBNF ist die Variablen-Deklaration wie folgt definiert:

```
VarDecl =    Type ident {"," ident} ";".
```

Unsere Erweiterung ist ziemlich klein. Bei grinj heisst die entsprechende Zeile:

```
VarDecl =    Type Ident { ',' Ident } ( '=' Expr ) ";".
```

Dazu kam eine kleine entsprechende Änderung am ATG-File. Die neue Definition der Prozedur „VarDecl“ sieht wie folgt aus:

```
VarDecl                (. char * name; int type; Obj obj;
```

```
                          int type2; .)
= Type<type>
  Ident<&name>          (. obj = tab->NewObj(name, var, type); .)
  { ',' Ident<&name>    (. obj = tab->NewObj(name, var, type); .)
  }
  ( '=' Expr<type2>     (. if (type2 != type) SemErr("incompatible
                                                    types");
                          if( type2 != string ) {
                               if (obj.level == 0)
                                    gen->Emit(STOG, obj.adr);
                               else gen->Emit(STO, obj.adr);
                          } .)
  ) ';'.
```

Hinzufügen eines neuen Typs `string`

In der obigen ATG-File-Änderung ist eine zweite Neuerung von grinj gegenüber Taste sichtbar: Wir haben einen neuen Variablen-Typ `string` eingeführt.

Das Verarbeiten von Strings war uns ziemlich bald ein wichtiges Anliegen. Allerdings ist das Feature nicht im Minimal-Standard erwähnt; wir haben ihm deshalb in der ersten Zeit keine Aufmerksamkeit geschenkt.
Ganz zum Schluss der Projektarbeit kam aber doch der Wunsch wieder hoch, wenigstens zur Ausgabe auch ASCII-Zeichen verwenden zu dürfen. Damit würden sich dann wenigstens schon mal kleinere Zeichnungen oder Status-Meldungen ausgeben lassen.

Unsere String-Implementation ist ziemlich banal und würde einer Erweiterung keinesfalls standhalten - wichtig war zum Zeitpunkt der Implementierung, dass das Feature wie gewünscht funktioniert, keine weiteren Probleme verursacht und keinen Zeitdruck für die Entwickler erzeugt. Wir werden uns im Anhang darüber unterhalten, wie eine „echte“ String-Implementation gemacht werden müsste.

Neben der offensichtlichsten Änderung - in der Sprachdefinition wird ein weiterer Typ `string` aufgeführt - und einigen Fehler-Abfang-Routinen - wegen der internen Repräsentation sind Operatoren auf Strings verboten - betreffen die Änderungen hauptsächlich die Prozedur `Factor`.

Zum einen müssen Strings deklariert werden können. Ohne strukturelle Erweiterungen des Codes geht die Verarbeitung von Zeichenketten nur atomar, das heisst, Strings werden vom grinj-Compiler als Aneinanderreihung von einzelnen Zeichen verstanden. Ziel der Änderung war, dass der Compiler für jedes eingelesene Zeichen eines deklarierten Strings einen `CONST`-Opcode, gefolgt von einem `LOAD`-Opcode generiert, um jeweils ein Zeichen in ein virtuelles Register der VM zu laden. Wenn die Zeichenkette abgeschlossen ist, wird ein weiteres Element vom Typ `endstring` in den Parser-Baum gepackt.
Beim `WRITEC`-Opcode geht's umgekehrt: Der Compiler soll den Code so generieren, dass nach einem angegebenen zu ladenden Register solange die folgenden Register

ausgegeben werden, bis im Parser-Baum ein Objekt vom Typ `endstring` auftritt.

Das Deklarieren und Schreiben von Strings funktioniert so ziemlich gut. Für die einfache Funktionalität war neben dem Erweitern der Prozeduren im ATG-File nur noch eine weitere Variable in der `Parser`-Klasse nötig, die festhält, ob sich der Baum gerade hinter einer `WRITEC`-Anweisung befindet oder nicht.

Die beteiligten Klassen

Für eine genauere (Code-)Sicht der Klassen verweisen wir auf die `DoxyGen`-Klassen-Dokumentation. Dort sind alle Klassen, deren Variablen und Funktionen aufgelistet und nochmals im Zusammenhang dargestellt.

Parser und Scanner mit Hilfsklassen

Die Klassen `Parser` und `Scanner` geben nicht mehr viel Arbeit. Zusammen mit den Hilfsklassen `Errors`, `BitArray`, `Token` und `Buffer` werden sie direkt von CocoR aus dem ATG-File heraus generiert. Die Klassen sind so sofort in eigenen Projekten einsetzbar.

`Errors` enthält einen Counter, mit dem der Compiler beim Durchlaufen überprüft, ob ein Fehler aufgetreten ist. Ausserdem sind Funktionen enthalten, die es erleichtern, Fehlermeldungen direkt auszugeben. Wenn der Wunsch besteht, die Fehlerausgabe in einen anderen Kanal umzuleiten - beispielsweise in eine anzugebende Datei oder in einen Socket - , so ist die Änderung dazu in dieser Klasse anzubringen.

Interessant ist vor allem noch die `Token`-Klasse. Hier handelt es sich um ein Datenpaket, das nicht nur den gerade gültigen Token-Typ (Opcode und Art des Opcodes oder Variable) enthält, sondern daneben auch noch Angaben beinhaltet, wo im Sourcecode der Token gelesen worden ist.
Wir nützen diese Struktur im `CodeGen`-Objekt aus, um daraus die Debug-Informationen zu gewinnen, die an jede Output-Datei (bei Bedarf) angehängt werden.
Die `Token`-Objekte sind als verkettete Liste organisiert; jede Instanz hat wiederum einen Pointer auf eine weitere `Token`-Instanz eingebaut.
Unser Ziel war es weiterhin, keine Änderungen direkt im Sourcecode der Parser- und Scanner-Klassen zu machen. Weil dieser Code direkt von CocoR aus dem Input-File generiert wird, wäre jede Änderung bei einer Erweiterung der Sprache hinfällig und müsste mühsam von Hand neu hinzugefügt werden. Dort, wo wir anfangs Änderungen in `Parser.cpp` vorgenommen hatten, haben wir uns darum bemüht, diese Änderungen auch im .atg-File zu integrieren. Beim momentanen Projektstand ist es damit so, dass ein erneutes Generieren der Klassen zu lauffähigem Code führt.

SymbolTable

Die Klasse `SymTab` beinhaltet die Opcode-Baumstruktur, die mit Hilfe von `Parser` und `Scanner` aus dem Sourcecode generiert wird. Dabei handelt es sich um eine einfach verkettete Liste aus `Obj`-Instanzen.
Die Klasse `SymTab` und ihre Implementierung wurde direkt vom `Taste`-Beispiel

übernommen und wird hier deswegen auch nicht mehr weiter diskutiert.

CodeGen und abgeleitete Klassen

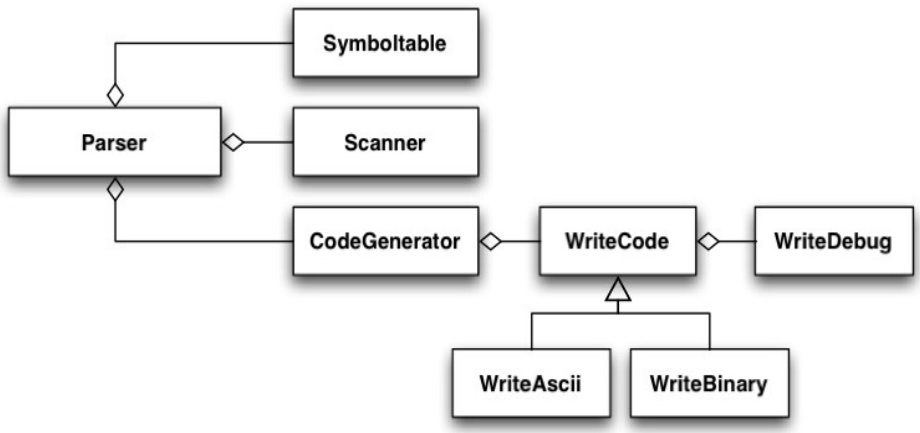
Auch bei der Klasse `CodeGen` handelte es sich zu Beginn um eine C++-Kopie der `CodeGen`-Klasse aus dem `Taste`-Beispiel. Im Laufe des Projekts hat sie sich aber ziemlich verändert.
In dieser Klasse wird während des Parsens eine verkettete Liste von Opcode-`Token` generiert, die dann abschliessend wieder ausgegeben werden können.

Das `Taste`-Beispiel unterstützt keine Ausgabe des Kompilats, da es sich um einen Interpreter handelt. Eine der erste Änderungen betraf deshalb das Hinzufügen von Methoden, die das Schreiben einer Binär-Datei ermöglichen. Dazu gehört das Erstellen eines RFC-konformen Headers sowie das anschliessende Schreiben der Opcodes in binärer Form.

Nach und nach hat sich die Klasse weiter verändert. Eine weitere Methode zum Hinzufügen der Debug-Informationen wurde geschrieben, und bald einmal kam auch der Wunsch auf, zu Testzwecken das Kompilat nicht nur in binärer Form sondern auch in einer Art Pseudo-Assembler auszugeben. Ausserdem mussten wir die `put()`- und `emit()`-Methode des ursprünglichen `Taste`-Codes anpassen, damit sie zu unserem eigenen Dateiformat mit einem Byte Opcode und vier Bytes Parameter-Werte passten.

Die Klasse war mittlerweile ziemlich angewachsen. Das Anti-Pattern „Blob“ war klar sichtbar, Code war dupliziert vorhanden, und `if`- und `switch`-Anweisungen häuften sich - alles in allem klare Anzeichen für ein nötiges Refactoring.

Wir haben uns also dazu entschlossen, die File-Operationen aus der `CodeGen`-Klasse herauszulösen und mittels Vererbung den Code für Binär- und Ascii-Output möglichst kompakt, klar und übersichtlich zu halten. Unser erstes neues Klassen-Design sah in etwa so aus:

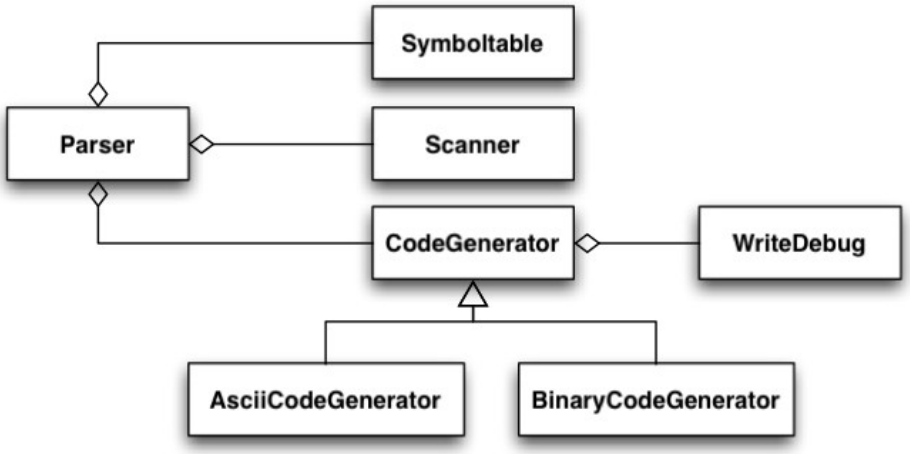


`CodeGen` aggregiert je nach Aufruf-Parameter des Compiler-Binaries entweder eine Instanz von `WriteBinary` oder von `WriteAscii`; ausserdem entscheidet ein weiterer

Aufruf-Parameter darüber, ob eine Instanz von `WriteDebug` erstellt und in die Writer-Instanz aggregiert wird.

Funktioniert hat's, keine Frage. Unser Problem dabei war: Die `Token`-Struktur sass nach wie vor in der Klasse `CodeGen`. Wenn `CodeGen` der Writer-Instanz den Befehl zum Schreiben der Datei gab, musste diese Instanz während des Schreibvorgangs andauernd auf die `Token`-Struktur in `CodeGen` zugreifen. Eine Lösung wäre das Kopieren der gesamten verketteten Liste und Übergabe der Kopie an die Writer-Instanz gewesen - fanden wir hässlich. Eine andere Lösung wäre die Übergabe eines Startpointers an die Writer-Instanz gewesen - noch hässlicher.

Schlussendlich haben wir uns auf die folgende Struktur besonnen:



Eine Klasse wurde dabei wieder gestrichen. Die Funktionalität von `WriteCode` - vornehmlich eine Schleife zur Generierung des Codes - wurde zurück in `CodeGen` integriert und die spezialisierten Writer-Klassen direkt von `CodeGen` abgeleitet.
Damit haben wir einerseits die Modularität des Strategie-Patterns beinahe zurückgewonnen, andererseits vermeiden wir das Problem mit der `Token`-Struktur, die das Anti-Pattern „Neid“ hervorgerufen hatte. Damit können wir leben.

Verwendung des Compilers

Das Verwenden des Compilers ist ziemlich einfach. Nach erfolgreicher Generierung der `Parser`- und `Scanner`-Klassen mit CocoR reicht eine Kompilierung der `src/*`-Dateien aus, und das Binary des grinj-Compilers wird erstellt.

Der Aufruf erfolgt in der Kommandozeile mittels
`grinjc [-a -d] SOURCEFILE OUTPUTFILE`

Ohne weitere Optionen erstellt grinjc ein binäres File ohne Debug-Informationen, das bei installierter VM direkt durch ein Aufrufen des Namens (oder per Doppelklick) gestartet werden kann.
Mit der Option `-a` wird als Output ein ASCII-File generiert, in dem die Opcodes als Pseudo-Assembler aufgeführt sind. Damit lassen sich eventuelle Fehler im Compiler -

oder auch einfach nur der genaue Programmablauf - besser verfolgen.
Die Option `-d` schaltet die Debug-Informationen hinzu. Wenn der interne Debugger der VM verwendet werden soll, ist diese Option zwingend nötig.

Arbeitsweise / Rückblick

Wie schon zu Beginn erwähnt wurde, ist das Erstellen eines eigenen Compilers nicht ganz einfach. Nach den Basisvorträgen, welche wir durchgeführt haben, um uns einen Überblick über die Projekt-Thematik (Compiler, VM, Debugger, Editor) zu verschaffen um uns dann für eines der Teilgebiete entscheiden zu können, sind wir als erstes auf die Suche nach Coco/R - Unterlagen gegangen.
Das User-Manual, welches dem Coco/R Sourcecode beilag, war unser erster grosser Fang. Wie sich aber beim mehrmaligen Durchblättern und Lesen herausstellte, bot dieses Dokument keine praktische Einführung in den Compilerbau. Es fiel uns teilweise schwer, Gelesenes in einen Kontext zu bringen, es fehlte eine Übersicht um die im Dokument erwähnten Sachverhalte richtig einzuordnen um daraus Erkenntnisse zu ziehen. Das ebenfalls gefundene Buch zum Thema Compilerbau war hier mit seinen kurzen Beispielen eher hilfreich, aber dessen Umfang zu gross, um es in gegebener Zeit durchzuarbeiten. Schliesslich ging es darum, möglichst bald eine einfache, aber funktionierende Version des Compilers dem VM-Team zur Verfügung zu stellen.
Wir wählten einen pragmatischen Ansatz. Der fertige Compiler/Interpreter für die einfache Sprache "Taste", welche im User-Manual als Beispiel-Compiler aufgeführt ist, war unser Ausgangspunkt. Zuerst schrieben wir die Klassen SymbolTable und Codegenerator in C++ um (das Beispiel war in C# gegeben). Die Sprachdefinition (EBNF) blieb, bis auf den Namen, unverändert. Jedoch musste das ATG-File, aus welchem Coco/R den Parser und Scanner erstellt, noch für C++ angepasst werden. Ausserdem wollten wir mit unserem Compiler "nur" einen Binary-Code generieren und nicht den Taste-Sourcecode gleich interpretieren und ausführen. Aus diesem Grund wurde die Funktionalität der Klasse CodeGenerator angepasst. Eine, wie wir später bemerkten, nicht ganz fehlerfreie Version des Compilers hatten wir schliesslich zum Laufen gebracht. Ein erstes Erfolgserlebnis, welches uns wieder Mut schöpfen liess! Sie war geboren, die Compiler Blackbox, welche auf magische Art und Weise Sourcecode in Binarycode umwandeln konnte. Natürlich ist diese Aussage etwas überspitzt, aber wir waren zu Beginn noch weit davon entfernt, den Compiler zu verstehen (inzwischen sind wir ein Stück näher gerückt).
Die Weiterentwicklung unseres Grinj-Compilers geschah teilweise überlegt und teilweise durch Versuch und Irrtum. Wir arbeiteten immer noch mit einer Blackbox, die wir von Mal zu Mal ein Stückchen besser kennen lernten. Mit jeder neuen Anforderung an den Compiler und dem entsprechenden Eingriff in den Code, tasteten wir uns heran. Zuweilen befanden wir uns an der Grenze zur Überforderung und der Gedanke daran, dass wir dieses Projekt vielleicht lieber nach der Compiler-Vorlesung des nächsten Semesters hätten durchführen sollen, nagte an unserer Motivation. Gegen Ende des Projekts haben wir uns sogar daran gemacht, die Funktionalität der Sprache zu erweitern. Wobei ich an dieser Stelle einräumen muss, dass nicht alle Mitglieder des Compilerteams im Stande waren, sich diese Fertigkeit anzueignen.
Es bleibt zum Schluss zu sagen, dass in unserem Falle der praktische Weg zum Ziel geführt hat und dass wir, obschon sehr vieles gelernt, noch immer ungeklärte Fragen (die einen mehr, die anderen weniger) mit uns tragen. Wir sind jedoch zuversichtlich, diese im Laufe des kommenden Semesters selber beantworten zu können.

VM

Versionsübersicht

1.0

In GRINJ-VM v1.0 wurden die Basis-RFCs 002 (Opcodes), 003 (FileDefinition) und 004 (Debugger) implementiert. Alle dort definierten Strukturen und Vorgänge gelten für dieses Dokument und werden in der folgenden Source-Doku nicht mehr speziell erwähnt.

1.1

Kurz nach der Präsentation des Projektes ergaben sich ein paar Wünsche benzüglich der Möglichkeiten in der Programmiersprache „GRINJ". Da die VM v1.0 im Wesentlichen nichts weiter kann als Integer-Operationen, werden Erweiterungen in Richtung Hardware/Betriebsystem gefordert. Mit einem einfachen Treiber-Interface soll GRINJ ein minimales „Tor zur Welt" verpasst werden. Die Erweiterungen sind in folgenden Dokumenten spezifiziert:

- RFC 005: OP-Codes
- RFC 006: Debug-Protokoll
- RFC 007: Treiber-Interface
- RFC 008: Binary-Fileformat

Die Erweiterungen betreffen

- Anbindung rekonfigurierbarer Ressourcen („Drives")
- Bit-Operationen
- Sleep-Funktion

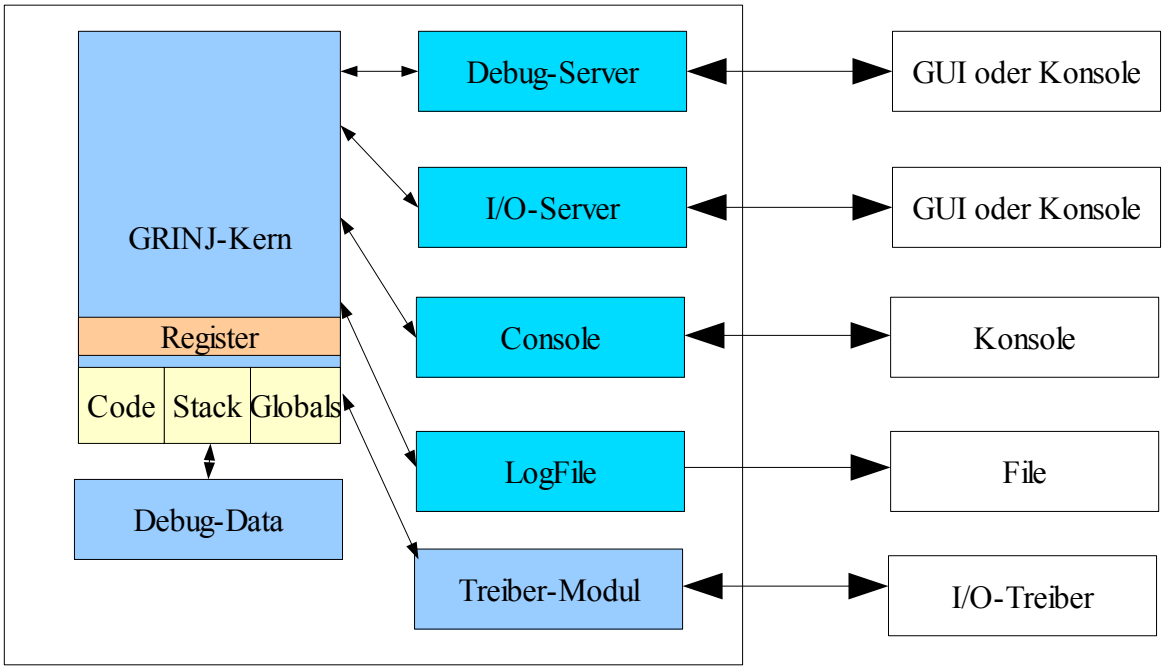
Features

Die GRINJ VM 1.0 implementiert folgende Features:

- Hardware-Emulation mit Code-, Daten und Stack-Speicher, inkl. Register
- Debug-Funktionalität mittels Sockets
- I/O via Sockets
- LogFile-Funktionalität

Kurzübersicht

Die VM hat einen modularen Aufbau und besteht aus der eigentlichen Virtuellen Maschine sovie diversen HilfsModule wie z.B. I/O, Threads oder Protokollierung:



Die VM kann also nicht nur GRINJ-Binaries ausführen, sondern auch über diverse Schnittstellen mit der Aussenwelt kommunizieren. In den nächsten Kapiteln wird der Reihe nach die Bedienung, der Aufbau der Virtuellen Maschine sowie das Software-Design erklärt.

User-Manual

VM

Die VM ist eine reine Konsolenanwendung und ist dementsprechend einfach in der Bedienung. Eine GRINJ-Binary wird ausgeführt, in dem der Dateiname der VM als Parameter angegeben wird:

```
$ grinj sample-prim
```

Auf UNIX-Systemen wird GRINJ auf /usr/local/bin installiert. Bei solchen Plattformen kann ein GRINJ-Programm auch direkt (wie ein normales Programm oder ein Shellscript) gestartet werden:

```
$  ./sample-prim
```

Die VM besitzt einige wenige Parameter:

Parameter	***Beschreibung***
--console	Die Programm I/O soll über die Konsole geführt werden (Standard)
--debug-server=xxxx	Erstellt eine Debug-Konsole auf dem Port „xxxx“. Für die GUI ist dies üblichweise 50005
--io-server=xxxx	Erstellt einen Input/Output-Server auf dem Port „xxxx“. Die Verwendung dieses Parameters schliesst die gleichzeitige Verwendung einer Konsole aus.
--file=xyz	Erstellt eine Log-Datei „xyz“ und protokolliert darin den VM-Output.

Die Default-Einstellung ist „--console“. Wird eine Log-Datei gewünscht, sollte unbedingt mit angegeben werden, über welche Art der Programm-Input erfolgen soll (z.B. --console oder –io-server=50006):

```
$ grinj sample-prim --io-server=6000 --file=/tmp/grinj.log
```

Bei diesem Beispiel erfolgt die Eingabe über den Socket-Server auf Port 6000, die Ausgabe ebenfalls auf diesen Socket und zusätzlich in die Datei /tmp/grinj.log.

Achtung: Bei einem I/O-Server muss zwingend connected werden. Bis die Verbindung steht, ist die VM blockiert.

Debugger

Der Debugger kann direkt als Konsole (Socket-Verbindung) bedient werden. Der komplette Funktionsumfang plus die Protokoll-Definition befinden sich in den RFCs im Anhang. Nachfolgend wird deswegen nicht das gesamte Protokoll dokumentiert, sondern nur der Umgang mit Debugger und Konsole. In der VM v1.0 existieren erst einige rudimentäre Funktionen, so dass z.B. der Stack nur als String ausgegeben werden kann. Im Normalbetrieb mit der GUI wird dieser String tabellarisch dargestellt. Trotzdem eröffnet eine Debug-Konsole viele Möglichkeiten, wie z.B. das Fernsteuern der VM via einer Scriptsprache.

Eine Debug-Session ohne GUI wird wie folgt erstellt:

1. In einer ersten Konsole muss die VM gestartet werden :
   „*grinj –debug-server=5000*“ (der Port 5000 ist dabei natürlich variabel).
2. In einer zweiten Konsole kann nun auf den offenen Port connected werden:
   „*telnet localhost 5000*“
3. Über die zweite Konsole kann nun die VM gesteuert werden. Die Programm-I/O erfolgt bei diesem Beispiel in der ersten Konsole, also dort wo die VM gestartet wurde. Eine typische Kommunikation sieht in etwa wie folgt aus:

```
 user@host:~$ telnet localhost 500
Trying 127.0.0.1...
Connected to localhost.localdomain
load sample/bin/sample-prim
00 02 9 ./src/sample-prim.grinj
run
00 03 9 ./src/sample-prim.grinj
stop
00 05 23 ./src/sample-prim.grinj
cont
00 03 23 ./src/sample-prim.grinj
stop
00 05 30 ./src/sample-prim.grinj
quit
00 05 30 ./src/sample-prim.grinj
Connection closed by foreign host
user@host:~$
```

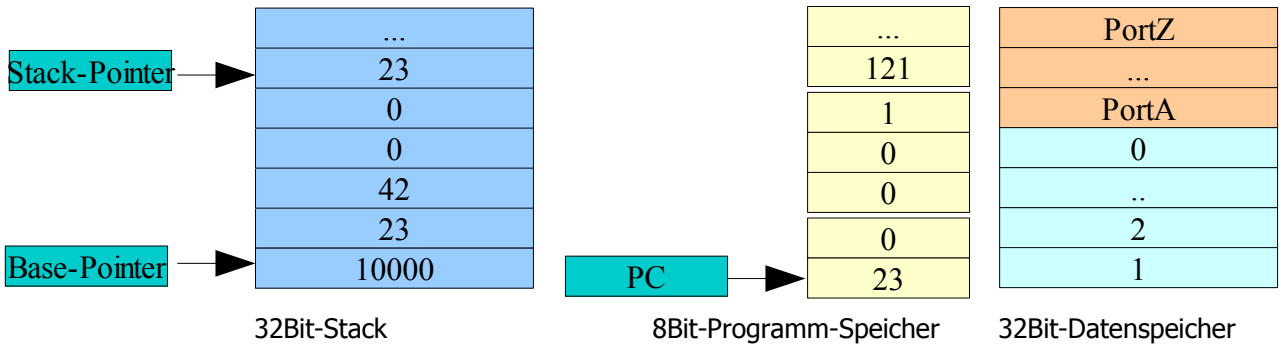
```
Rot: Kommando
Grün: Antwort der VM
```

Die genaue Bedeutung sollte dem Anhang entnommen werden. Dort ist auch die Übersicht zu finden, welche Kommandos es gibt und welche Return-Werte sie liefern. Trotzdem sollen kurz zwei Beispiele erläutert werden:

- `00 02 9 ./src/sample-prim.grinj` auf „load“ bedeutet, dass das Kommando angenommen wurde (00), die VM im Zustand READY (02) ist und die Position des Program-Counters der Zeile 9 in der Datei „./src/sample-prim.grinj“ entspricht.
- `00 05 23 ./src/sample-prim.grinj` auf „stop“ bedeutet, dass das Kommando angenommen wurde (00), die VM im Zustand „STOPPED“ (05) ist und die aktuelle Position des Program-Counters der der Zeile 23 in der erwähnten File entspricht.

Hardware-Emulation

Die VM emuliert ein Stack-Basierendes System mit getrenntem Code- und Daten-Speicher und drei Registern:

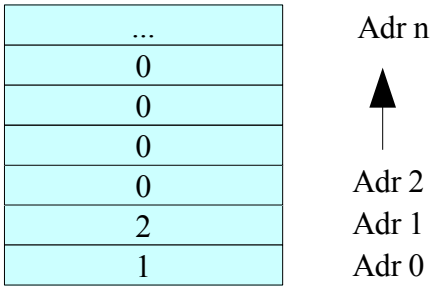


Alle Register sind ebenfalls 32Bit breit. Diese Architektur erlaubt nun folgende Funktionen:

- Stack-Basierte Operationen auf lokale oder temporäre Variablen
- Globale Variablen mit Initialwerten in Datenspeicher
- Sprünge
- Funktionen mit Parametern (Parameter nach RFC002 noch nicht gefordert)

Arithmetische Operationen werden auf 32Bit „signed Integer" beschränkt (s. auch RFC002).

Alle Daten-Zugriffe erfolgen linear als 32Bit-Adresse:



Ports

Im Bereich der globalen Variabeln existieren einige fix vergebene Adressen: im Bereich 1000-1025 wurden die Ports „PortA" - „PortZ" untergebracht. Diese globalen Variabeln besitzen die spezielle Eigenschaft, dass I/O-Treiber eingeklinkt werden können. Ein Treiber ersetzt die Speicherzelle komplett. Jeder Zugriff darauf wird über die I/O-Methoden des Treibers abgewickelt.

Compiler können die Ports als normale, globale Variabeln behandeln, müssen jedoch die speziellen, fixen Adressen 1000-1025 dazu vergeben.

Weitere Infos zur Treiberanbindung folgen im VM-Abschnitt „Ressourcen".

Stack

Grundsätzlich können Arithmetische Funktionen und Vergleiche nur auf Daten im Stack angewendet werden. Möchte man eine Arithmetische Funktion auf globale Daten anwenden, so müssen diese zuerst in den Stack gelesen werden. Danach kann eine beliebige Arithmetische Funktion auf die geladenen Daten angewendet werden. Das Resultat liegt anschliessend wieder auf dem Stack und muss entsprechend verschoben werden.

Der Stack und seine Handhabung, soll anhand einer einfachen Addition dargestellt werden. Wir nehmen an, dass bereits 2 globale Variablen a und b existieren.

```
a = a + b;      // LOADG 0
                // LOADG 1
                // ADD
                // STOG 0
```

Globale Daten:

Adr 0:	20	=> VAR A
Adr 1:	10	=> VAR B

Vorgang auf dem Stack:

Nach dem ersten Aufruf vo LOADG 0 liegt die Variable A (= Adresse 0) zu oberst auf dem

StackPointer -->	20

Stack:

Nach dem zweiten Aufruf von LOADG 1 wird die zweite Variable B an der Adresse 1 auf den Stack gelesen. Die zweite Variable B liegt nun an oberster Stelle im Stack und die Variable A liegt direkt unter der Variablen B:

StackPointer -->	10
	20

Wird nun der Befehl ADD auf dem Stack ausgeführt, werden zuerst die zwei obersten Variablen aus dem Stack heraus gelesen, addiert und anschliessend wird das Ergebnis wieder auf den Stack zurück geschrieben.

StackPointer -->	30	=> Resultat

Letzt endlich wird das Ergebnis, mittels STOG 0, wieder aus dem Stack an die richtige Speichersteller verschoben werden:

Adr 0:	30	=> VAR A
Adr 1:	10	=> VAR B

Der Stack wächst mit jedem Element, welches eingelesen wird nach oben an. Um eine arithmetische Anweisung auf dem Stack ausführen zu können müssen mindestens 2 Werte eingelesen werden. Einzige Ausnahme bildet der NEG-Befehl. Dieser wird immer auf den obersten Eintrag im Stack angewendet, und schreibt das Ergebnis wieder in den Stack zurück. Damit können diverse Arithmetische Operationen auf dem Stack ausgeführt werden, bis nur noch ein Wert, das Endresultat, auf dem Stack liegt.

Diesen Sachverhalt soll anhand einer einfachen Rechnung veranschaulicht werden:

```
A = A + B + C;
```

Wir nehmen an, dass die Variablen A, B und C bereits definiert sind, und in den Stack eingelesen wurden. Da bei der Addition die Reihenfolge der Summanden keinen Einfluss auf das Resultat hat, ist die Reihenfolge im Stack egal, achten Sie darauf, das bei einer Subtraktion das Vorzeichen eine Rolle spielt:

StackPointer -->	C
	B
	A

Nach der ersten Addition wird B + C ausgerechnet und das Resultat wieder auf den Stack zurück gelegt:

StackPointer -->	B + C
	A

Nach der zweiten Addition wird A zu (B + C) dazu addiert. Im Stack liegt nun das Resultat der beiden Additionen:

StackPointer -->	A + (B + C)	=> Resultat

Nun kann das Resultat aus dem Stack an eine beliebige Variable (Adresse) verschoben werden.

Globale Variablen

Globale Variablen werden im Datenspeicher abgelegt. Der Compiler bestimmt selbst, wo ein Wert „x" abgelegt werden soll. Die Ankündigung der Variable selbst ist für die VM irrelevant:

```
int a
int b;

a = 10;  // const 10
         // stog 0
b = 20;  // const 20
         // stog 1
```

Das Compilat enthält also lediglich:

```
020 000 000 000 010
022 000 000 000 000
020 000 000 000 020
022 000 000 000 001
```

Globale Variablen werden also direkt adressiert. Im obigen Fall entschied der Compiler, der Variabel „a" die globale Adresse 0 und „b" die Adresse 1 zu zuordnen.

Der Compiler muss also "Buchhaltung" über die globalen Variablen führen. Bei der Ankündigung muss eine freie, globale Adresse vergeben werden und den Zugriff jeweils über diese abgewickelt werden.

Initialisierte Variablen müssen gem. RFC003 ausserdem in der Datei angekündigt werden.

So ergibt der folgende Pseudo-Code zwar keine Maschinen-Anweisungen (es wird ja nichts ausgeführt), aber eine Ankündigung in der Binary-File:

```
int a;
int b=3;
int c=4;
```

ergibt (Unter der Annahme, dass sich der Compiler bei "b" und "c" für die Adressen 1 und 2 entscheidet) die Binary (GLOBALS-Ausschnitt):

```
255
000 000 000 001 000 000 000 003
000 000 000 002 000 000 000 004
255
```

Funktionsaufrufe

Funktionsaufrufe werden komplett über den Stack abgewickelt. Da nicht nur der

eigentliche Aufruf, sondern auch die lokalen Variablen über den Stack behandelt werden, werden nun die Funktionsaufrufe vor anderen Operationen wie Arithmetik, I/O, Sprünge usw. erklärt:

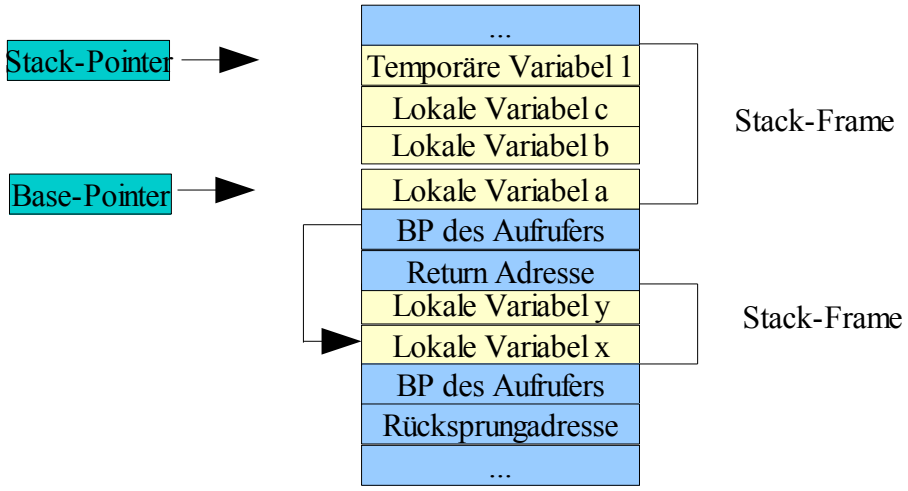
Die Daten einer Funktion werden komplett auf den Stack gelegt. Als "Daten" werden hier bezeichnet:

- die Rücksprungadresse zur aufrufenden Funktion
- der Base-Pointer, welcher ein Stack-Frame definiert (folgt in diesem Kapitel)
- die Parameter und der Returnwert der Funktion (in dieser Version noch nicht behandelt)
- die lokalen Variablen der Funktion
- temporäre Daten aus z.B. arithmetischen Operationen oder I/O

Aber eines nach dem Anderen. Schauen wir uns erstmal die Stackaufteilung an:

Stackframes

Eine Funktion erzeugt beim Eintritt jeweils ein neues "Stack-Frame". Dieses dient dazu, die lokalen Variablen direkt im Stack Adressieren zu können. Als Markierungen dieses Frames dienen jeweils Base-Pointer und Stack-Pointer. Der Base-Pointer markiert die untere Grenze des Frames, der Stack-Pointer die obere:



Ein Frame kann also wachsen, hat jedoch immer die minimale Grösse der Anzahl lokalen Variabeln:

```
{
  int a;
  int b,c;
}
```

Diese Funktion ergäbe also eine minimale Framegrösse von "3"

Frameerzeugung

Ein Stack-Frame wird erzeugt mittels "ENTER n", wobei n die minimale Framegrösse (=Anzahl lokale Variabeln) ist.

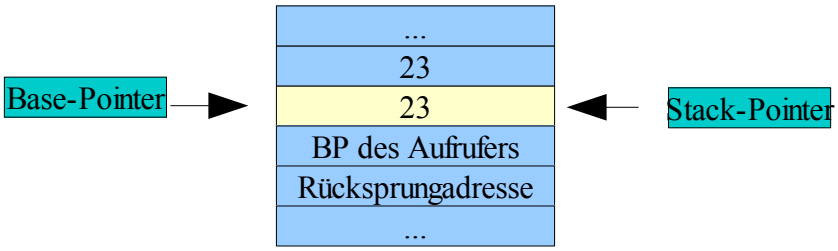
Dazu gleich ein Beispiel; erzeugt wird folgende (vollkommen sinnfreie) Funktion:

```
Fnc()       // ENTER 1
{
  int a;
  a = 23;   // CONST 23
            // STO 0
}
```

Daraus erzeugt also der Compiler folgenden Code:

```
122 000 000 000 001
020 000 000 000 023
023 000 000 000 000
```

Der daraus entstandene Stack sähe dann so aus:



Die Zahl "23" liegt noch im (temporären) Bereich oberhalb des Frames (mittels "const 23" dort hin geschrieben).

Dass SP und BP auf die gleiche Stelle zeigen ist (solange der Compiler kein "pop" zu viel einpflanzt) unkritisch.

Dann wollen wir als nächstes doch einmal einen kompletten Aufruf implementieren:

Call

Der eigentliche Funktionsaufruf beschränkt sich auf das Sichern und Wiederherstellen der Basis-Register PC, BP. Jeder Funktionsaufruf folgt daher immer dem gleichen Schema:

1. **CALL a**: Sichern der nachfolgenden Adresse auf dem Stack und Sprung in die Funktion
2. **ENTER n**: Erstes Kommando in der Funktion: Frame erzeugen
3. .... Funktionsanweisungen
4. **LEAVE**: Stack wiederherstellen, Abbrechen des Frames
5. **RET**: Rücksprung durch wiederherstellen des PC

Dazu gleich ein (nun komplettes) Beispiel:

```
fnc()       // ENTER 1
{
  int a;
  a = 23;   // CONST 23
            // STO 0
}           // LEAVE
            // RET

main()      // ENTER 0
{
   func();  // CALL @fnc
}           // LEAVE
            // RET
```

Der Compiler legt nun z.B. die Funktion "main" auf Adresse 1000 und "fnc" auf 2000:

```
 Adr nachfolgendes Kommando
--------------------------
1000 122 000 000 000 000
1005 120 000 000 007 208
1010 123
1011 121
...
2000 122 000 000 000 001
2005 020 000 000 000 023
2010 023 000 000 000 000
2015 123
2016 121
```

Beim letzten RET nimmt der PC einen ungültigen Wert an (0) und die VM beendet das Programm schliesslich.

Parameter und Return-Values

Parameter und Return-Werte für Funktionsaufrufen werden von der vorliegenden Version 1.0 unterstützt. Eine Parameter-Übergabe ist nichts weiter als ein Zwischenspeichern der Werte auf dem Stack. Die Parameter werden wie lokale Variabeln behandelt:

```
fnc(int x)     // ENTER 1
{
               // STO 0
  print x;     // LOAD 0
               // PRINT
}              // LEAVE
               // RET

main()         // ENTER 0
{
   func(23);   // CONST 23
               // CALL @fnc
}              // LEAVE
               // RET
```

Ein Return-Wert könnte vor „LEAVE" auf den Stack gespeichert und vom Aufrufer restauriert werden. Diese beiden Features werden vom Compiler bisher nicht unterstützt.

Arithmetische Operationen

Arithmetische Operationen können nur auf dem Stack ausgeführt werden. Die VM von Grinj kennt folgende arithmetische Operationen:

- ADD
  Addiert die ersten zwei Elemente auf dem Stack und schreibt das Ergebnis wieder auf die oberste Stelle des Stacks zurück.
- SUB
  Subtrahiert das oberte Element im Stack vom 2 obersten Element. Anschliessend wird das Ergebnis wider auf den Stack zurück gelegt.
- DIV
  Dividiert das zweite Element im Stack durch das oberste Element. Nach der Division wird das Ergebnis auf den Stack zurück geschrieben.
  ACHTUNG: Divisionen durch 0 sind nicht erlaubt und müssen vermieden werden!
- MUL
  Multipliziert das oberste Element auf dem Stack mit dem zweitobersten. Das Resultat wird anschliessend wieder zurück auf den Stack gelegt.
- NEG
  Vorzeichen des obersten Elementes auf dem Stack wird getauscht. Das Ergebnis wird wieder auf den Stack gelegt.

Standard-Input/-Output

In der VM v1.0 können lediglich Integers eingelesen und ausgegeben werden.

Als Erstes wird nun das Einlesen und Ausgeben über eine globale Variable demonstriert:

```
VAR a;

read a;      // READ
             // STOG 0
write a ;    // LOADG 0
             // WRITE
```

Annahme: der Compiler hat der Variable "a" die globale Adresse 0 zugewiesen.

Handelt es sich um eine lokale Variable, geht das ganze über den Stack:

```
{
  VAR a;

  read a;    // READ
             // STO 0
  write a;   // LOAD 0
             // WRITE
}
```

Bei der lokalen Variante entstünde also der GRINJ-Maschinen-Code (Annahme: Code startet ab Adr. 2000):

```
 Adr nachfolgendes Kommando
---------------------------
...
2000 100
2001 023 000 000 000 000
2006 021 000 000 000 000
2011 101
...
```

Kontrollkonstrukte

Ohne Kontrollkonstrukte liessen sich kaum sinnvolle Programme erstellen. Deshalb folgen nun Beispiele zu IF und SWITCH:

IF/ELSE

Als Entscheidungsgrundlage dient jeweils das letzte Element auf dem Stack. Normalerweise wird dieses durch eine logische oder -eher selten- durch eine arithmetische Operation erstellt. Im folgenden Beispiel gibt der Benutzer eine Zahl ein. Wenn die Zahl grösser als 7 ist, wird „1" ausgegeben, sonst „0". Dieses sinnvolle und nützliche Programm sieht also wie folgt aus:

Nach dem üblichen Read folgt der IF mit einem Vergleich und zwei JMP-Anweisungen.

Falls das Ergebnis 0 (also FALSE) ergibt, springe direkt in den Else-Block. Der Then-Block hingegen überspringt mit einem unbedingten JMP über den ELSE-Block.

```
{
  int a;

  read a;     // READ
              // STO 0
  If (a > 7) // LOAD 0
              // CONST 7
              // GTR
              // FJMP @else
  then
     a = 1;   // CONST 1
              // STO 0
              // JMP @Write
  else
     a = 0;   // CONST 0
              // STO 0
  end

  write a ;   // LOAD 0
              // WRITE
}
```

Der Vergleich (GTR) kann auch gegen den LSS getauscht werden. In diesem Fall müssen einfach die Elemente in der anderen Reihenfolge auf den Stack gelegt werden. Die Reihenfolge und entsprechendem Vergleich (GTR oder LSS) bleibt somit dem Compiler überlassen.

Im Beispiel wird GTR (also „a grösser 7", nicht „7 kleiner a") gewählt, weil das Lesen des OP-Codes so logischer ist.

Nachfolgend also das komplette Beispiel:

```
 Adr nachfolgendes Kommando
---------------------------
1000 122 000 000 000 001   // ENTER 1
1005 100                   // READ
1006 023 000 000 000 000   // STO 0
1011 020 000 000 000 007   // CONST 7
1016 021 000 000 000 000   // LOAD 0
1021 060                   // GTR
1022 081 000 000 004 018   // FJMP 1042
1027 020 000 000 000 001   // CONST 1
1032 023 000 000 000 000   // STO 0
1037 080 000 000 004 034   // JMP 1058
1042 020 000 000 000 000   // CONST 0
1048 023 000 000 000 000   // STO 0
1053 021 000 000 000 000   // LOAD 0
1058 101                   // WRITE
1059 123                   // LEAVE
1060 121                   // RET
```

Das ist also z.B. ein „IF“. Selbstverständlich existieren (wie erwähnt z.B. mit GTR) noch weitere Möglichkeiten.

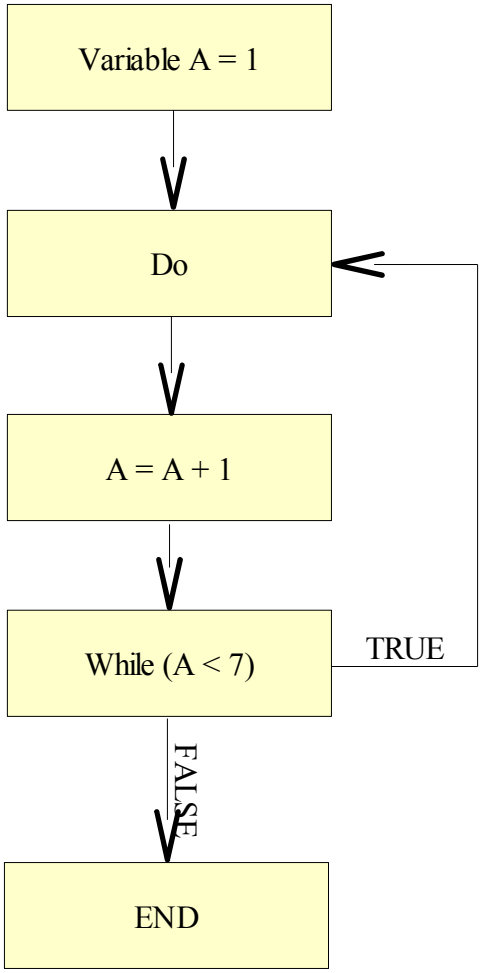
Schleifen

Schleifen setzen sich aus zwei verschiedenen Arten von Sprüngen zusammen. Es gibt Schleifen, welche ihren Inhalt unbedingt Ausführen. D. h. unbedingte Schleifen werden mindestens einmal durchlaufen, während bedingte Schleifen nicht zwingend durchlaufen werden müssen.

Bei den folgenden Beispielen wird auf die Angabe von Debug-Informationen in der Binärdatei verzichtet, da diese Informationen nicht zum Verständnis von Schleifen beitragen.

Bedingte Sprünge

Ist typischerweise eine do-while Schleife. Schematische Darstellung eines bedingten Sprunges:



Umsetzung

Nachfolgend wird ein sehr einfaches Beispiel für einen bedingten Sprung erläutert. Die do-while Anweisung soll zu einer Zahl 1 addieren bis diese einer eingelesenen Zahl entspricht. Das Assemblerprogramm sieht wie folgt aus:

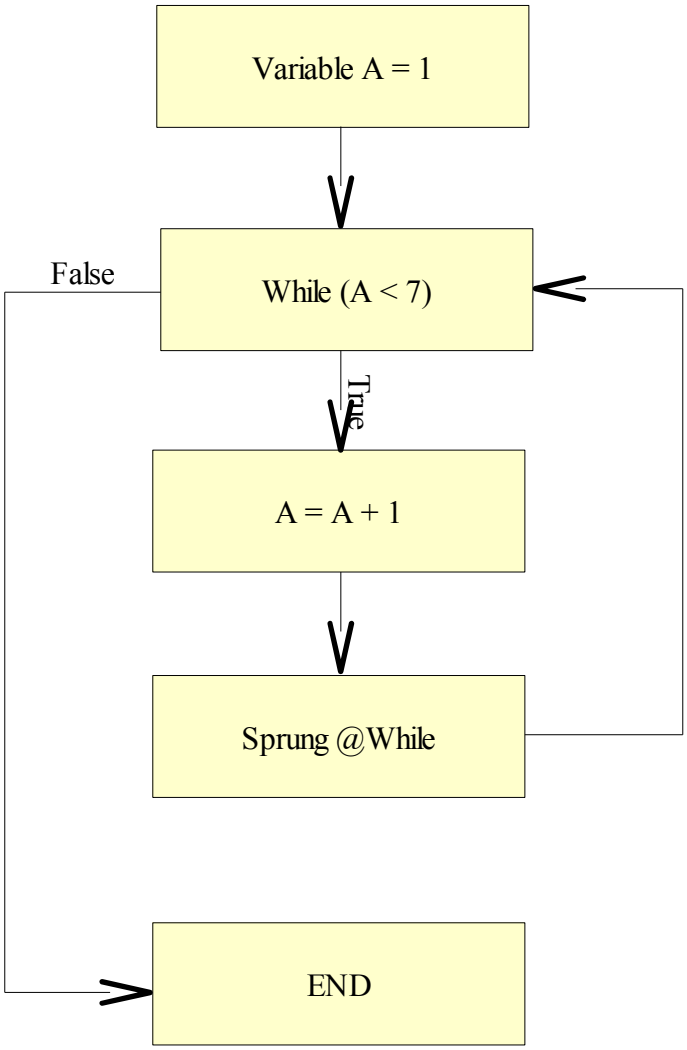
```
int a = 1, b;          // CONST 1
                       // STOG 0
read  b;               // READ
                       // STOG 1
do
{                      // ENTER 1
      write a;         // LOADG 0
                       // WRITE
      a = a + 1;       // LOADG 0
                       // CONST 1
                       // ADD
                       // STOG 0
}                      // LEAVE
while(a < b);          // LOADG 1
                       // LOADG 0
                       // GTR
                       // FJMP @do
                       // RET
```

Zum vorangegangenen Beispiel für eine do-while Schleife sieht das Binärprogramm wie folgt aus:

```
 Adr nachfolgendes Kommando
---------------------------
1000 020 000 000 000 001   // CONST 1
1005 024 000 000 000 000   // STOG 0
1010 064                   // READ
1011 024 000 000 000 001   // STOG 1
1016 122 000 000 000 001   // ENTER 1
1021 022 000 000 000 000   // LOADG 0
1026 101                   // WRITE
1027 022 000 000 000 000   // LOADG 0
1032 020 000 000 000 001   // CONST 1
1037 040                   // ADD
1038 024 000 000 000 000   // STOG 0
1043 123                   // LEAVE
1044 022 000 000 000 001   // LOADG 1
1049 022 000 000 000 000   // LOADG 0
1054 062                   // GTR
1055 081 000 000 000 3F8   // FJMP 1016
1060 121                   // RET
```

Unbedingte Sprünge

Ein unbedingten Sprung ist typischerweise die while-Schleife. Die folgende Abbildung zeigt die schematische Darstellung eines unbedingten Sprunges:



Umsetzung

Nachfolgend wird ein sehr einfaches Beispiel für eine while Schleife erläutert. In diesem Beispiel soll, mittels einer while Schleife, solange 1 zu einer Zahl addiert werden, bis diese einem eingelesenen Wert entspricht.

```
int a = 1, b;          // CONST 1
                       // STOG 0
read  b;               // READ
                       // STOG 1
while(a<b)             // LOADG 1
                       // LOADG 0
                       // GTR
                       // FJMP @END
{                      // ENTER 1
      write a ;        // LOADG 0
                       // WRITE
      a = a + 1;       // LOADG 0
                       // CONST 1
                       // ADD
                       // STOG 0
}                      // LEAVE
                       // JMP @while
                       // RET
```

Das Assemblerprogramm für eine while Schleife sieht wie folgt aus:

```
 Adr nachfolgendes Kommando
---------------------------
1000 020 000 000 000 001   // CONST 1
1005 024 000 000 000 000   // STOG 0
1010 064                   // READ
1011 024 000 000 000 001   // STOG 1
1016 022 000 000 000 000   // LOADG 0
1021 022 000 000 000 001   // LOADG 1
1026 062                   // GTR
1027 081 000 000 000 3F8   // FJMP 1065
1032 122 000 000 000 001   // ENTER 1
1037 022 000 000 000 000   // LOADG 0
1042 101                   // WRITE
1043 022 000 000 000 000   // LOADG 0
1048 020 000 000 000 001   // CONST 1
1053 040                   // ADD
1054 024 000 000 000 000   // STOG 0
1059 123                   // LEAVE
1060 080 000 000 000 001   // JMP 1016
1065 121                   // RET
```

Ressourcen

Ressourcen -oder Devices- sind globale Variabeln, welche sich an externe I/O-Treiber koppeln lassen. In der VM v1.1 werden lediglich Integer-Variabeln getauscht, so dass keine komplexen Protokolle möglich sind. Trotzdem sollen sie die Sprache flexibler machen. Besonders gegenüber Hardware- oder Betriebssystem-Funktionen.

Beispiel

```
dev PortA;

void main()
{
  use 'StdTimer' for PortA;
  PortA = 0;
  writec '10s Timer:';
  while (PortA < 10)
  {
    writec '.';
    rtsleep 500;
  }
  newline;
  writec 'fertig';
}
```

Der Treiber „libStdTimer.so“ (oder .dll, je nach Plattform) wird bei der Anweisung 'use' der Ressource „PortA“ zugewiesen. Dabei geht die VM wie folgt vor:

1. Beenden eines möglichweise bereits zugeordneten Treibers.
2. Laden des Treibers „libStdTimer.so“
3. Initialisierung der Treiber-Funktionen (I/O- und Status-Methoden)
4. Zuordnung des Treiber-Handlers zur Adresse 1000 (PortA)

Ressourcen können also im Betrieb geändert werden – nicht nur zu Debug-Zwecken.

Für die „use“ -Anweisung wird in der Binary für die Ressource nur noch eine ID eingesetzt. Damit dies möglich wird, muss die Ressource mit ID vorher angekündigt werden. Im ASCII-Block „Drivers“ wird dies realisiert (s. aktuelle RFC für die Binary-File).

Standard-Drivers

Weil sich die Ressoucen vom Programm aus laden lassen, wäre es schwierig portable Programme zu schreiben, ohne dass die jeweiligen Treiber mitkopiert werden. Damit würde auch gleich die Plattform-Unabhängigkeit zunichte gemacht werden. Um diesem Problem auszuweichen, soll auf jeder GRINJ-Installation ein „Basis-Packet“ an Treibern zur Verfügung stehen. Dabei wird nicht die Technologie festgelegt (die ist abhängig von BS und Hardware), sondern lediglich die Funktionalität:

- **StdRandom**
  Zahlengenerator von 0-1000
- **StdTimer**
  Relative Zeitmessung in Sekunden oder Millisekunden:
  PortA = -1; // Initialisierung auf Sekunden (Default)
  PortA = -2; // Initialisierung auf Millisekunden

  Das Device beinhaltet die relative Zeit in der gegebenen Einheit ab der Initialisierung. Das Device kann jederzeit neu Initialisiert werden:
  PortA = 20; // wird in 10s den Wert 30 haben...
- **StdLocalPort**
  Irgend ein Standard-Port für (lokale) Hardware. Empfohlen sind z.B. LPT oder sonstiger paralleler Port (z.B. LED oder Switches auf Eval-Boards für Embedded-Systeme).
- **StdRemotePort**
  Ein Daten-Port für Remote-Zugriff. Empfohlen sind z.B. TCP-Socket oder Serielle-Schnittstelle. Die jeweiligen Einstellungen (Port-Nummer oder sonstiges) werden dabei komplett dem Treiber überlassen (fix oder Konfigurations-File denkbar)

Weitere sind denkbar und werden sicher folgen.

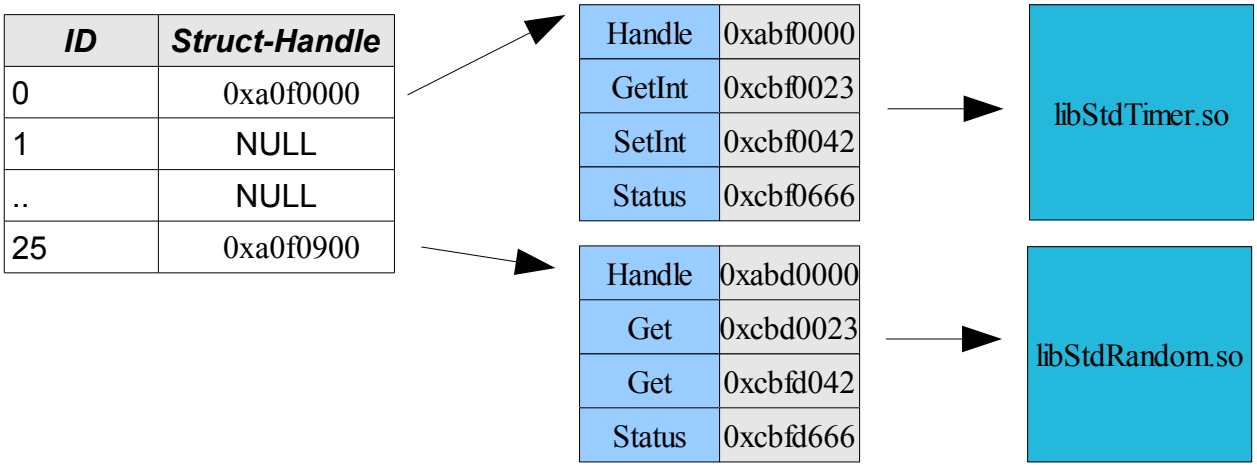
Implementierung

Die Implementierung nutzt dlopen(), das Interface zum „*dynamic linker loader*“. Dazu ein Beispiel -ausnahmsweise in C:

```
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    double foo;
    void *handle;                       // Handle zum Treiber
    double (*cosine)(double);           // Funktionspointer zur Methode „cos“
    handle = dlopen ("libm.so", RTLD_LAZY);      // open, und ..

    *(void **) (&cosine) = dlsym(handle, "cos"); // ermitteln des Pointers

    foo = (*cosine)(2.0));              // Aufrufen der Funktion
    dlclose(handle);                    // ...
    return 0;
}
```

Natürlich fehlt hier die Fehlerbehandlung komplett. Aber die Abindung ist relativ simple. In der VM werden deswegen nur noch Handler und Funktionspointer zwischengespeichert:

Ressourcen-Speicher als Array in GrinjHardwareEmul:



Die ID des Struct-Handle ist die Position innerhalb vom Array und zugleich Adress-Offset oberhalb von 1000. Das fixe Array ist zwar starr, doch lässt sich so der Treiber ohne aufwendiges Suchen ansprechen. Der Speicherhunger der VM wird über die Struct-Zwischenlagerung etwas rediziert.

Das Treiberinterface wurde als RFC (s. Anhang) definiert und wird an dieser Stelle deshalb nicht nochmals beschrieben.

Debugger-Informationen

Die Debug-Informationen liegen als Klartext im Anhang der Binary. Der Compiler listet einfach alle Zeilen auf, auf denen GRINJ-Source in GRINJ-OP-Codes umgesetzt wurden und liefert entsprechende Adressen dazu.

Nehmen wir also nochmals ein vollständiges Beispiel

```
Zeile Code          OP          Code-Adresse in der Binary
-------------------------------------------------------------
 1 // Nur Kommentar
 2 {                ENTER 1     1000
 3 int a;
 4 read a;          READ        1005
                    STO 0
 5 If (a > 7)       CONST 7     1011
                    LOAD 0
                    LSS
                    FJMP @else
 6 then
 7   a = 1;         CONST 1     1027
                    STO 0
                    JMP @Write
 8 else
 9   a = 0;         CONST 0     1042
                    STO 0
10 end

11 write a ;        LOAD 0      1053
                    WRITE
12 }                LEAVE       1059
                    RET
```

Dies gäbe als gem. RFC 003 den Debug-Block:

```
1000  2 beispiel.grinj
1005  4 beispiel.grinj
1011  5 beispiel.grinj
1027  7 beispiel.grinj
1042  9 beispiel.grinj
1053 11 beispiel.grinj
1059 12 beispiel.grinj
```

Auf diese Weise kann der Debugger auf das Kommando „Halte in der Datei beispiel.grinj auf Zeile 4“ ermitteln, wo er im Code-Speicher zu halten hat..

Software-Doku

Systemanforderungen

Die GRINJ-VM basiert auf C++ und stellt an das System folgende Anforderungen:

- C99-kompatibler Compiler
- POSIX-Support (Threads und Sockets)
- etwa 2MB RAM zur Laufzeit

Das Interface für Java-Anwendungen wurde mit der Java Runtime 1.50_06 entwickelt und getestet.

Damit lässt sich die VM für eine Vielzahl an Betriebsysteme und Architekturen generieren:

Plattform

Die VM wurde auf folgenden Systemen getestet:

- Linux/x86 mit gcc 4.0.3
- Windows/Cygwin (NT-5.x) mit gcc 3.4.4
- Darwin/PPC mit gcc 4.0.0
- Darwin/x86 mit gcc 4.0.1
- Sun Solaris/SPARC mit gcc 3.4.x
- Sun Solaris/SPARC mit Sun Compiler (CC) 5.4

Wichtig ist, die entsprechende Plattform vor dem generieren in der Header-Datei „vm/include/configure.h“ anzupassen (siehe „Programmgenerierung“).

Programmgenerierung

Die GRINJ VM wurde mittels C++ (C99) und POSIX implementiert (s. „Systemanforderungen“). Der Source kann also mit jeder IDE, welche diese beiden Voraussetzungen unterstützt, generiert werden. Für Systeme mit dem Tool „GNU-Make“ kann die GRINJ-VM direkt am Verzeichnis erstellt werden:

```
user@host:~/grinj$ cd vm
user@host:~/grinj/vm$ cd obj
user@host:~/grinj/vm/obj$ make
g++ -Wall -g -c  ../src/../src/main.cc -o main.o
g++ -Wall -g -c  ../src/../src/GrinjBinaryFile.cc -o GrinjBinaryFile.o
g++ -Wall -g -c  ../src/../src/GrinjHardwareEmul.cc -o GrinjHardwareEmul.o
[...]
g++ -Wall -g -c  ../src/../src/GrinjIOFile.cc -o GrinjIOFile.o
g++ -Wall -g -c  ../src/../src/GrinjIOSocket.cc -o GrinjIOSocket.o
g++ -Wall -g -c  ../src/../src/GrinjException.cc -o GrinjException.o
g++ -Wall -g -c  ../src/../src/GrinjCoreDumpException.cc -o GrinjCoreDumpExcepti
on.o^^^^^^^^^^^^^^^^^^^^^^^^^^
g++ -Wall -g -lpthread main.o GrinjBinaryFile.o GrinjHardwareEmul.o GrinjCoreDum
p.o GrinjVmThread.o GrinjServerThread.o GrinjSocketServer.o GrinjDebugModule.o G
rinjIODevice.o GrinjIOConsole.o GrinjIOFile.o GrinjIOSocket.o GrinjException.o G
rinjCoreDumpException.o -o ../bin/grinj
================================================================================
/home/user/grinj/vm/obj/../bin/grinj generiert!
================================================================================
user@host:~//grinj/vm/obj$ sudo make install
cp ../bin/grinj /usr/local/bin
install vm complete
user@host:~/grinj/vm/obj$
```

Achtung: Teile der VM sind an die jeweilige Plattform optimiert. Die Zielplattform ist also unbedingt in der Datei

grinj/vm/include/configure.h

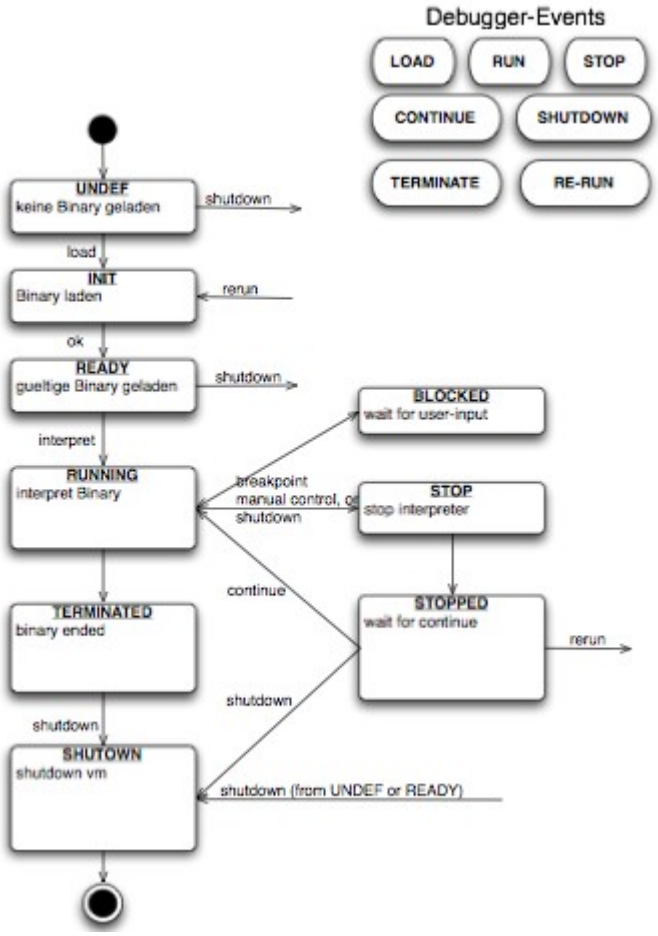
einzutragen. Wird dies nicht gemacht, ist die neu erstellte VM instabil und wird (je nach System) bei der Programmausführung abstürzen.

VM States

Die VM ist eine Finite-State-Machine in zweierlei Hinsichten:

- Zustand der VM hinsichtlich der VM-Steuerung
- Zustand der VM hinsichtlich der Programm-Interpretierung

Die Interpretation des GRINJ-Codes wird wie üblich via FETCH-DECODE-EXECUTE abgewickelt und wird hier nicht weiter behandelt. Die VM-States hingegen sind wichtig für den Umgang mit der VM:



Dies bedeutet:

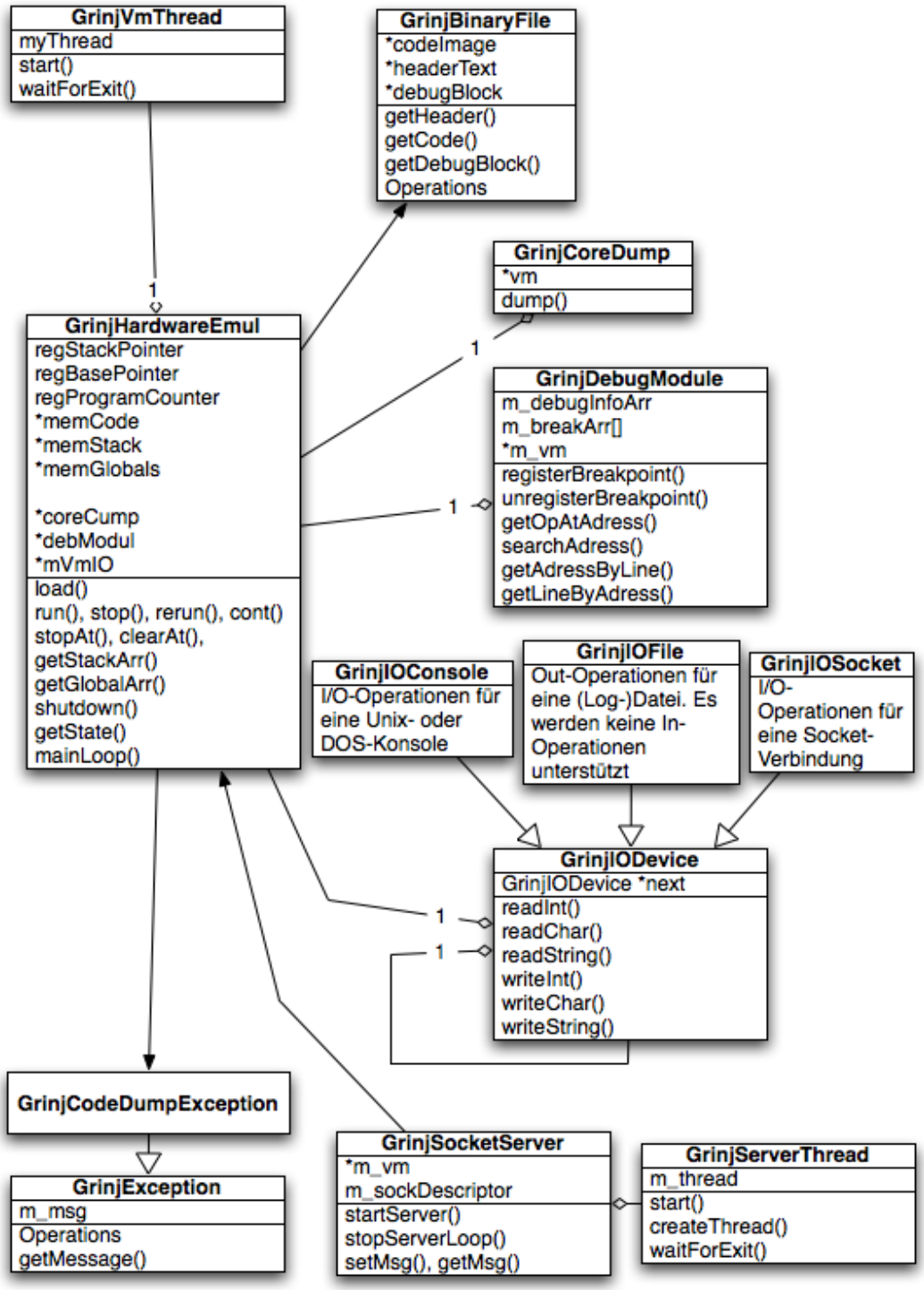
- Die VM geht immer in den Zustand READY, wenn eine gütlige Binary geladen wurde.
- Beendet werden kann die VM immer, ausser sie ist im Zustand „BLOCKED“. Dieser Zustand wird immer angenommen, wenn vom Programm ein User-Input verlangt wird.
- Bei einer laufenden Debug-Session, dienen die Events dazu, die VM Zustände zu steuern. Ohne Debugging durchläuft die VM immer UNDEF->INIT->READY->RUNNING->TERMINATED->SHUTDOWN.

Klassenbeschreibung

Klasse	*Kurzbeschreibung*
GrinjHardwareEmul	VM-Kern mit Interpreter und Hauptloop. Programm-Ausführung und Event-Reaktion ist vollständig in dieser Klasse abgebildet.
GrinjBinaryFile	Interpreter der GRINJ-Binärdatei (GRINJ-Programm). Instanzen dieser Klasse sind kurzlebig und werden nur während der Init-Phase benötigt.
GrinjDebugModule	Container aller Debug-Informationen (wie den Referenz-Daten aus der Binärdatei, Breakpoint-Daten)
GrinjCoreDump	Analyzer für Software-Crashes. Im Falle eines Fehlers während der GRINJ-Programm-Ausführung gibt diese Klasse interne Informationen wie Stack-Daten und Register-Zustände aus.
GrinjSocketServer	Debug-Konsole mit Protokoll-Interpreter
GrinjServerThread	Erzeugerklasse für den Debug-Thread
GrinjIOConsole	I/O-Modul für eine Konsole
GrinjIOSocket	I/O-Modul für einen Socket-Server
GrinjIOFile	I/O-Modul für eine Log-Datei
GrinjIODevice	Basisklasse für alle I/O-Module
GrinjCoreDumpException	Exception löst ein Core-Dump der VM aus
GrinjException	Basisklasse für Exceptions

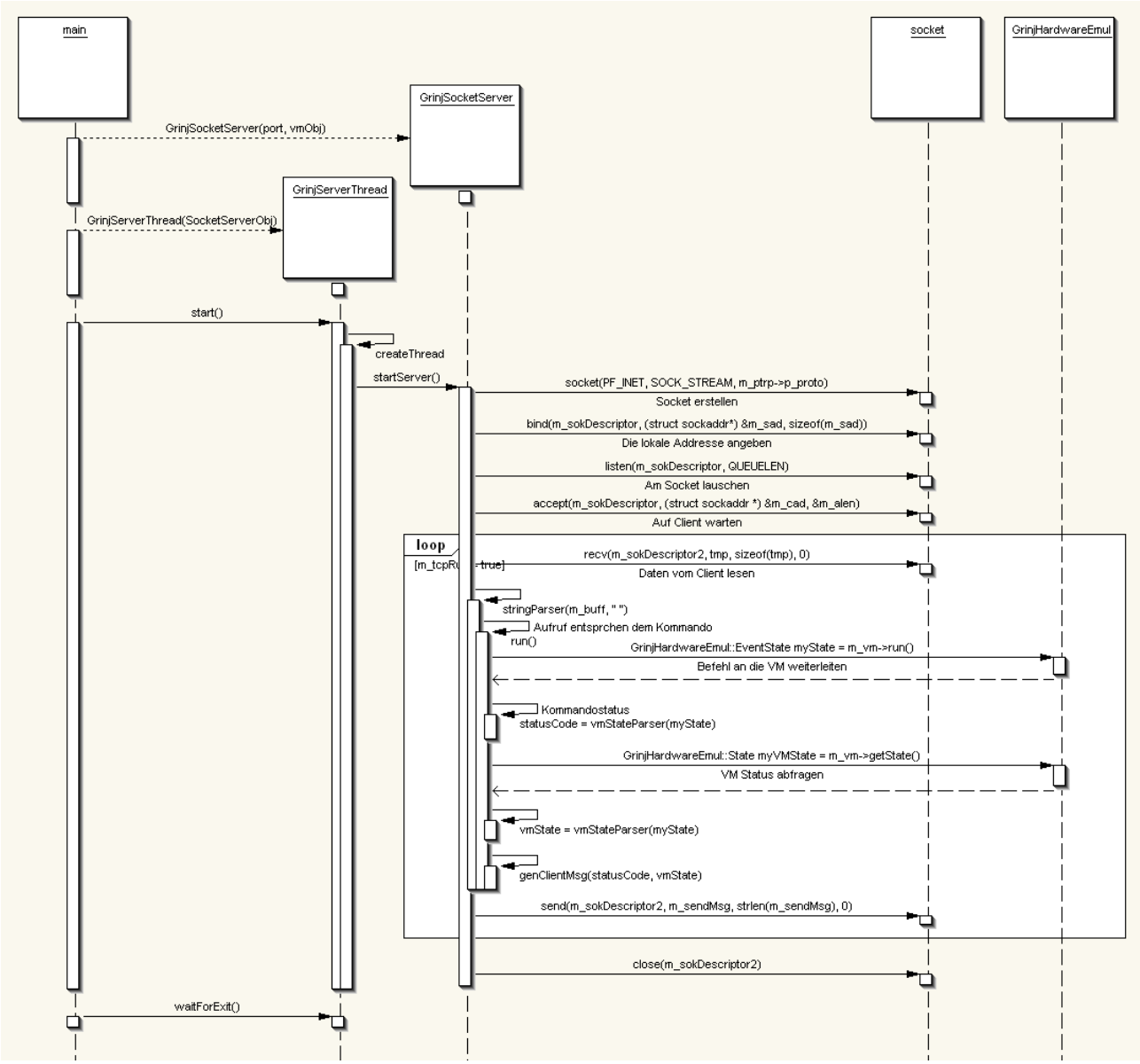
Klassendiagramm

Zwecks besserer Übersicht sind im Klassendiagramm die Beziehungen unvollständig. Die meinsten Module greifen bei ihrer Tätigkeit (z.B. Zwecks Informationsbeschaffung) zurück auf die VM. Die Aggregationen durch diese „Parent-Pointer“ fehlen. Dafür sind die wichtigsten Eigenschaften (z.B. Chain-of-Responsibility via GrinjIODevice) deutlich sichtbar:



VM Erzeugung

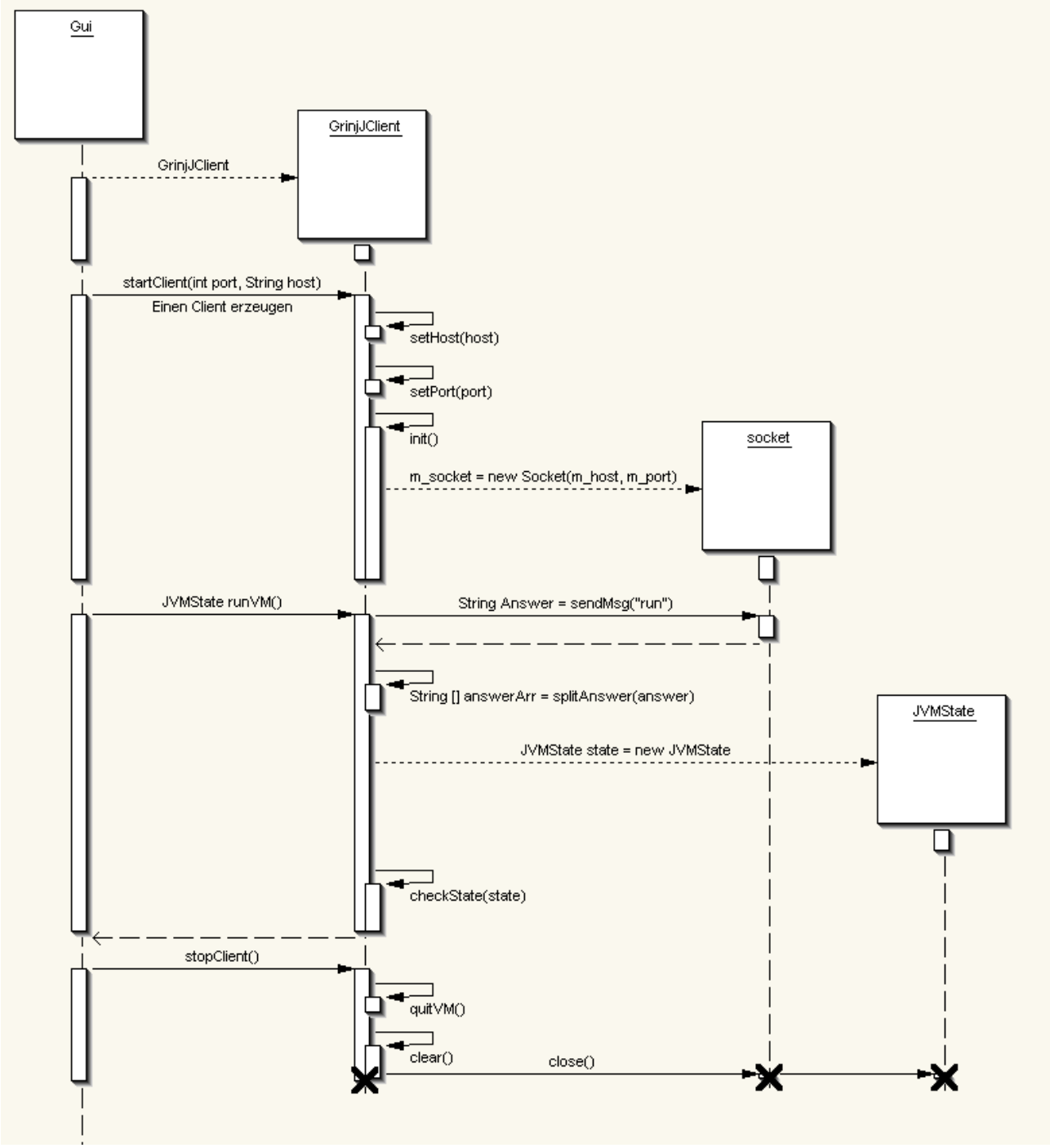
Das nachfolgende Sequenzdiagramm zeigt die Erzeugung der VM während der Startphase und das Eventhandling zwischen Socket-Server und VM im Debug-Betrieb:



Java-Client

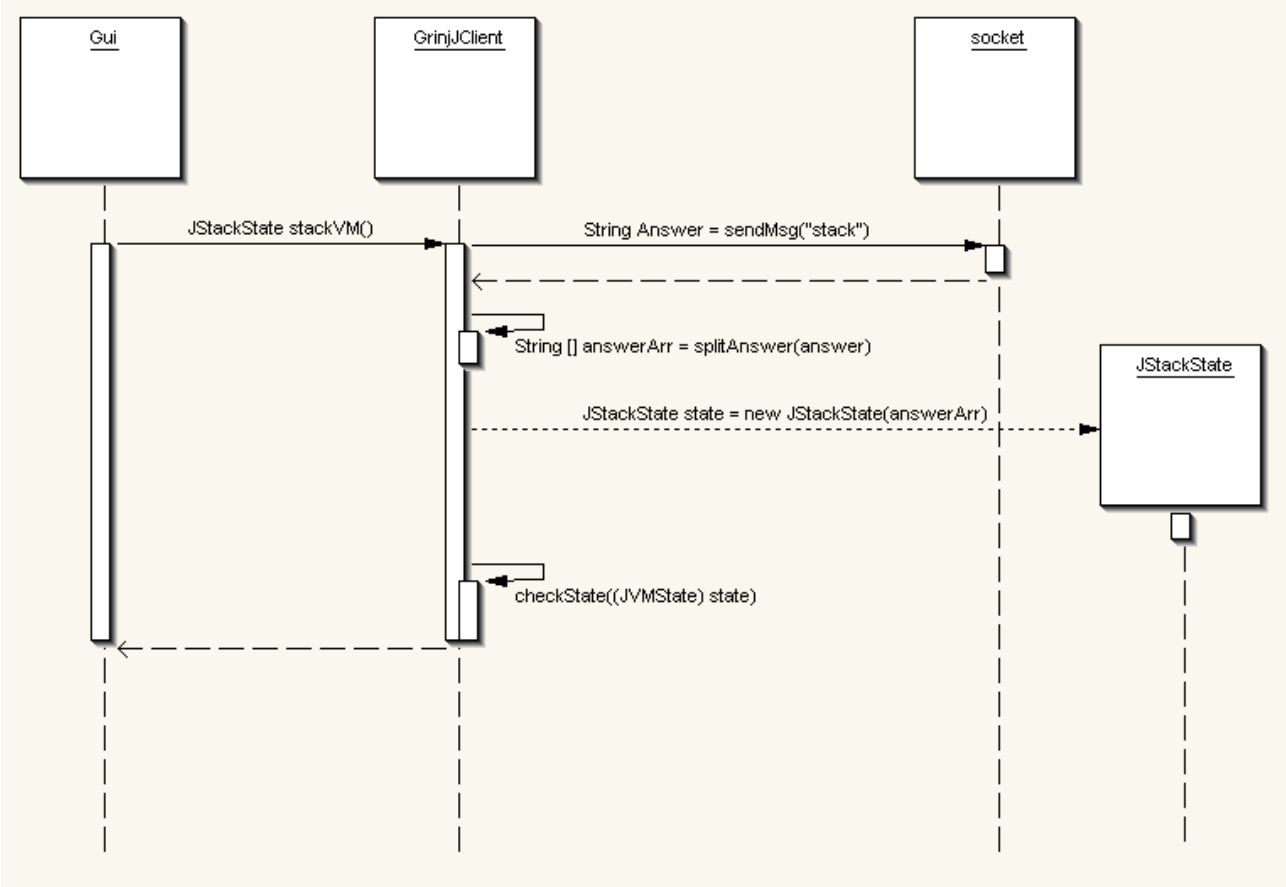
Init-Sequenz

Der Java-Client (GUI oder beliebige anderer Client) für Debugging kann über die Klasse GrinjJClient mmit der VM kommunizieren. Eine typische Sequenz für Erzeugung, Init, Start und Stopp sieht wie folgt aus:



Kommando-Sequenz

Die Debug-Kommandos werden dabei immer gleich behandelt:



Im Wesentlichen werden jeweils folgende Punkte abgearbeitet:

1. Sendes des Kommandos (z.B. „Stack“, für das Abfragen des Stack-Inhalts)
2. Empfangen der Nachricht
3. Decodieren der Nachricht (splitAnwser)
4. Interpetieren der Stati (JstackState)
5. Bei einem Fehler werfen einer Exception (checkState)

HMI

Einführung

Zur Erstellung des HMI (Human-Machine-Interface) war für uns, nach einer kurzen Evaluationsphase, klar, dass wir das Eclipse-Framework verwenden wollten.
Natürlich standen uns Alternativen zu Verfügung, die wir auch zu Projektstart eingehend prüften. Aus verschieden Gründen (siehe Abschnitt „Warum Eclipse für GRINJ?“) entschlossen wir uns, nicht zuletzt auch wegen persönlicher Interessen, für das Eclipseframework.

Verschiedene Arbeiten wurden uns dadurch vom System abgenommen. Wiederum andere Eigenheiten erhöhten wiederum den Aufwand. So waren zum Beispiel die Dokumentationen zur allerneusten Eclipseversion nicht immer auf dem aktuellsten Stand. Lehrgeld das man bei jeder Einarbeitung in ein Framework zahlen muss.

Insgesamt bleibt aber festzuhalten, dass das Framework sehr gross und sehr mächtig ist. Einige Konzepte müssen verstanden sein, und dann wird die Arbeit sehr vereinfacht. Beeindruckend ist auch die Einfachheit im Aufbau des Frameworks. So ist es sehr einfach eine Applikation, auch eine bestehende, um weitere Funktionen zu erweitern und mit anderen Komponenten interagieren zu lassen.

Eclipse

Eclipse ist der Nachfolger von IBM Visual Age. Heute ist es ein Open-Source-Framework zur Entwicklung von Rich-Client-Applikationen (Rich Client = Fat Client = Smart Client = Verarbeitung der Daten vor Ort auf dem Client; meist mit grafischer Oberfläche).

Die bekannteste Anwendung ist wohl die Nutzung als Entwicklungsumgebung (IDE), bekannt unter dem Namen „Eclipse SDK“ mit Werkzeugen zur Entwicklung von Java-Applikationen (Java Development Tools JDT) und Eclipse-Plugins (Plug-in Development Environment PDE).

Von IDE zu RCP

Mit Version 3.0 hat sich Eclipse vom Konzept der erweiterbaren IDE entfernt. Nur noch der schlanke Kern ist Eclipse selbst. Die Plugins, die er lädt, stellen die gesamte Funktionalität zur Verfügung. Das wird RCP (Rich Client Platform) genannt. RCP basiert auf dem OSGi-Standard, dessen Merkmal es unter anderem ist, dass Service-Anwendungen (Bundles) während der Laufzeit geladen und auch wieder entfernt werden können.

Sowohl Eclipse als auch die Plugins sind komplett in Java implementiert.

Als GUI-Framework wird SWT (Standard Widget Toolkit) verwendet; eine Entwicklung von IBM eigens für Eclipse. SWT ist vergleichbar mit dem AWT, da es auch die nativen grafischen Elemente des Betriebssystems verwendet. Somit ist Eclipse nicht plattformunabhängig, wird aber für verschiedenste Architekturen bereitgestellt.

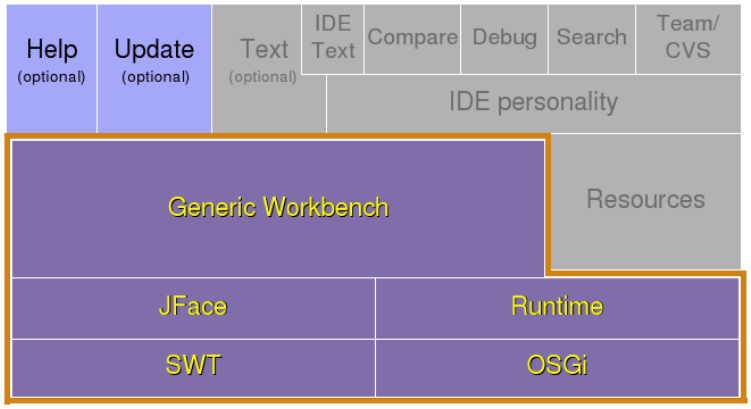


Abbildung 1: Verschiedene Komponenten können optional zur RCP dazugeladen werden. Zum Kern gehören nur OSGi, Runtime, Generic Workbench, SWT und JFace, welches eine Abstraktionsstufe mehr für SWT bietet.

Warum Eclipse für GRINJ?

Die GRINJ-GUI sollte eine kleine Entwicklungsumgebung für die GRINJ-Sprache darstellen. Sie sollte mit einer VM und einem Compiler kommunizieren können.

Eclipse war ursprünglich als erweiterbare IDE geplant. Es steckt alles darin, was eine IDE bieten muss und was wir von ihr fordern. Und noch viel mehr. Wir können uns auf bewährte Praktiken und Konzepte stützen und auf einer durchdachten Architektur aufbauen.
JFace bietet jede Menge grafischer Elemente für eine Programmierer-GUI an, die wir für unser Projekt verwenden und weiterentwickeln können. Eclipse ist Open-Source.
Viele grafische Elemente lassen sich bequem über eine Konfigurationsdatei (`plugin.xml`) lösen. Das bietet den klaren Vorteil der Effizienz und der einfachen Anpassungsmöglichkeit.

Gegen die Verwendung von Eclipse spricht unsere bescheidene Erfahrung mit RCP. Um fremde Konzepte erfolgreich zu adaptieren und erweitern müssen wir sie verstanden haben. Bei einer Entwicklung mit SWING oder mit AWT hätten wir unsere eigene Architektur aufbauen können (müssen).

RCP-Applikation vs. Plugin

In Eclipse ist alles ein Plugin. Also auch eine RCP-Applikation.

Während das Plugin ein installiertes Eclipse voraussetzt, auf dem es aufsetzen kann, wird bei der RCP-Applikation beim „Produkt-Export“ ein kompilierter und plattformabhängiger, schlanker Kern mitgeliefert, der das Plugin lädt. Dieser Kern wiegt weniger als 7 MB. Ein durchschnittlicher PC-Benutzer wird darum eine RCP-Applikation nicht als ein Eclipse-Plugin identifizieren können.

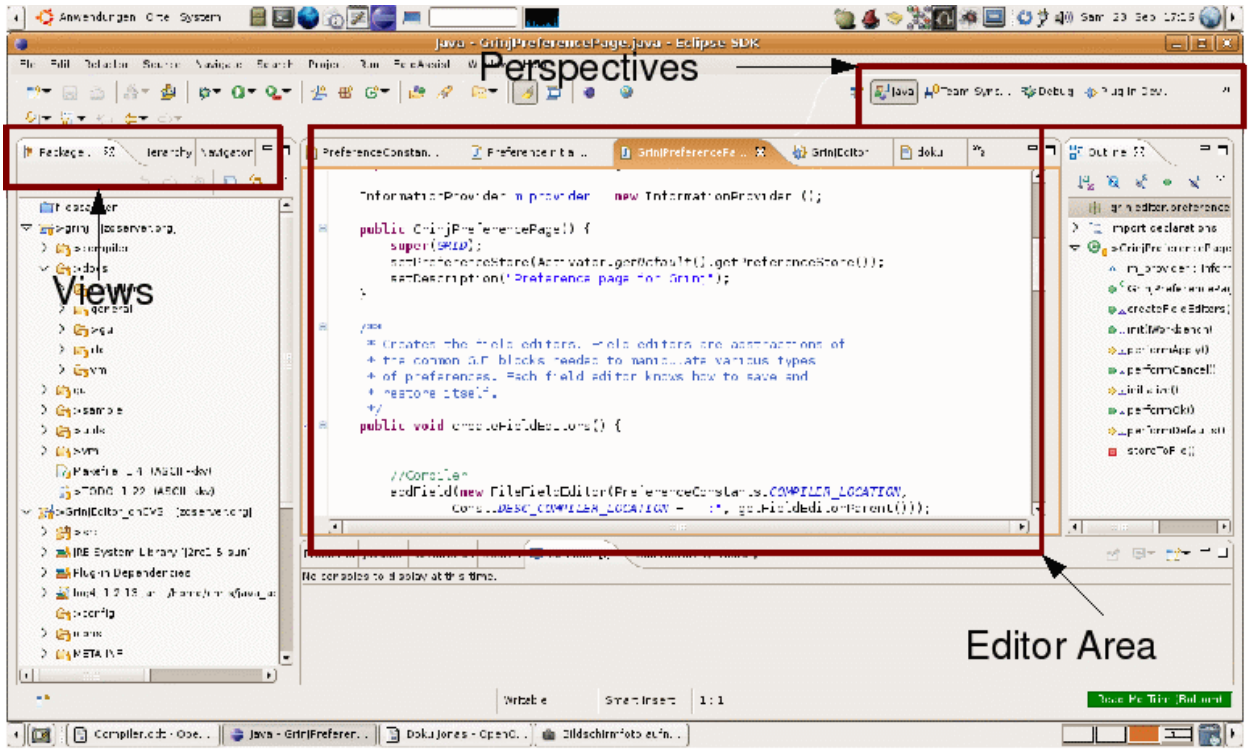
Aus Gründen der Unabhängigkeit haben wir uns für unsere GRINJ-GUI für die Option der RCP-Applikation entschieden.

Eclipse – Konzepte

Um das Eclipse-Framework effizient und auch möglichst fehlerfrei nutzen zu können, mussten wir uns mit einigen Konzepten vertraut machen.
Grundsätzlich kann, muss aber nicht, eine RCP-Applikation folgende Kompentenen beinhalten:

- Perspective

- View
- Editor



Perspectiven

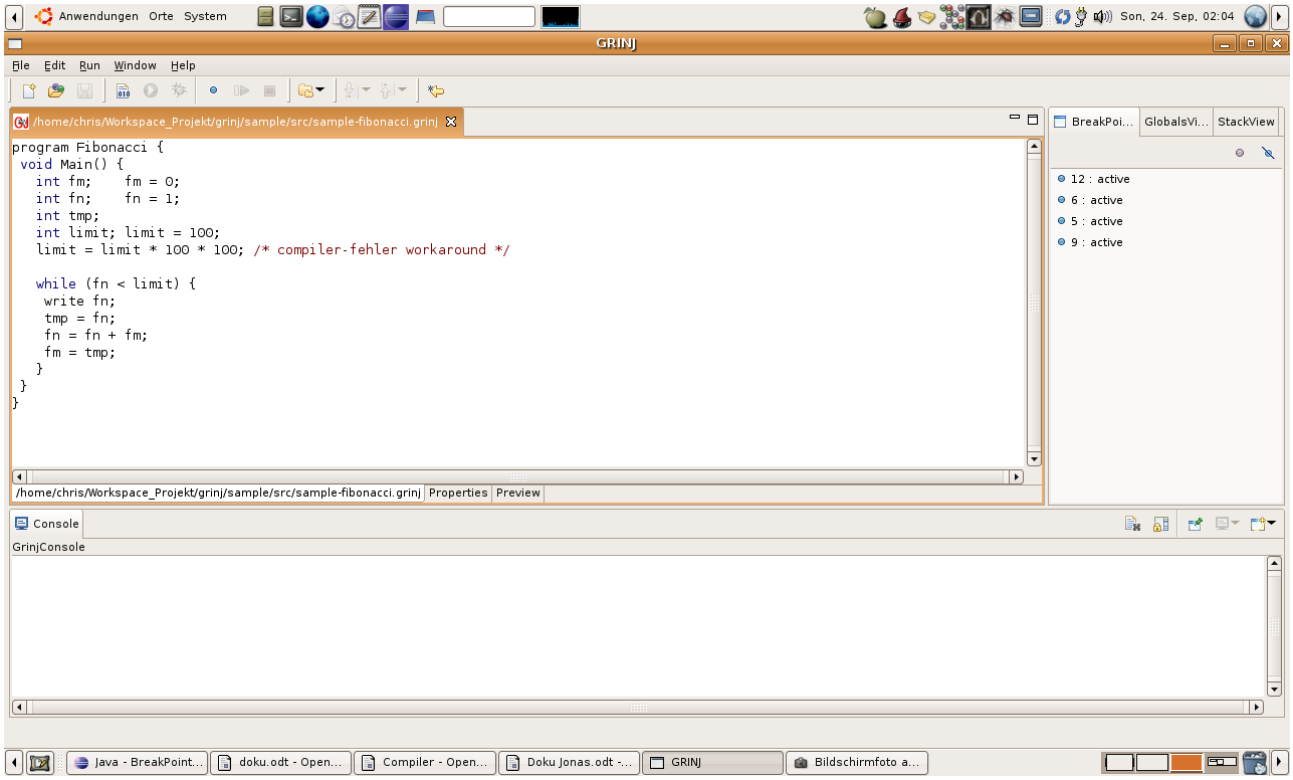
Eine Perspective ist eine Zusammenstellung von mehrer Views die zu einem eigenen Layout. Damit ist es möglich ohne grossen Aufwand die Ansicht von verschiedenartigen Information voneinander zu trennen und trotzdem schnell zugänglich zu machen.
Es ist möglich eigene Perspektiven zu definieren und anzuzeigen. Es kann aber immer nur eine Perspective angezeigt werden.
In der Applikation Grinj gibt es nur eine Default-Perspective. Es hätte kein Sinn ergeben mehr als eine zu haben.

Views....

Views sind seperat einblendbare Fenster. Sie sind sehr variabel und vielseitig einsetzbar. So können sie zum Beispiel von Anfang an angezeigt werden, oder erst zu einem späteren Zeitpunkt hinzugefügt werden. Sowohl ihre Position als auch die Grösse sind vollkommen frei wählbar. Auch die Zugehörigkeit zu einer Perspektive ist vom Benutzer konfigurierbar.
In unserer Applikation bedienten wir uns dieser Vielseitigkeit für die Ansicht der Breakpoints, der globalen Variablen und des Stacks.
Um eine Einblick in das Funktionieren der Darstellung zu geben, ist das Beispiel der BreakpointView ideal.

```
public class BreakPointView extends ViewPart {
	private TableViewer viewer;

	private Action toggleActive;

	private Action removeBreakpoint;

	class ViewContentProvider implements IStructuredContentProvider {
		public void inputChanged(Viewer v, Object oldInput, Object newInput) {
		}

		public void dispose() {
		}

		public Object[] getElements(Object parent) {
			return ((ArrayList) parent).toArray();
		}
	}

	class ViewLabelProvider extends LabelProvider implements
			ITableLabelProvider {
		public String getColumnText(Object obj, int index) {
			return ((GRINJBreakpoint) obj).getRepresentation();
		}

		public Image getColumnImage(Object obj, int index) {
			return getImage(obj);
		}

		public Image getImage(Object obj) {
			if (((GRINJBreakpoint) obj).isActive()) {
				return Activator.getImageDescriptor("icons/breakpoint.gif")
						.createImage();
			} else {
				return Activator.getImageDescriptor(
						"icons/breakpointInaktiv.gif").createImage();
			}
		}
	}

// Code omitted //

	/*
	 * Versucht den aktuellen Editor, und die dazughoerigen Breakpoints, zu
	 * resolven und darzustellen
	 */
	public void refreshVIEW() {
		GrinjEditor grinj = (GrinjEditor) PlatformUI.getWorkbench()

	.getActiveWorkbenchWindow().getActivePage().getActiveEditor();

		ArrayList<GRINJBreakpoint> breakPointList = null;
		if (grinj != null) {
			breakPointList = ((GRINJDocumentProvider) grinj.getEditor()
					.getDocumentProvider()).getBreakpoints();
		}

		viewer.setInput(breakPointList);
		viewer.refresh(true);
}
}
```

Wichtiger Bestandteil dieser View sind die beiden InnerClasses ViewContentProvider und ViewLabelProvider. Die Klasse ViewContentProvider liefert die darzustellenden Daten an den Tableviewer (welcher alternativ auch ein TreeViewer sein könnte) und formatiert sie entsprechend. Der ViewContentProvider hingegen ist verantwortlich die jeweiligen Kolonneninformation aus den Daten zu holen und zu beschriften.



Kommentar zur Graphik:
Deutlich sind die Unterschiede zwischen dem grossen Eclipse (Graphik 1) und dem kleinen Eclipse, unserem Grinj (Grafik 2), zu sehen. Aber auch die Gemeinsamkeiten sind deutlich zu sehen. So zum Beispiel die Aufteilung des Fenster.

...und Editoren

Die Editoren sind ein Hauptbestandteil in Eclipse. Weshalbe ihnen auch die zentrale Position innerhalb des Workbenches zukommt. Editoren unterstützen die klassischen Editierfähigkeiten, wie zum Beispiel Copy-Paste, aber auch weiterführende Funktionen wie Syntaxhighlightning.

```
public class GRINJPartitionScanner extends RuleBasedPartitionScanner
{
	public static final String SINLGE_LINE_COMMENT = "singleComment";
	public static final String MULTI_LINE_COMMENT = "multiComment";

	public static final String[] PARTITION_TYPES = {SINLGE_LINE_COMMENT,
			MULTI_LINE_COMMENT};

	public GRINJPartitionScanner() {
		IToken singleLineCommentToken = new
Token(GRINJPartitionScanner.SINLGE_LINE_COMMENT);
		IToken multiLineCommentToken = new
Token(GRINJPartitionScanner.MULTI_LINE_COMMENT);

		IPredicateRule[] rules = new IPredicateRule[2];
		rules[0] = new EndOfLineRule("//",singleLineCommentToken);
		rules[1] = new MultiLineRule("/*", "*/", multiLineCommentToken, (char) 0, true);

		setPredicateRules(rules);
	}
}


public class GRINJCodeScanner extends RuleBasedScanner {


		private String[] fgKeywords= { "abstract", "break", "case", "catch", "class",
"continue", "default", "do", "else", "extends", "final", "finally", "for", "if", "implements",
"import", "instanceof", "interface", "native", "new", "package", "private", "protected",
"public", "return", "static", "super", "switch", "synchronized", "this", "throw", "throws",
"transient", "try", "volatile", "while" };

		private String[] fgTypes= { "void", "boolean", "char", "byte", "short", "int",
"long", "float", "double" };
		private String[] fgConstants= { "false", "null", "true" };
		public GRINJCodeScanner () {
			IToken keyword= new Token(new TextAttribute(new GRINJColorProvider
().getColor(GRINJColorProvider.KEYWORD)));
			IToken type= new Token(new TextAttribute(new GRINJColorProvider
().getColor(GRINJColorProvider.TYPE)));
			IToken string= new Token(new TextAttribute(new GRINJColorProvider
().getColor(GRINJColorProvider.STRING)));
			IToken comment= new Token(new TextAttribute(new GRINJColorProvider
().getColor(GRINJColorProvider.SINGLE_LINE_COMMENT)));
			IToken other= new Token(new TextAttribute(new GRINJColorProvider
().getColor(GRINJColorProvider.DEFAULT)));

//			// Add rule for single line comments.
			IRule singleCommentRule = new EndOfLineRule("//", comment);

//
//			// Add rule for strings and character constants.
			IRule stringRule = new SingleLineRule("\"", "\"", string, '\\');
//
//			// Add generic whitespace rule.
			IRule whitespaceRule = new WhitespaceRule(new JavaWhitespaceDetector());

			// Add word rule for keywords, types, and constants.
			WordRule wordRule= new WordRule(new JavaWordDetector (), other);
			for (int i= 0; i < fgKeywords.length; i++)
				wordRule.addWord(fgKeywords[i], keyword);
			for (int i= 0; i < fgTypes.length; i++)
				wordRule.addWord(fgTypes[i], type);
			for (int i= 0; i < fgConstants.length; i++)
				wordRule.addWord(fgConstants[i], type);

			IRule[] rules = new IRule[] {singleCommentRule, stringRule, wordRule};

			setRules(rules);
		}
	}
}
```

Um das Syntaxhighlightning gewährleisten zu können, wird ein Sourcefile in ein Document (Datensicht des Sourcefiles) gewandelt und durch einen Partionscanner grob in verschiedene Teile aufgeteilt. Des weiteren werden nun diese „groben“ Teile durch einen Tokenscanner, der durch Rules defniert ist, weiter aufgeteilt. So liegt nun des ganze Sourcefile in seine Einzelteile zerlegt vor, die nun mit Farben eingefärbt, oder sonstig contextabhängige Veränderungen vorgenommen werden können. So gehören jetzt zum Beispiel die jeweiligen Klammern zusammen zu einem Block. Dieser Block als solches ist nur gültig solange bei Klammern vorhanden sind. Wird nun eine Klammer entfernt, kann man entsprechende Fehlermeldung einbauen um darauf zu reagieren.

Die Vielsprachigkeit der GUI

Die GUI stellt das Bindeglied zwischen Benutzer, Dateisystem, Compiler und Virtueller Maschine dar. Sie stellt also mindestens vier Schnittstellen zur Verfügung, die in unserem speziellen Fall alle eine andere Sprache reden: grafische Elemente und sichtbarer Text beim Benutzer, Zugriffe auf das Dateisystem, das Lesen und Schreiben in die plattformunabhängige Konsole beim Compiler und schliesslich Sockets bei der Virtuellen Maschine.

Alle diese Schnittstellen bringen Voraussetzungen mit. Diejenigen des GRINJ-Projektes wurden in RFCs definiert und von uns umgesetzt.

Der Compiler spricht konsolisch

Der Befehl zum Kompilieren einer GRINJ-Datei erfolgt relativ simpel über die Konsole. Es erfolgt die Bestätigung des erfolgreichen Ausführens ebenfalls in die Konsole. Der Compiler braucht also nur gerade während diesen wenigen Augenblicken ausgeführt zu werden.

Hier eine Kommunikation über Sockets zu realisieren wäre also nicht nur unverhältnismässig, sondern auch ressourcenintensiv, da ja der Compiler im Hintergrund ständig weiterlaufen müsste um die Verbindung aufrecht zu erhalten.

Die Standard-Java-Bibliothek bietet uns die Schnittstelle zur Konsole sehr komfortabel an:

```
Process myProc = Runtime.getRuntime().exec(myCommand);
```

Auch das Auslesen der Compiler-Antwort erfolgt simpel:

```
String answer = "";
int tmp = 0;
while ((tmp = myProc.getInputStream().read()) != -1) {
      answer += (char) tmp;
}
```

In der GUI werden die Konsolen-Kommandos im Hintergrund verschickt und die Antworten des Compilers auf der GUI-Konsole präsentiert. Es erfolgt nur die Auswertung des Rückgabe-Strings (erfolgreich/nicht erfolgreich). Der Benutzer muss selbständig Massnahmen ergreifen, falls die Kommunikation nicht klappt oder der Compiler Fehler zurückmeldet.

Die Virtuelle Maschine versteht Sockets

Ganz andere Aspekte bringen klar den Vorteil der Sockets bei der Kommunikation zur VM hervor. Wie z.B. die Java-VM soll auch die GRINJ-VM im Hintergrund laufen. Ein GRINJ-Programm in Ausführung muss angehalten und unterbrochen werden können. Stackvariablen müssen in der GUI ausgewertet werden. Ja, der Benutzer soll sogar die Outputs seines GRINJ-Programms einsehen und Inputs tätigen können.

Das verlangt erst einmal eine besondere GUI-Konsole. Sie soll verschiedenartige Meldungen gleichzeitig im selben Fenster anzeigen und sogar User-Inputs entgegennehmen können. Das Eclipse-Framework bietet uns mit der `org.eclipse.ui.console.IOConsole` das geeignete Werkzeug.

Es können `IOConsoleOutputStream`s erzeugt und auf einen `IOConsoleInputStream` zugegriffen

werden. Wir verwenden unterschiedliche OutputStreams um die Ausgaben der Kommunikationspartner in verschiedenen Farben darzustellen.

Doppelspuring fährt es sich einfacher
Die GUI baut die Kommunikation zur VM über zwei parallele Socket-Verbindungen auf. Voraussetzung ist, dass die VM mit den Parametern –`debug-server`=[Port-Nummer] `--io-server`=[Port-Nummer] gestartet wurde. Seitens der VM besteht die Voraussetzung, dass eine Reihenfolge beim Aufbau der beiden Socket-Verbindungen einzuhalten ist. Das führt auf der Seite der GUI zur Konsequenz, dass die beiden Socket-Verbindungen nur gleichzeitig in Betrieb sein können. Fällt eine aus, wird auch die zweite gekappt.

Die erste Verbindung kommt ohne Protokoll aus. Sie dient nur der Ein- und Ausgabe des GRINJ-Programms. Textzeilen aus der GRINJ-Konsole werden mit ENTER mit Hilfe des InputStreams an die VM verschickt.

Von Seiten der VM werden die Nachrichten gepollt. Das heisst, ein eigener Thread läuft während der ganzen Verbindungsdauer und schreibt Programm-Outputs der VM auf die GUI-Konsole.

Die zweite Verbindung ist für Befehle der GUI an die VM reserviert und folgt den Protokoll-Definitionen in `RFC_0004`. Da die Kommandos an die VM nur in einer bestimmten Reihenfolge Sinn machen, wollten wir den Benutzer dabei unterstützen, diese Reihenfolge einzuhalten.

Benutzer ist König
Jeder geöffnete Editor besitzt einen (VM-Kommunikations-) Status. Davon abhängig werden Menu-Einträge und Buttons sowie Shortcuts aktiviert und deaktiviert. Zwei Use Cases auf Systemebene dienten uns als Leitfaden:

GRINJ-Datei ausführen:

1. Datei-Laden-Befehl an VM
2. Erhalte Antwort „ready“
3. Ausführen-Befehl an VM
4. Erhalte Antwort „running“
5. Statusabfrage an VM schicken solange Antwort „running“
6. Erhalte Antwort „terminated“ - Programm ausgeführt!

GRINJ-Datei debuggen:

Datei-Laden-Befehl an VM

1. Erhalte Antwort „ready“
2. Breakpoint-Setzen-Befehl an VM, so lange Breakpoints vorhanden und Antwort „ready“
3. Ausführen-Befehl an VM
4. Erhalte Antwort „running“
5. Statusabfrage an VM schicken solange Antwort „running“
6. Erhalte Antwort „stopped“
7. Stackabfrage-Befehl an VM
8. Erhalte Antwort „stopped“ (inkl. Stack)
9. Globals-Stackabfrage-Befehl an VM
10. Erhalte Antwort „stopped“ (inkl. Globals)
11. Fortfahren-Befehl an VM
12. Erhalte Antwort „running“
13. Statusabfrage an VM schicken solange Antwort „running“
14. Erhalte Antwort „terminated“ - Programm ausgeführt!

Auf jeden Befehl folgt postwendend die Antwort. Voraussetzt, die VM läuft und die Verbindung besteht noch. Genau dieser Punkt hat dann auch zu einigen Herausforderungen in der Umsetzung geführt.

Threads bringen Stablilität
Wenn eine Socket-Verbindung nämlich erst mal aufgebaut wurde, ist es nicht so einfach herauszufinden, ob sie auch immer noch besteht. Nachrichten können schliesslich weiterhin problemlos versendet werden. Falls aber eine Antwort erwartet wird, kann das ganze Programm unter Umständen so lange lahm liegen, bis jene auch eintrifft. Es bleibt der Vorstellungskraft des Lesers vorbehalten, sich auszumalen was geschieht, wenn gar keine Antwort mehr erfolgt.

Wir haben für unsere GUI entschlossen, einen Timeout einzuführen, der die Verbindung als unterbrochen betrachtet, wenn nach einer bestimmten Zeitspanne keine Antwort erfolgt. Die eigens zur Kommunikation mit der VM eingeführte Klasse mit dem selbstsprechenden Namen `VMCommunicator` ist als Singleton realisiert. Damit gehen wir sicher, dass wirklich nur *eine* Verbindung existiert und können auf die VM von verschiedenen Orten zugreifen.

Kernelemente dieser Kommunikation arbeiten mit nebenläufigen Threads und befinden sich mitunter in den folgenden zwei Methoden, die wir hier näher betrachten wollen:

```
	/**
	 * Eine Nachricht (= Befehl) an den Server der VM senden.
	 *
	 * @param msg
	 *            Nachricht (bzw. Befehl) fÃ¼r die VM.
	 * @return Antwort auf den gesendeten Befehl.
	 * @throws IOException
	 */
	private String sendMsg(String msg) throws IOException {
		this.tempSendMessage = msg;
		this.tempAnswer = null;

		if (!connectionIsEstablished()) {
			try {
				establishConnection();
			} catch (IOException e) {
				closeConnection();
				throw new IOException(
					"Die Verbindung zur VM wurde unterbrochen und
konnte nicht wiederhergestellt werden.");
			}
		}

		Thread sendThread = new Thread(new Runnable() {
			public void run() {
				out.println(tempSendMessage + "\n");
				try {
					tempAnswer = in.readLine();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		});
		sendThread.start();
		waitForAnswer(500);

		if (tempAnswer == null) {
			closeConnection();
			throw new IOException(
				"Die VM reagiert nicht. Möglicherweise muss sie neu
gestartet werden oder die Verbindung wurde unterbrochen.");
		}

		return tempAnswer;
	}
```

Kommentar zum Java-Code:
Es als erstes folgt die Überprüfung der Verbindung. Falls sie noch nicht aufgebaut wurde, wird das jetzt versucht. Der Thread `sendThread` wird gestartet, sendet das Kommando und wartet auf die Antwort. Währenddessen läuft der Main-Thread weiter und ruft die Methode `waitForAnswer(500)` auf, die im weiteren erläutert werden wird.
Konnte nach der Wartephase immer noch keine Antwort registriert werden, so wird die Verbindung als unterbrochen angesehen und allfällige Connections geschlossen mit dem Aufruf von `closeConnection()`.

```
/**
 * @param timeout
 *            so lange wird auf die Antwort gewartet, ansonsten wird die
 *            Verbindung als unterbrochen angesehen
 */
private void waitForAnswer(int timeout) {
        try {
                for (int i = 0; i < timeout && tempAnswer == null; i++) {
                        Thread.sleep(1);
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}
```

Kommentar zum Java-Code:
Während eine Anzahl Millisekunden (`int timeout`) wird der Main-Thread schlafen gelegt und überprüft fortwährend, ob die Antwort bereits eingetroffen ist. In diesem Fall würde die Warteschleife nämlich abgebrochen.

Schlusswort

Im Laufe der Projektarbeit hat sich bei uns ein Ehrgeiz entwickelt, die GUI benutzerfreundlich bis ins Detail zu gestalten. Jetzt, kurz vor der Abgabe der Arbeit, müssen wir den Tatsachen ins Auge sehen und stellen ernüchtert fest: Eine GUI ist eine Baustelle, die nie fertig ist. Es findet sich immer ein Punkt, der noch verbessert werden kann; immer noch ein Ausnahmefall, dessen negative Folgen man dem Benutzer vorenthalten möchte.

Es sind doch immer ähnlich Elemente, die eine GUI dem Benutzer zur Verfügung stellt: konfigurierbare Shortcuts, ein- und ausblendbare Buttons, Rechtsklick-Menues, Syntax-Highlighting, etc.
Je tiefer wir ins Eclipse-Framework blicken, desto mehr finden wir solche Komponenten vor, die uns grosse Teile der Arbeit abnehmen. Eine aufwendige GUI lässt sich so vielleicht doch in knapper Frist realisieren.

Verwendete Software

Entwicklungsumgebung:	Eclipse 3.2 M6
Java Version:	JRE: Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Logging:	log4j-1.2.13

Schlusswort

Das ganze Projekt war an der Grenze von Herausforderung und Überfoderung. Dies lag daran, dass wir kein Wissen(oder sehr wenig) über Compilerbau, virtuelle Maschinen oder Eclipse-Plugin vorhandenen war. Dadurch war die ganze Aufgabe nahe an der Realität.

Der Schlüssel zum Erfolg war das Teamwork, welches sehr gut funktioniert hat. Ab und zu gab es kleine Unstimmigkeiten / Missverständnisse, die darauf zurück zu führen sind, weil ein Teil der Klasse bereits ein gemeinsames Projekt hinter sich hatte und viele Dinge für diese Personen schon klar war.

Im Grossen und Ganzen kann man sagen, dass das Projekt wirklich gelungen ist und jeder hat seinen Teil dazu beigetragen.

Grinj rulez

Anhang

RFC_0001_veraltet

Dieser RFC wurde geloescht, weil die "alte" Sprachdefinition (Brainf**k) nicht akzeptiert wurde.

RFC_0002_opcodes

TOC

---

1. Anm. zur VM

Die Definition der Hardware-Emulation ist nicht Gegenstand dieser RFC. Der genaue Aufbau und Beispiele bleibt der Emulations-Doku vorbehalten.
Trotzdem werden hier nochmals kurz alle vorhandenen Komponenten aufgelistet:
* Programm-Speicher, Byte-Array mit OP-Codes
* Daten-Speicher, Int-Array mit globalen Daten
* Stack, Int-Array mit Stack-Daten
* BP: Basepointer, Basis-Adresse des aktuellen Kontextes
* SP: Stack-Pointer, aktuelle Position im Stack
* PC: Programm-Counter, aktuelle Position im Programm-Speicher

Die VM implementiert ein 32Bit-System. Der Daten-Speicher, Stack und alle Register haben eine Groesse von 32Bit.
Alle Daten werden als Big-Endian abgelegt.

2. Definition

Nachfolgend sind die OP-Codes der ersten VM-Version definiert.

OP (dec)	OP (hex)	Typ	Beschreibung	VM-Aktion
20	14	Stack	CONST n	Push(n)
21	15	"	LOAD a	Push(stack[bp+a])
22	16	"	LOADG a	Push(globals[a])
23	17	"	STO a	stack[bp+a]=Pop()
24	18	"	STOG a	globals[a]=Pop();
40	28	Arithmetik	ADD	Push(Pop()+Pop());
41	29	"	SUB	Push(-Pop()+Pop());
42	2a	"	DIV	x=Pop(); Push(Pop()/x);
43	2b	"	MUL	Push(Pop()*Pop());
44	2c	"	NEG	Push(-Pop());
60	3c	Vergleiche	EQU	if (Pop()==Pop()) Push(1); else Push(0);

```
 61       | 3d        |  "          | LSS          | if (Pop()>Pop()) Push(1); else Push(0);
 62       | 3e        |  "          | GTR          | if (Pop()<Pop()) Push(1); else Push(0);
 63       | 3f        |  "          | LSE          | if (Pop()>=Pop()) Push(1); else Push(0);
 64       | 40        |  "          | GTE          | if (Pop()<=Pop()) Push(1); else Push(0);
 80       | 50        | Spruenge    | JMP  a       | pc = a;
 81       | 51        |  "          | FJMP a       | if (Pop()==0) pc=a;
100       | 64        |  I/O        | READ         | Push(ReadInt());
101       | 65        |  "          | WRITE        | WriteIntLine(Pop());
102       | 66        |  "          | READC        | Push(ReadChar());
103       | 67        |  "          | WRITEC       | WriteChar(Pop());
120       | 78        |  Methoden   | CALL a       | Push(pc+2); pc=a;
121       | 79        |  "          | RET          | pc = Pop(); if (pc==0) return;
122       | 7a        |  "          | ENTER a      | Push(bp); bp=sp; sp+=a;
123       | 7b        |  "          | LEAVE        | sp=bp; bp=Pop();
153       | 99        |  "          | NOP          |
----------+-----------+-------------+--------------+-----------------------------------------
```

3. Beispiel

Addition mittles globalen Variabeln:

```
Pseudocode | Mnemonicode (in Bytes) | Kommentar
-----------+------------------------+-------------------------------------------------------------------------------
VAR a,b;   |                        | jede Variable bekommt einen Nummer, a -> 0, b -> 1
b = 42;    | 020 000 000 000 042    | 42 auf den Stack legen
           | 024 000 000 000 001    | den auf den Stack gelegten Wert (42) der Variable mit Nr 1 zuweisen
READ(a);   | 100                    | Auf Eingabe auf xy warten
           | 024 000 000 000 000    | Eingelesener Wert der Variable 0 zuweisen
a = a+b;   | 022 000 000 000 000    | Wert der Varibale 0 auf Stack legen
           | 022 000 000 000 001    | Wert der Varibale 1 auf Stack legen
           | 040                    | beiden vorhergehenden Wert zusammenzaehlen(2 x Pop()) und auf Stack legen
           | 024 000 000 000 000    | Wert auf Stack der Variable mit Nr 0 zuweisen
WRITE(a);  | 022 000 000 000 000    | Wert der Variable 0 in den Stack laden
           | 101                    | Pop() auf xy ausgeben
-----------+------------------------+-------------------------------------------------------------------------------
```

Ausgabe im "echten" GRINJ, einmal in Pseudo-Assembler, einmal in Zahlencode:

1: ENTER 2
4: CONST 42
7: STO 1
10: READ
11: STO 0
14: LOAD 0
17: LOAD 1
20: ADD
21: STO 0
24: LOAD 0
27: WRITE
28: LEAVE
29: RET

1: 122 2
4: 020 42
7: 023 1
10: 100
11: 023 0
14: 021 0
17: 021 1
20: 040

21: 023 0
24: 021 0
27: 101
28: 123
29: 121

RFC_0003_fileformat

Fileformat der Executable

TOC

---

1. Inhalt der Binary-Datei

Die Executable-Datei mit dem GRINJ-Opcodes
hat folgenden Inhalt:

- Header: (Meta)Informationen vom Compiler fuer die VM oder das OS (Shell)
- Body: besteht aus zwei Teilen
- GLOBALS: Initialwerte fuer die Globalen Variabeln
- CODE: Groesse der Programm-Code und der Programm-Code
- Debug: Debug-Informationen

2. Definition

Header-Block:

Aufbau:
1. String "#!/usr/local/bin/grinj"
2. String-Abschluss "\n"
3. Beliebiger Text
4. Abschluss mit ASCII-Code 255

Globals

Die Globalen Initialwerte bestehen
aus zwei Integern:
1. 32Bit: Variablen-Nummer
2. 32Bit: Variablen-Wert

Am Ende der Globals folgt immer
Abschluss-Code 255

Code

Der Code-Block besteht aus der Groesse des Programm-Codes(Op-cods) und den GRINJ-Op-Codes.
Am Ende des Code-Blocks steht immer das Kommando "RET" und der Abschluss-Code 255

Debug

Die Debug-Informationen sind vollstaendig as ASCII-TEXT
aufgebaut und haben folgende Informationen:

Adresse Zeilennummer Dateiname
..

3. Beispiele

Achtung: Bei allen Beispielen wird die Endmarke 255
und die Opcodes in ASCII dargestellt, nicht als Binaere Daten...

3.1. Minimale Datei:

#!/usr/local/bin/grinj
255
255
000 000 000 006
122
000
000
000
000
121
255

Die Endmarken Trennen Header, Globals, Code und Debug-Block.
Jedes Programm startet mit Op-Code 122 (ENTER) und wird mit Op-Code 121 abgeschlossen:
> ENTER 0
> RET

3.2. Addition mit globalen Daten:
Pseudocode:

VAR a = 23;
VAR b = 42;

```
a = a+b;
WRITE(a);
--------------------------
```

Ergibt die Datei

```
---------------------------
#!/usr/local/bin/grinj
255
000 000 000 000 000 000 000 023
000 000 000 001 000 000 000 042
255
000 000 000 028
122 000 000 000 000
022 000 000 000 000
022 000 000 000 001
040
024 000 000 000 000
022 000 000 000 000
101
121
255
1000 4 addition.grinj
1016 5 addition.grinj
---------------------------
```

Bemerkungen zur Datei:
Aus Gruenden der Lesbarkeit wurde die Zahlen mit drei Stellen versehen und mit einem Abstand. Im richtigen Opcode gibt es keine Leerzeichen.
Jeder neue Opcode steht auf einer neuen Linie. Der Zeilenumbruch ist im richtigen Opcode nicht vorhanden

RFC_0004_debug

Austausch von Debug-Infos zwischen VM und GUI

TOC

1. Debug-Kommunikation

```
-------------------------------------------------
```

Die Kommunikation zwischen VM (Debugger) und GUI wird mittels Sockets realisiert.

2. Protokoll

Kommando: "load binaryfilename"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "run"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "stop"
Antwort : Status-Code VM-State Zeilennummer Filename

Kommando: "break zeilennummer filename"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "clear zeilennummer filename"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "cont"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "state"
Antwort : Status-Code VM-State Zeilennummer Filename

kommando: "stack"
Antwort : Status-Code VM-State Zeilennummer Filename Grösse Address1 Value1 AddressN ValueN

kommando: "global"
Antwort : Status-Code VM-State Zeilennummer Filename Grösse Address1 Value1 AddressN ValueN

3. Rueckmeldung von der VM (Antwort)

Der Aufbau der Status-Meldungen wird mittels eines Abstandes voneinander getrennt.

Beispiel:
00 03 23 beispiel.grinj

=> die Binary der Datei beispiel.grinj steht bei Zeile 23 im Source und ist in Betrieb.

Status-Codes:	- 00 EV_DONE	// Command executed
(EventState)	- 01 EV_WRONG_STATE	// Command not possible
	- 02 EV_NO_BINARY	// No Binary loaded yet
	- 03 EV_WRONG_PARAM	
	- 04 BAD_COMMAND	// Wrong Command sent
VM-State:	- 00 UNDEF = 0	// VM in undef state (no binary loaded)

(State)	- 01 INIT	// Initialize VM (load Binary)
	- 02 READY	// Ready for running (File loaded)
	- 03 RUNNING	// VM is running
	- 04 STOP	// stop VM asap (breakpoint or by user)
	- 05 STOPPED	// VM stopped
	- 06 TERMINATED	// Program terminated (end or internal error)
	- 07 SHUTDOWN	// Terminate Binary and finish VM
	- 08 BLOCKED	// VM blocked

3.1. Return Wert (Antwort)

Fuer einzelne Antworten der VM, machen die Angaben von Zeilennummer und Filename nur bedingt Sinn.
In Faellen wo Zeilennummer und Filename nicht zurueckgegeben werden koennen, wird ein ? zurueckgegeben.