Technische Universität München SS02
Fakultät für Informatik
Lehrstuhl Sprachen und Beschreibungsstrukturen in der Informatik
Prof. Dr. Helmut Seidl

Systementwicklungsprojekt

Visual Emugen

Generierung einer Komponente zur Visualisierung von EMUGEN

Michael Petter
28. März 2003

Inhaltsverzeichnis

1 Einleitung **4**

2 Planungsphase **4**
2.1 Ausgangslage 4
2.2 Ziele 5
2.3 Rahmenbedingungen und Risiken 6
2.4 Handlungalternativen 7
2.5 Fazit 8

3 Realisierung **8**
3.1 Technologiefestlegung 8
3.2 Rahmenbedingungen 9
3.3 Design 9
3.3.1 Paketstruktur 9
3.3.2 Interoperation der Pakete 10
3.3.3 Generierung 11
3.4 Implementierung 12
3.4.1 Der Rahmen 12
3.4.2 Die Hauptanwendung 13
3.4.3 Das zentrale Dokument 15

3.4.4 Der Umwandlungsvorgang . . . 16
3.4.5 Die Grafikengine . . . 18
3.4.6 Syntaxhighlighting . . . 18
3.4.7 Syntaxergänzung . . . 20
3.4.8 Fehlerkennzeichnung . . . 22
3.4.9 Antintegration . . . 22
3.4.10 Emugen-Weiterleitung . . . 22

4 Dokumentation **23**
4.1 Visual Emugen Benutzung . . . 23
4.1.1 Aufruf von der Kommandozeile . . . 23
4.1.2 Aufruf aus Ant . . . 23
4.1.3 Grafische Oberfläche . . . 24
4.1.4 Start als Applet . . . 25
4.2 Entwicklung . . . 27
4.2.1 Visual Emugen umwandeln . . . 27
4.2.2 Graphendarstellung . . . 28
4.2.3 Java-Compiler verwenden . . . 30
4.2.4 Syntaxhighlighting . . . 31
4.2.5 Fehlerkennzeichnung . . . 32
4.2.6 Syntaxvervollständigung . . . 33
4.2.7 SplashScreen . . . 33
4.2.8 Installer . . . 34

5 Fazit **35**
5.1 Bewertung . . . 35
5.1.1 Erfüllung der Ziele . . . 35
5.1.2 Anwendbarkeit . . . 36
5.2 Ausblick . . . 36

Abbildungsverzeichnis

1 Paketstruktur in VisualEmugen . . . 11
2 Stark vergröberte Übersicht über die Paketschnittstellen . . . 12
3 Umwandlungsvorgang mit ANT . . . 13
4 Verzeichnisstruktur mit Erläuterungen . . . 14
5 Schema des Grace-Unterpaktes . . . 19
6 Syntaxhighlighting im Detail . . . 20
7 Aufbau eines Suffix Baum . . . 21
8 Syntaxvervollständigung mit einem Suffix Baum . . . 21
9 Ablauf zur Fehlerkennzeichnung . . . 22

10 Screenshot nach dem Start 25
11 Syntaxhighlighting mit Syntaxfehler und Syntaxergänzung 26
12 Baumansicht 26
13 Generierung von ausführbaren Dateien 27

1 Einleitung

Emugen ist ein System zur Generierung von Java-Swing basierten Benutzerschnittstellen. Ausgehend von der Beschreibung der gewünschten Datenstruktur erzeugt *Emugen* die entsprechende Dialogmaske. Zusätzlich kann der Benutzer an einigen Stellen Erweiterungen an der automatisch erzeugten Oberfläche vornehmen, und so seine Business-Logik einfügen.

Emugen ist ein mächtiges Werkzeug für RAD (Rapid Application Development). Ein großer Nachteil schränkt die Benutzbarkeit von *Emugen* für Jedermann jedoch stark ein: Die Entwicklung und Umwandlung von mehr oder weniger korrektem *Emugen* -Code in ein fertiges Programm muss über einen externen Editor und manuelle Compileraufrufe getätigt werden, was viele Benutzer abschreckt oder gerade Anfänger vor große Probleme stellt. An dieser Stelle soll *Visual Emugen* ansetzen, dass eine attraktive einfache Oberfläche für *Emugen* bereitstellen soll...

2 Planungsphase

In diesem Kapitel werden die Kriterien aufgestellt, die ausschlaggebend für die Implementierungsphase sind. Dazu wird die Ausgangslage analysiert, in der die Situation des herkömmlichen Umgangs mit *Emugen* beschreibt. Darauf folgt die Beschreibung der Ziele, die durch die Einführung von Visual Emugen erreicht werden sollen. Die dabei zu beachtenden Rahmenbedingungen, die von aussen vorgegeben sind, sowie die potentiellen Risiken, die einer Realisierung im Wege stehen, werden im nächsten Abschnitt erwähnt. Für die nun festgelegte Aufgabe werden die Handlungsalternativen festgehalten, die nun offenstehen. Eine Entscheidung dieser bildet den Abschluss dieses Kapitels.

2.1 Ausgangslage

Der bisherige Arbeitsablauf bei der Verwendung von *Emugen* beinhaltet das zuerst das manuelle Erstellen von *Emugen* dateien über einen selbstgewählten beliebigen Texteditor. Im Anschluss daran muss der Benutzer die `.emu` Datei abspeichern, und dann per Kommandozeile eine Folge von Befehlen eintippen wie:

```
# java -jar emugen.jar -f -fo meinedatei.emu
# javac -classpath emu_runtime.jar:. MeineKlasse.java
# java -classpath emu_runtime.jar:. MeineKlasse
```

Die Umwandlung von `.emu` Dateien in Java-Klassen wird dabei stabil und plattformunabhängig von dem bereits existierenden Werkzeug *Emugen* vorgenommen. Dieses wird intensiv von Alfons Brandl weiterentwickelt, und ist demnach ständig im Wandel begriffen.

Diese Schnittstelle zu *Emugen* ist robust, weist allerdings einige gravierende Nachteile auf:

- gewöhliche Editoren bieten oft nur mangelhafte Unterstützung bei der Entwicklung von `.emu` Dateien. Diese werden behandelt wie gewöhliche Textdateien, erfordern also eine vollständige Eingabe aller Schlüsselwörter und Variablennamen. Dies ist auf Dauer anstrengend und fehlerträchtig.
- Syntaxfehler im `.emu` Code werden erst zur Umwandlungszeit festgestellt, so dass der Benutzer unter großer Mühe und Zeitaufwand den Fehler im Quellcode suchen und ausbessern muss.
- die Umwandlung von `.emu` Code in `.class` Dateien erfolgt manuell von Hand, und ist daher zeitaufwändig und fehlerträchtig.
- eine folgende Verpackung des Codes in eine (lauffähige) `.jar` Datei wird aufgrund der aufwändigen Syntax in der Praxis nie vollzogen, so dass die Weitergabe von lauffähigen Programmpaketen erschwert ist. Dadurch wird der Nutzungsgrad der mit *Emugen* erstellten Programme in der Praxis stark eingeschränkt.

Eine Ähnliche Situation war bereits vor einigen Jahren bei der Entwicklung eines anderen Werkzeugs entstanden - *Classgen* existierte analog zu *Emugen* anfangs nur als Kommandozeilenwerkzeug. Für dieses Tool wurde mit *Visual Classgen* eine Entwicklungs-Oberfläche geschaffen. Da sich die beiden Projekte in einigen Bereichen ähneln, lassen sich einige für *Visual Emugen* relevante Erfahrungen ableiten:

- die Entwicklung der grafischen Oberfläche *Visual Classgen* fand zu einem Zeitpunkt statt, als die Entwicklung des zugrunde liegenden Werkzeugs noch nicht abgeschlossen war. Für eine erfolgreiche Erstellung der Oberfläche war jedoch die Abstützung auf eine feste Version von *Classgen* notwendig, da die entsprechenden Klassen fest in die Oberfläche mit integriert wurden. Als Resultat entstanden nun zwei verschiedene Zweige von *Classgen* , die bislang noch nicht wieder miteinander vereint werden konnten: Eine grafisch ansprechende, aber veraltete Version, sowie eine weiterentwickelte Kommandozeilenvariante
- die Anwendung *Visual Classgen* , die in Ihrer Funktionalität sehr stark der des erwünschten Werkzeugs *Visual Emugen* ähnelt, erschwert durch die sehr spezielle verflochtete Struktur ihre Wiederverwertung als Grundgerüst für *Visual Emugen*
- die fehlende Entwickler/API Dokumentation in Kombination mit der sowohl unübersichtlichen als auch umfangreichen Klassensammlung erschwert Wartung, Weiterentwicklung und Adaption des Programms

2.2 Ziele

Das primäre Ziel von *Visual Emugen* stellt sicherlich die ansprechende Visualisierung der `.emu` Dateien dar. Das erstellte Datenmodell soll dabei analog zu *Visual Classgen* in einem

Graphen auf dem Bildschirm dargestellt werden. Gleichzeitig soll es möglich sein, zu Dokumentationszwecken diese Graphen auch als `.jpg` Dateien abzuspeichern.

Weiterhin ist für die Entwicklung mit *Emugen* noch die Bereitstellung einiger Zusatzmerkmale, wie man sie aus professionellen Entwicklungsumgebungen wie NetBeans oder Eclipse kennt, wünschenswert. Dazu zählen:

- die farbige Kennzeichnung von Emugen und Java Schlüsselworten
- die Kennzeichnung von Syntaxfehlern bereits im Editor
- ein Mechanismus zur automatischen Vervollständigung der Codeeingaben des Benutzers
- der automatisierte Ablauf der erforderlichen Compile-Vorgänge auf Knopfdruck
- Unterstützung bei der Weitergabe der mit *Visual Emugen* entwickelten Programme

Schließlich soll die einfache Benutzung von *Visual Emugen* als Paradigma für die Entwicklung des Werkzeugs gelten.

Von der technischen Seite her betrachtet, soll *Visual Emugen* weitere Kriterien erfüllen. So ist zum Beispiel die einfache, transparente Integration von *Emugen* unbedingt nötig, um zu Wartungszwecken sauber zwischen den Quellcodes der verschiedenen Autoren trennen zu können. Trotzdem soll es auch für Laien möglich sein, sehr schnell ein "aktualisiertes" *Visual Emugen* zu erstellen.

Weiterhin soll beim Design der Software der Gedanke der Modularisierung, Objektorientierung und funktionalen Trennung beachtet werden, so dass eine leicht wartbare Anwendung entsteht, deren Module sehr leicht wiederverwendet werden können. Denkbar wäre die Realisierung in zwei Teilen, einem allgemein verwertbaren "Oberflächenkonstruktionskit" und einer spezifisch auf die Verwendung mit Emugen ausgelegten Klassensammlung.

Nachdem möglicherweise die grafische Ausgabe durch das Werkzeug *Grace* vorgenommen wird, und für *Grace* nur eine unzureichende Menge an Beispielanwendungen vorhanden sind, liegt die Verwendung von *Visual Emugen* als Beispiel für die Anwendung des Toolkits *Grace* nahe. Beim Design von *Visual Emugen* ist daher auf eine saubere, einheitliche, beispielhafte und sprechende Integration der Konzepte von Grace zu achten.

Schließlich liegt, wie bei allen Swing-basierten Programmen, der Anspruch der Ausführbarkeit als Applet auf Internetseiten nahe. Dieser sollte unter Beachtung gewisser Konventionen bei der Erstellung des Hauptfensters keine zusätzlichen Mühen bereiten.

2.3 Rahmenbedingungen und Risiken

Beim Erreichen der oben genannten Ziele sollte stehts auf die folgenden Punkte geachtet werden:

- es muss garantiert werden, dass *Visual Emugen* auf allen Systemen läuft, auf denen auch *Emugen* bisher lief. Ansonsten käme es zu einer unnötigen Einschränkung des potentiellen Benutzerkreises.

- zu jeder Zeit muss die Trennung von *Emugen* und *Visual Emugen* Bestandteilen möglich sein, um eine getrennte Fortentwicklung beider Werkzeuge zu ermöglichen. Trotzdem sollte eine Integration beider Bausteine in ein gemeinsames Paket nicht unnötig erschwert werden.

- um jeden Preis ist die unnötige "Aufblähung" des Quellcodes zu einem "undurchdringlichen Klassendschungel" zu vermeiden, um aussenstehenden Personen einen leichteren Einstieg in die Quellcodes offen zu halten.

- eine ausführliche Dokumentation der geschaffenen APIs ist aus selbigen Grund unumgänglich

2.4 Handlungalternativen

Um die obigen Kriterien zu erfüllen, sind mehrere Schritte denkbar:

- Anpassung des vorhandenen VisualClassgen Eine Bearbeitung des bestehenden Werkzeuges *Visual Classgen* ist technisch möglich, da die Quellen offen vorliegen. *Visual Classgen* ist in Java geschrieben, also plattformunabhängig. Das Problem an dieser Lösung ist jedoch, dass die Struktur von *Visual Classgen* sehr verwoben ist, und nur schlecht dokumentiert ist. Eine Anpassung der Klassen würde die Übersichtlichkeit des Codes nur noch weiter verschlechtern. Bestehende Designschwächen werden in dieser Lösung einfach übernommen.

- Entwurf eines Plugins für einen bestehenden Editor Ein weiterer Weg wäre der Entwurf eines Plugins für etablierte Editoren wie Emacs oder JEdit. Dazu muss zuerst herausgefunden werden, auf welche Weise und in welcher Sprache das Plugin verfasst werden muss. Dadurch müssen eventuell proprietäre Techniken angewandt werden. Gleichzeitig macht man sich abhängig vom Entwicklungsstand des "Wirtprogrammes".

- Entwicklung der Oberfläche mit Emugen selbst Bei der Entwicklung eines grafischen Werkzeuges für Emugen stellt sich naturgemäß die Frage, ob man nicht auch das Werkzeug selbst mit Emugen entwickelt. Der sich dabei einstellende Vorteil wäre wie bei Emugen üblich eine Abstraktion des Dialogs, mit der Einschränkung, dass die Entwicklung des Softwaretools nicht mehr so stark beeinflusst werden kann.

- Entwicklung einer eigenen Oberfläche Die Eigenentwicklung der Oberfläche stellt einen vor das Problem, das Rad eventuell nochmal erfinden zu müssen, mit dem Vorteil von Anfang an eine saubere und unabhängige Konzeption durchsetzen zu können.

2.5 Fazit

Der Entwurf eines Plugins für einen bestehenden Editor gestaltet sich sehr kompliziert und bietet auch nicht die gewünschte Selbständigkeit des geschriebenen Programmes. Diese Lösung fällt also von vornherein für *Visual Emugen* weg. Auch die Entwicklung mithilfe Emugen selbst erscheint wenig attraktiv, da die Hauptmerkmale von *Visual Emugen* im Bereich des Editors und der auf Visualisierungsseite liegen, was beides Themen sind, die von Emugen selbst nicht abgedeckt werden.

Fest steht damit also die Entwicklung von Emugen als eigenständiges Java-Programm. Die Anpassung von *Visual Classgen* würde zwar zum gewünschten Erfolg führen, jedoch kann durch die Übernahme des kompletten Quellcodes nicht mehr garantiert werden, dass die interne Struktur schlüssig und sauber realisiert ist. Die Lösung wird daher sein, *Visual Emugen* komplett neu zu entwickeln, aber dabei die Erfahrungen mit *Visual Classgen* zu berücksichtigen. Gute Strukturen und Konzepte aus *Visual Classgen* sollen ruhig übernommen werden.

3 Realisierung

Visual Emugen ist als Java-Programm realisiert, und stützt sich auf einige Quellcodegenerationssprachen ab, die an der TU-München entwickelt wurden.

3.1 Technologiefestlegung

In Zusammenhang mit der Entwicklung von *Visual Emugen* kamen folgende Technologien zum Einsatz:

java Version 1.4+ als Basisarchitektur, empfehlenswert ist jedoch das Release 1.4.2, da die Performance dieser Version um einiges höher ist.

ant Version 1.5.1+ übernimmt die Übersetzungskoordination und stellt die einzelnen Releases der Software zusammen.

grace Version 1+ dient der Generierung einer Graphendarstellung der *Emugen* Spezifikationen mittels einer eigenen Beschreibungssprache. Es generiert Java-Klassen.

JFlex Version 1.3.5+ stellt einen mächtigen Scanner zur Verfügung, der das Syntaxhighlighting für *Visual Emugen* übernimmt.

emugen Version 20030612+ ist das Basissystem von *Visual Emugen* und wird von *Visual Emugen* zur Umwandlung des Quellcodes in Java-Dateien verwendet.

emu_runtime Version 20030612+ wird zur Ausführung von fertigen *Emugen* Programmen benötigt.

Bei der Entwicklung des Programmes *Visual Emugen* wurde darauf geachtet, dass der Quellcode sehr modular gehalten wurde, um eine spätere Wiederverwendung für weitere *Visual* Projekte zu ermöglichen. Insbesondere entstand dabei das *compilerbau* Paket, welches allgemeine Routinen für die Erstellung von integrierten Entwicklungsumgebungen (IDE) verwendet werden können. Zusätzlich dazu kann man *Visual Emugen* dank seiner klaren Strukturierung als Demonstrationsanwendung für die Benutzung von *Grace* sehen.

3.2 Rahmenbedingungen

Die Unterstützung des Umwandlungsvorganges durch *Ant* zieht einige Konsequenzen nach sich: Es ist mit diesem Werkzeug sehr gut möglich, Quellcode und Bytecode von Javaprogrammen zu trennen, auch sind verschiedene Generierungsschritte sehr gut in den Umwandlungsprozess mit einbindbar. Wenn *Ant* richtig eingerichtet ist, so braucht sich der Programmierer keine Gedanken mehr darum machen, welche Generierungsschritte er nun der Reihe nach ausführen muss, um zu dem von ihm gewünschten Programmm zu gelangen oder welche Bibliotheken er in den Klassenpfad einbinden muss, damit der Umwandlungsversuch nicht fehlschlägt. Diese Eigenschaft rückt besonders im Hinblick auf die dynamische Einbindung der neuen *Emugen* Bibliotheken in den Vordergrund. Gleichzeitig ist es mit *Ant* sehr gut möglich, automatisch differenzierte Releases des aktuell in der Entwicklung befindlichen Programmes als ausführbare .jar-Archive zu generieren. Dies erleichtert das Testen der Software und die Inspektion des Fortschritts der Entwicklung erheblich. Auch die Generierung der Javadoc-Dokumentation obliegt nunmehr *Ant* . Letztendlich bietet *Ant* auch Vorteile bei der Portierung der Quellen auf andere Betriebssysteme und beschränkt die Umwandlungszeit auf das Minimum, indem nur die wirklich notwendigen Generierungsschritte durchgeführt werden.

3.3 Design

3.3.1 Paketstruktur

Die für *Visual Emugen* entwickelten Java-Klassen sind in eine baumartige Paketstruktur aufgeteilt. Auf der obersten Ebene wird zwischen allgemein für Compilerbauwerkzeuge brauchbaren Programmteilen und spezifisch auf *Visual Emugen* ausgerichteten Programmteilen unterschieden. Die allgemeinen Klassen können auch herausgetrennt und einzeln zur Programmentwicklung verwendet werden.

Im allgemeinen Paket `compilerbau` werden im Unterpaket `gui` Oberflächentoolkits zur Verfügung gestellt, die folgende Dienste bereitstellen:

- Splashscreens

- allgemeines Syntaxhighlighting
- allgemeine Syntaxergänzung

Zudem werden noch einige Routinen für das "Backend" durch das Paket `util` angeboten, so zum Beispiel:

- eine Klasse zum compilieren und zusammenpacken von .java Dateien
- eine Ersatzklasse für System.out zum Abfangen der Ausgabe von Compilern
- eine Klasse zum Aufbau eines Syntaxergänzungsbaums

Das zweite große Paket `emugen` enthält die Klassen mit den Routinen, die spezifisch auf *Emugen* ausgelegt sind. So stellt zum Beispiel das Unterpaket `anttask` eine Klasse zur Verfügung, mit der Emugen sehr gut in *Ant* integriert werden kann. Das Unterpaket `visitor` enthält visitoren, die mit dem im ursprünglichen *Emugen* mitgelieferten Parser zusammenarbeiten.

Schliesslich folgt noch das Unterpaket `visual`, welches genau die Klassen enthält, die für *Visual Emugen* verwendet werden sollen. Dieses Paket untergliedert sich in drei weitere Sparten, die verantwortlich sind für

- die Verwaltung des mit Hilfe von *Visual Emugen* bearbeiteten Dokuments
- die Ansteuerung von *Grace* zur grafischen Darstellung des Dokuments
- die Darstellung der Benutzerinteraktion mittels Dialogen

Die interne Struktur des `grace` Pakets ist an die Aufteilung bei *Grace* selbst angelehnt. Die einzige Inkonsistenz die dabei in Kauf genommen wurde sind die beiden Unterpakete `e` und `n` im Paket `model`. Eigentlich sollten diese Pakete `edges` und `nodes` heißen, mussten aber auf Grund der Problematik, die auf Seite 30 im Absatz 4.2.2 geschildert wird, in `e` und `n` umbenannt werden.

Weiterhin werden noch zwei externe Bibliotheken verwendet. Dabei handelt es sich um *Emugen* , für dessen Funktionalität *Visual Emugen* schliesslich die Oberfläche darstellt, sowie um die Laufzeitbibliothek von *Grace* , deren sich die von *Grace* erzeugten Klassen bedienen.

3.3.2 Interoperation der Pakete

Bei der Implementierung von *Visual Emugen* wurde darauf geachtet, die Schnittstellen möglichst klar zu definieren und auf wenige Klassen zu begrenzen, um die Übersichtlichkeit und Nachvollziehbarkeit nicht zu gefährden. Durch die klaren Schnittstellen wird eine leichtere Nachahmung des Programmes und Verwendung der bereitgestellten Bibliotheken maßgeblich vereinfacht.

Das Paket `compilerbau` ist in sich abgeschlossen, und könnte unabhängig von *Visual Emugen* verwendet werden, um Oberflächen für Programmiersprachen zu schreiben. Das Paket

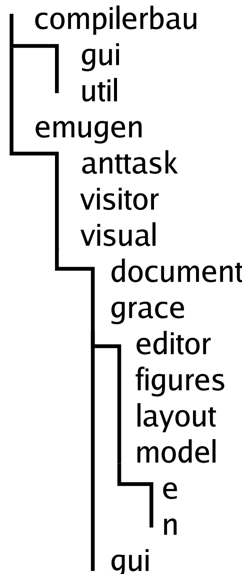

Abbildung 1: Paketstruktur in VisualEmugen

`emugen.visual` setzt direkt auf dieses Paket auf, indem es die entsprechenden Klassen verwendet, oder eigene Klassen von ihnen ableitet.

Dieses Paket `emugen.visual` übernimmt in *Visual Emugen* eine zentrale Rolle. Die Funktionalität von *Emugen* wird dabei von *Visual Emugen* über wenige Aufrufe realisiert, der bereits vorhandene Parser und der dabei enstehende abstrakte Syntaxbaum wird einfach für *Visual Emugen* weiterverwendet. Eigene Visitoren erlauben ein spezifisches Abarbeiten des entsprechenden Baums. Zur Darstellung der Syntaxelemente auf dem Bildschirm mit *Grace* muss von diesen Visitoren eine eigene Datenstruktur aufgebaut werden.

Für *Grace* wurde ein eigenes Unterpaket, `emugen.visual.grace`, geschaffen. Die Struktur dieses Pakets ist an der Struktur der Bibliothek *Grace* angelehnt, um ein einheitliches Muster zu gewährleisten.

Weiterhin existiert noch eine Verbindung zwischen *Ant* und `emugen.anttask`, da *Visual Emugen* zusätzlich noch einen Anttask beinhaltet, mit dem *Emugen* komfortabel in *Ant* -Skripte eingebunden werden kann.

3.3.3 Generierung

Zur Generierung des fertigen Programmpakets sind mehrere aufeinander aufbauende Schritte notwendig, die alle von *Ant* durchgeführt werden. Die ersten drei Schritte generieren die notwendigen Javaklassen. Diese werden mit den manuell erstellten Javaklassen kombiniert und einem Umwandlungsprozess unterzogen. Als Resultat entstehen zuerst die Bytecodedateien, die dann schliesslich zu einem verteilbaren .jar Archiv zusammengefasst werden. Bei Bedarf

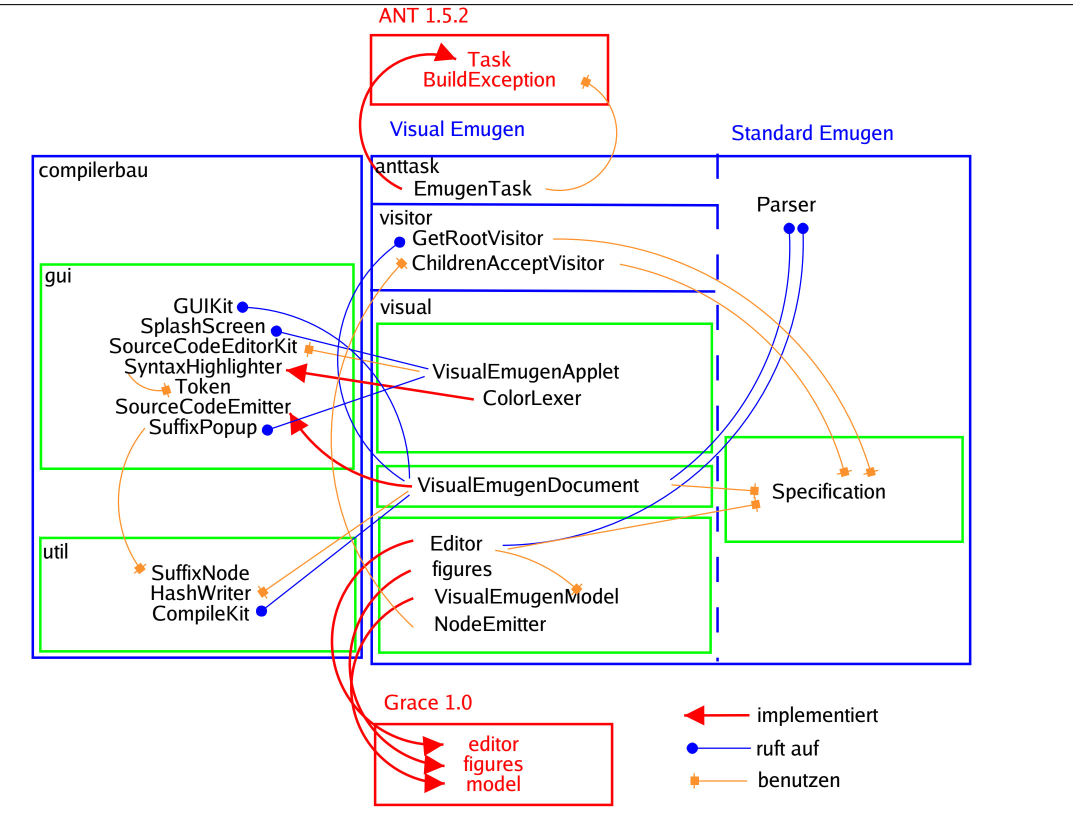

Abbildung 2: Stark vergröberte Übersicht über die Paketschnittstellen

werden dann alle notwendigen Javaarchive zusammengefasst und signiert, damit diese als im Browser ausführbaren Java-Applets funktionieren können.

3.4 Implementierung

3.4.1 Der Rahmen

Der Erzeugungsvorgang von *Visual Emugen* wird sehr stark durch die Fähigkeiten von *Ant* unterstützt. Dadurch konnte eine sehr flexible Verzeichnisstruktur (siehe Abbildung 4 auf Seite 14) als Arbeitsumgebung geschaffen werden.

In dieser Umgebung fällt es sehr leicht, Erweiterungen und Anpassungen am Code vorzunehmen. Die verschiedenen Stufen der Codegenerierung sind sauber voneinander getrennt - eigener Quellcode liegt nicht zusammen mit dem generierten Code in einem Verzeichnis, sondern wird mit den anderen Zusatzkompenten wie Bildern, etc. später erst hinzugefügt. Zusatzbibliotheken können mit minimalem Aufwand in die Umgebung eingepasst werden, in dem man die zugehörigen `.jar`-Archive in das `lib` Verzeichnis kopiert. *Ant* produziert zudem (wie in Abbildung 3 auf Seite 13 gezeigt) auf Anfrage jederzeit einen kompletten Schnapp-

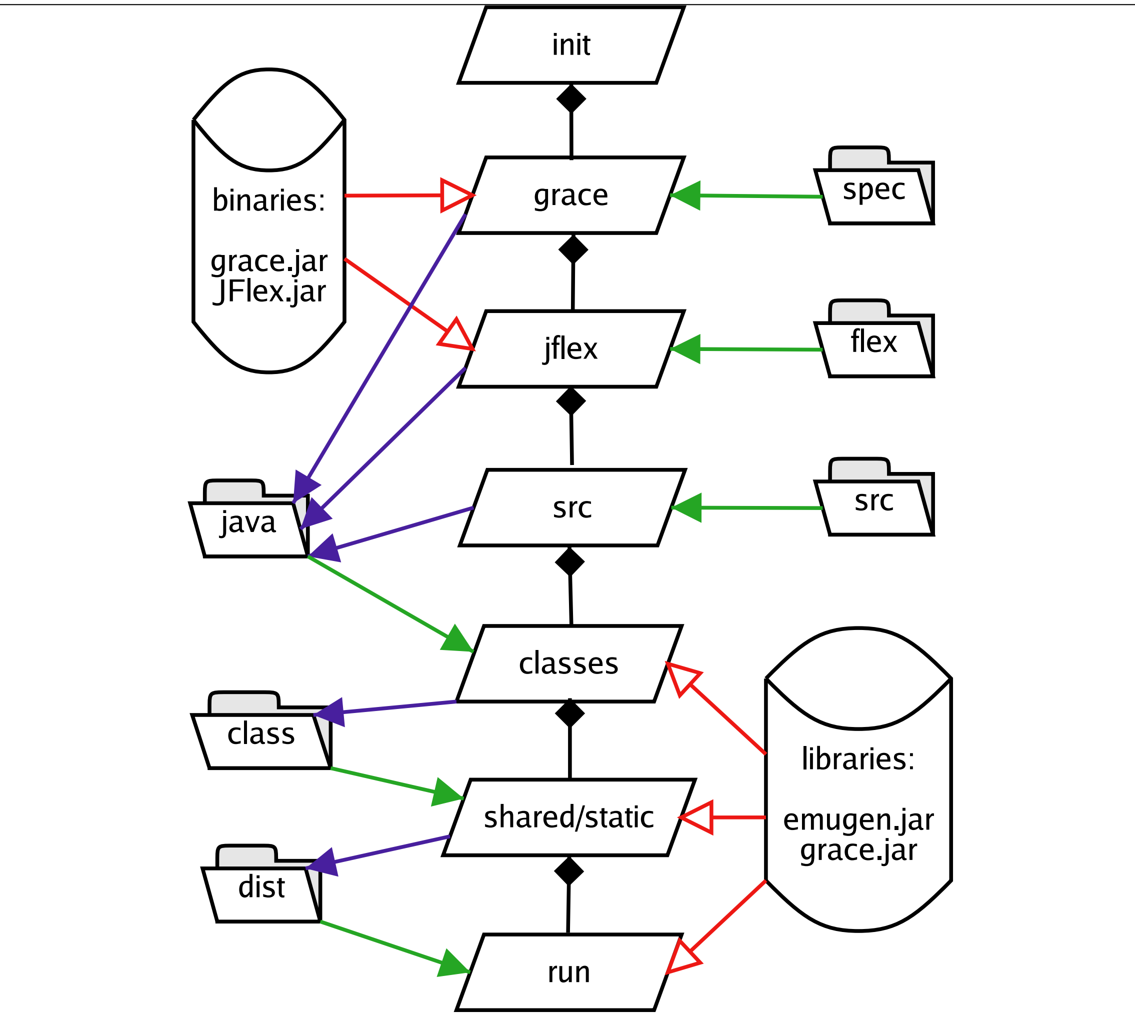

Abbildung 3: Umwandlungsvorgang mit ANT

schuss des aktuellen Codes als fertiges Release, was durch Eingabe von `ant web` erreicht werden kann. Diesen Schnappschuss findet man dann im `dist` Verzeichnis unter dem Namen `VisualEmugen-static-DATUM.jar`. Diese kann jederzeit durch Aufruf von Java mit der Zeile `java -jar VisualEmugen-static-DATUM.jar` ausgeführt werden.

3.4.2 Die Hauptanwendung

`emugen.visual.gui.VisualEmugenApplet` ist die Hauptanwendung von *Visual Emugen* und wird standardmäßig als erste ausgeführt. Diese Klasse ist darauf ausgelegt, sowohl als Standalone-Anwendung als auch als Applet im Browser ausgeführt zu werden. Diese Flexibilität kann dadurch erreicht werden, dass die ganze Klasse von `javax.swing.JApplet` an-

Abbildung 4: Verzeichnisstruktur mit Erläuterungen

statt von `javax.swing.JFrame` abgeleitet und die Main-Methode folgendermassen gestaltet wird:

```
public static void main(String[] args){
    final int width = 800;
    final int height= 600;
    // trying to be cute on Windows boxes
    try{
        UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
        );
    }catch(UnsupportedLookAndFeelException ulafe){
    }catch(ClassNotFoundException cnfe){
    }catch(InstantiationException ie){
    }catch(IllegalAccessException iae){
    }
    JApplet applet = new VisualEmugenApplet();
    JFrame frame = new JFrame("Visual Emugen "+version);
    SplashScreen splash = new SplashScreen(new ImageIcon(
       VisualEmugenApplet.class.getResource("/images/visualemugen.gif"))
       ,frame,5,version);
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20      frame.getContentPane().add(applet);
        frame.setSize(width,height);
        applet.init();
        applet.start();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
25      frame.setLocation(screenSize.width/2 - (width/2),
                          screenSize.height/2 - (height/2));
        frame.setVisible(true);
    }
```

Man sieht sehr schnell, dass in der Hauptroutine nur ein "Rahmen" (`JFrame`) für das Applet aufgebaut wird. Die weitere Ausführung dieses Programmes ist also die selbe wie die eines Applets. Es empfiehlt sich daher, jeder Anwendung auf diese Art und Weise zwei Anwendungsmöglichkeiten (im Web und lokal) zu geben.

Die Rolle der Hauptanwendung beschränkt sich in diesem Fall sowieso nur darauf, eine entsprechende grafische Hülle für die eigentliche Programmlogik zu bieten. So werden in dieser Klasse hauptsächlich Menüs aufgebaut und die entsprechenden Benutzeraktionen (Actions) auf die jeweiligen Behandlungsroutinen gelenkt.

Die eigentlichen Aufgaben erledigen dann die in dieser Hülle enthaltenen Klassen wie zum Beispiel das `VisualEmugenDocument`, das für die Verwaltung des eingetippten *Emugen* -Quellcodes verantwortlich ist. Weiterhin existieren ausserdem Klassen, die die Darstellung des Dokuments übernehmen, wie zum Beispiel der `Editor`, oder Klassen zur Benutzerinteraktion wie `SuffixPopup`, die ebenfalls eigenverantwortlich ihre Aufgaben erledigen.

3.4.3 Das zentrale Dokument

Visual Emugen ist nach dem MVC (Model-View-Controller) Prinzip aufgebaut. Dieses Konzept sieht vor, dass die Aufgaben, die in einem Programm anfallen nach ihren Funktionen getrennt in drei Kategorien unterteilt werden. Die "Model"-Komponente, also die Routinen für das Datenmodell stellt dabei `emugen.visual.document.VisualEmugenDocument`, die zentrale Dokumentklasse, bereit.

Dieses Zentraldokument muss also alle Aspekte die das Datenmodell betreffen behandeln. Es gibt allen anderen Komponenten Auskunft über Tatsachen, die den vom Benutzer editierten *Emugen* -Quellcode betreffen. Dabei handelt es sich zum Beispiel über den komplett eingegebenen Text, oder die Position von Syntaxfehlern im Quelltext. Dies kann dadurch erreicht werden, dass der bestehende *Emugen* -Parser auf den vorhandenen Quelltext angewendet wird, und die dabei ausgegebenen Fehler durch *Visual Emugen* als Syntax-Fehler registriert werden. Dies macht das Abfangen der Fehlerausgabe notwendig, die durch *Emugen* leider auf dem Standard-Ausgabestrom erfolgt. Das Abfangen ist nur durch einen Trick möglich:

```
PrintStream out = System.out;
HashWriter errors = new HashWriter();
System.setOut(errors); // let's collect the emugen errors in a Hashtable
```

Alle Routinen, die nun Ausgaben auf den Standardausgabestrom leiten, schreiben nun in den extra dafür geschriebenen `compilerbau.HashWriter`, der die Fehlermeldungen abfängt. Da *Emugen* seine Fehlermeldungen nach einem festen Schema ausgibt, kann man diese über Reguläre Ausdrücke “erkennen” und die Fehlerart, die Zeilennummer in der der Fehler auftrat und die Quelltextpassage extrahieren. Diese werden dann nach Zeilennummern sortiert in einer Hashtabelle abgelegt. Nun muss nur noch der Ausgabestrom restauriert werden:

```
System.setOut(out);
setErrors(errors);
```

Auch werden die gängigen Aktionen, die mit dem Quelltext in Verbindung stehen von der Dokumentklasse bearbeitet, wie zum Beispiel das Speichern und Laden von Dateien, sowie das Umwandeln des Quellcodes. Hierfür bedient sich *Visual Emugen* der Dienste von *Emugen* selbst, durch den Aufruf:

```
emugen.GlobalOptions.overwrite=true;
emugen.GlobalOptions.GEN_FORM_ONLY = true;
emugen.GlobalOptions.GEN_TABLE = true;
emugen.GlobalOptions.GENERATION_DIRECTORY = srcDir;
5 emugen.Main.generate(emufile.getPath());
```

Nun können aus den generierten `.java`-Quellcodes zuerst die gewünschten `.class`-Dateien generiert werden, die dann in ein ausführbares `.jar`-Paket zusammengepackt werden. Da die letzten beiden Vorgänge allgemeingültiger sind, stützt sich die Dokumentklasse auf die Dienste von Klassen des Compilerbau-Pakets ab, das im folgenden Abschnitt genauer beschrieben wird.

3.4.4 Der Umwandlungsvorgang

Die Umwandlung von *Emugen* -Quellcode in fertig verwendbare Java-Programme bedient sich unter anderem dem offiziellen Java-Compiler von SUN Microsystems. Diesen auch aus einem Java-Programm heraus anzusprechen ist weniger einfach. Java bietet zwar die Möglichkeit, über die exec Methode der `Runtime` Klasse Programme auszuführen, jedoch erfolgen diese Aufrufe immer plattformspezifisch. So ergeben sich schon Unterschiede beim Aufruf des Compilers unter Linux und Windows:

```
/usr/local/java/bin/javac /home/petter/Klasse1.java
vs.
C:\Programme\Java\bin\javac.exe D:\src\Klasse1.java
```

Diese Problematik kann nur dadurch umgangen werden, dass man die Tatsache ausnutzt, dass Suns Java-Compiler selbst ein Java-Programm ist, und daher als `.class`-Datei in einem `.jar`-Archiv vertrieben wird. Diese Tatsache wird von SUN nicht besonders ausführlich dokumentiert, da das entsprechende Archiv von Java Version zu Java Version variiert, und ist daher weitgehend unbekannt. Die Klasse `CompileKit` des Compilerbau-Paket von *Visual Emugen*

nutzt diese Tatsache aus, und lädt die benötigte Compiler-Klasse über den Java-Reflection Mechanismus, in welchem der möglichen Archive sie auch immer sein sollte:

```
public static int javac(String classpath, File destinationDir,
                        File srcDir, String javaFile)
{
  // dirty hack to call the java compiler by method...
  java.net.URL[] urls = new java.net.URL[2];
  File compilerJar = new File(System.getProperty("java.home")+
                          System.getProperty("file.separator")+
                          ".."+
                          System.getProperty("file.separator")+
                          "lib"+
                          System.getProperty("file.separator")+
                          "tools.jar");
  try{
     urls[0]=compilerJar.toURL();
  }catch (java.net.MalformedURLException mue){ };// can't occur
  compilerJar = new File(System.getProperty("java.home")+
                     System.getProperty("file.separator")+
                     "lib"+
                     System.getProperty("file.separator")+
                     "tools.jar");
  try{
     urls[1]=compilerJar.toURL();
  }catch(java.net.MalformedURLException mue){ }; // can't occur
  java.net.URLClassLoader ucl = new java.net.URLClassLoader(urls);

  Class clazz = null;
  try{
     clazz = ucl.loadClass("com.sun.tools.javac.Main");
  }catch (ClassNotFoundException cnfe) { }; // won't occur either...
  java.lang.reflect.Method compile = null;
  try{
     compile = clazz.getMethod("compile",new Class[]{ String[].class });
  }catch(NoSuchMethodException nsme){ }; // won't occur either...

  // preparing the parameters
  final String [] parameter = new String[7];
  parameter[0]= "-classpath";
  parameter[1] = classpath;
  parameter[2]= "-d";
  parameter[3]= destinationDir.getAbsolutePath();
  parameter[4]= "-sourcepath";
  parameter[5]= srcDir.getAbsolutePath();
  parameter[6]= javaFile;
  int result = 1;
  try{
     result = ((Integer)compile.invoke(clazz.newInstance(),
                 new Object[] {parameter})).intValue();
```

```
    }catch(Exception e) { }; // shouldn't occur ...
    return result;
50 }
```

Auch das Verpacken der generierten `.class`-Dateien in `.jar`-Archive wird von der Klasse `CompileKit` geleistet, und muss nur noch von der Dokumentklasse angestartet werden.

3.4.5 Die Grafikengine

Die grafische Ausgabe des Datenmodells des vom Benutzer neu verfassten *Emugen* -Quellcodes wird wie in *Visual Classgen* über die Bibliothek *Grace* realisiert. Um die Dienste von Grace nutzen zu können, muss eine Grace-Editorspezifikation vorliegen, die im Verzeichnis `spec` (siehe Abbildung 4 auf Seite 14) abgelegt ist. In dieser Spezifikation wird auf einzelne Klassen zur Repräsentation des Datenmodells als Knotenbaum Bezug genommen. Die so referenzierten Klassen müssen natürlich auch bereitgestellt werden. Im Falle von *Visual Emugen* geschieht die Bereitstellung des Modells in `emugen.visual.grace.model` sowie die Bereitstellung der sichtbaren Formen in `emugen.visual.grace.figures`. Die Klassen des Pakets `figures` repräsentieren die *Grace* - eigenen grafischen Bausteine, während die Klassen aus dem `model` Paket die Knoten- und Kantentypen darstellen. (siehe Abbildung 5 auf Seite 19). Die Verknüpfung der beiden Komponenten wird in der Grace-Spezifikation vorgenommen. Der Graph, der aus einer *Emugen* -Quellcodedatei enstehen soll, wird durch den NodeEmitter generiert. Er erstellt mithilfe seiner Spezialvisitoren die richtigen `model`-Klassen.

Zur Anordnung des Baums auf der Zeichenfläche übernimmt *Visual Emugen* den Algorithmus von *Visual Classgen* ohne Änderungen.

Theoretisch wäre an dieser Stelle alles so vorbereitet, dass *Visual Emugen* mit Hilfe von *Grace* den Kompletten Datenmodellbaum anzeigen kann, jedoch macht nun *Grace* einen Strich durch die Rechnung: Einige Bugs im generierten Editor verhindern die komplette Darstellung des ganzen Baumes, da der sichtbare Bereich der von *Grace* generierten Editor-Komponente zu klein ist. Bei der Entwicklung von *Visual Classgen* stiess man auf genau dasselbe Problem, was dadurch gelöst wurde, dass *Visual Classgen* eine eigene Editor-Klasse benutzt, die von der generierten Klasse abgeleitet ist. In dieser neuen abgeleiteten Klasse wird die Berechnung des sichtbaren Bereichs korrigiert, so dass der komplette Baum angezeigt wird. *Visual Emugen* geht dabei einen ähnlichen Weg, aber verwendet eine überarbeitete Version der korrigierten Berechnung.

3.4.6 Syntaxhighlighting

Das Syntaxhighlighting in *Visual Emugen* entsteht durch das Zusammenspiel mehrerer Komponenten. Beteiligt daran sind:

1. ein JFlex-Scanner
2. eine Klasse mit Farbdaten

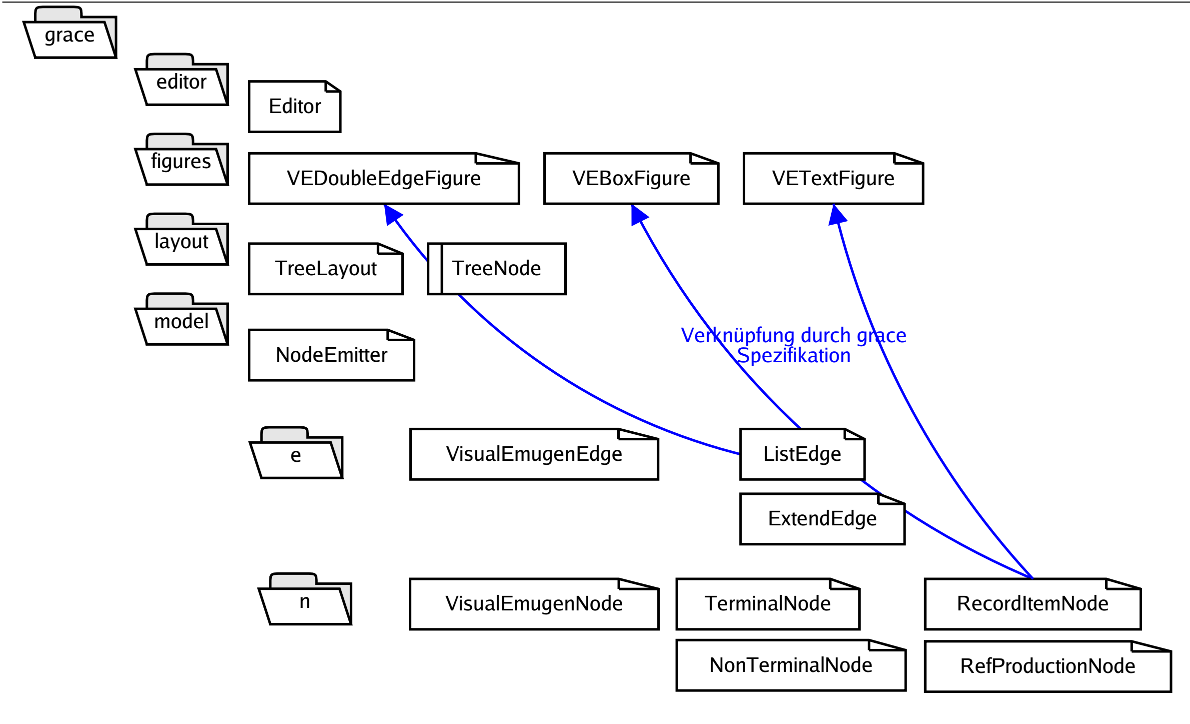

Abbildung 5: Schema des Grace-Unterpaktes

3. die Klasse, die die Codeansicht darstellt
4. die Dokumentklasse

Die Dokumentklasse fungiert beim Syntaxhighlighting in genau derselben Aufgabe wie sonst auch: Sie stellt die grundlegenden Daten, also den *Emugen* -Code selbst den anderen Klassen zur Verfügung.

Die Klasse `compilerbau.gui.SourceCodeView` nimmt diesen Quelltext entgegen, um ihn im Fenster darzustellen. Um die Informationen, mit welcher Farbkodierung die Darstellung vorgenommen werden soll, zu erhalten, übergibt die `SourceCodeView`-Klasse den Quellcode an einen Scanner weiter, der daraus dann Folgen von Farbdaten-Klassen erzeugt.

Die Integration des Scanners erfolgt so, dass speziell für den *Emugen* -Sourcecode ein JFlex-Scanner implementiert wird, der die entsprechend gewünschten Farben für den Sourcecode generieren muss. Diese Farbdaten gibt der Scanner wieder an die `SourceCodeView`-Klasse zurück.

Die `SourceCodeView`-Klasse ist realisiert als Kindklasse einer `PlainView`, die über die ererbten Methoden `drawXXXXText` für die Ausgabe des Textes verantwortlich ist. An dieser Stelle werden also die Farbdaten aus der Klasse `compilerbau.gui.Token` ausgewertet, und in die entsprechenden Färbungen umgesetzt.

Der ganze Vorgang wird in Abbildung 6 auf Seite 20 deutlich.

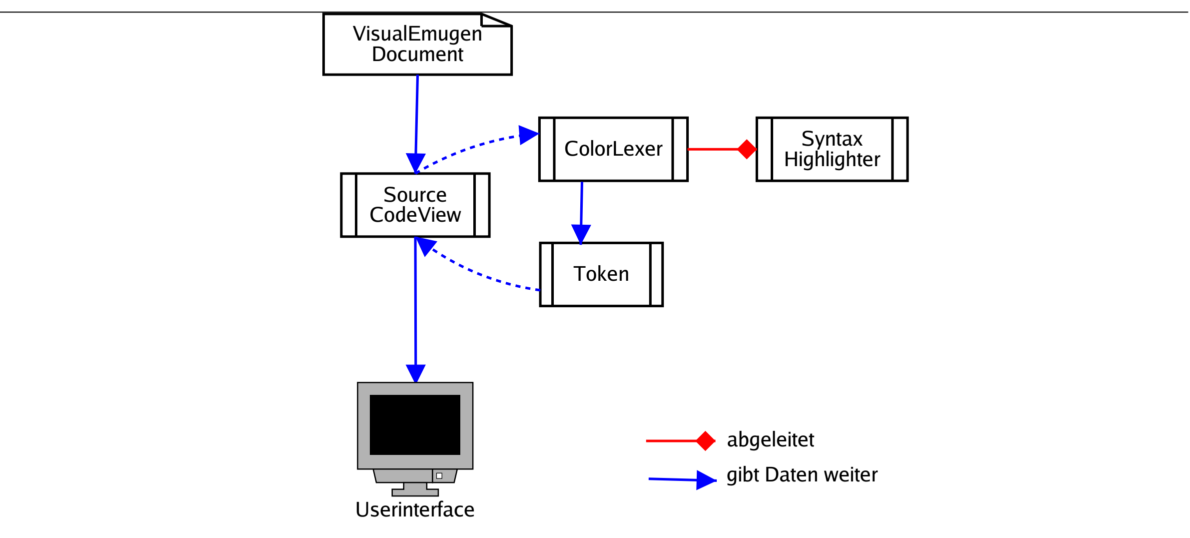

Abbildung 6: Syntaxhighlighting im Detail

An dieser Stelle ergibt sich eine kleine Einschränkung: Die Methode, das Syntaxhighlighting durch eine abgeleitete `PlainView`-Klasse ausführen zu lassen wurde direkt aus *Visual Classgen* übernommen. Leider ist das Syntaxhighlighting auch dort so realisiert, dass der Scanner nicht mit dem kompletten Quellcode gefüttert wird, sondern nur die aktuell neu darzustellende Codezeile bekommt. Das führt dazu, dass der Scanner nicht mehr alle Klammerausdrücke als solche erkennen kann, und auch die Scanner-Zustände so gut wie unbrauchbar werden. Es sind also nur sehr einfache Einfärbungen mit diesem Konzept möglich...

3.4.7 Syntaxergänzung

Die Syntaxergänzung in *Visual Emugen* ist realisiert über die Verwendung von zwei Klassen: `compilerbau.util.SuffixNode` und `compilerbau.gui.SuffixPopup`. Mit der Klasse `SuffixNode` kann eine Baumstruktur aufgebaut werden, mit deren Hilfe berechnet werden kann, ob und wie ein eingegebener Suffix nach den vorhergegangenen Worten ergänzt werden kann. Die Klasse `SuffixPopup` ist für die Integration dieser Datenstruktur in ein beliebiges `JEditorPane` zuständig. Abbildung 7 auf Seite 21 veranschaulicht einen Beispielsuffixbaum. Der Algorithmus zum Aufbau dieses Baums startet beim ersten Wort und schreibt dieses Komplett als Knoten in den Baum. Dann macht er mit dem nächsten Wort weiter. Falls dieses Wort nun genauso beginnt wie ein bereits existierender Pfad im Baum, so wird der maximale Präfix der beiden Wörter bestimmt, und der existierende Knoten an dieser Stelle in zwei zusammenhängende Knoten aufgetrennt. Nun kann der Suffix des neuen Wortes angefügt werden. Um auch Präfixe von Worten als Worte selbst markieren zu können fügt man noch ein Wortendesymbol an den entsprechenden Stellen ein. Diese Funktionalität liefert die Methode `createTree()` der Klasse `SuffixNode`.

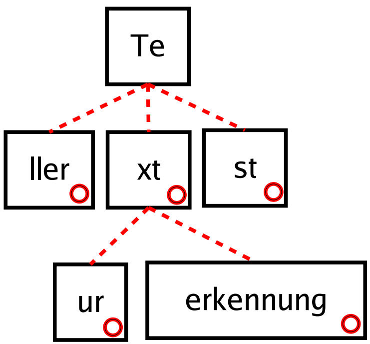

Abbildung 7: Aufbau eines Suffix Baum

Sollen nun alle möglichen Endungen eines neu übergebenen Präfixes berechnet werden, so kann einfach von der Wurzel des Baumes solang in Richtung des präsentierten Präfixes gegangen werden, bis entweder ein Mismatch auftritt (dann gibt es keine gültige Ergänzung), ein Blatt erreicht wird (dann ist der Präfix selbst die einzig gültige Ergänzung). Ist keines von beiden der Fall, so können die gültigen Ergänzungen durch eine Breitensuche nach Wortendesymbolen gefunden werden. Abbildung 8 auf Seite 21 veranschaulicht die Suche nach gültigen Ergänzungen des Präfix "Tex" im Beispielbaum aus Abbildung 7. Das Ergebnis sind drei Ergänzungen: "Text", "Textur" und "Texterkennung".

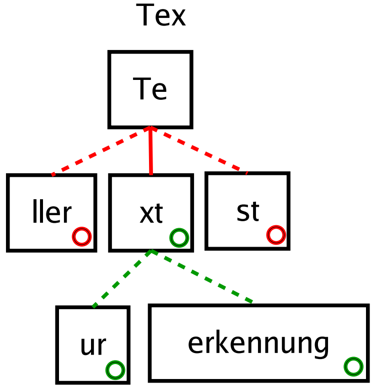

Abbildung 8: Syntaxvervollständigung mit einem Suffix Baum

3.4.8 Fehlerkennzeichnung

Die Fehlerkennzeichnung in *Visual Emugen* ist nicht besonders komplex realisiert - *Emugen* tut hier die ganze Arbeit, und *Visual Emugen* übernimmt nur die Markierung der Fehler im Quellcode. Spannend wird dieser Aspekt nur dadurch, dass *Emugen* keine besonders entgegenkommende Schnittstelle für den Aufruf aus anderen Programmen heraus besitzt. Ein Aufruf des *Emugen* -Parsers resultiert bei aufgetretenen Fehlern in Fehlerausgaben auf dem Standard Ausgabestrom. In *Visual Emugen* ist daher ein entsprechender Abfangmechanismus für die Standardausgabe vorgesehen. Zudem müssen die Fehlermeldungen, die *Emugen* an den Benutzer weitergibt, entsprechend nach Zeilen und Spaltennummern geparst werden.

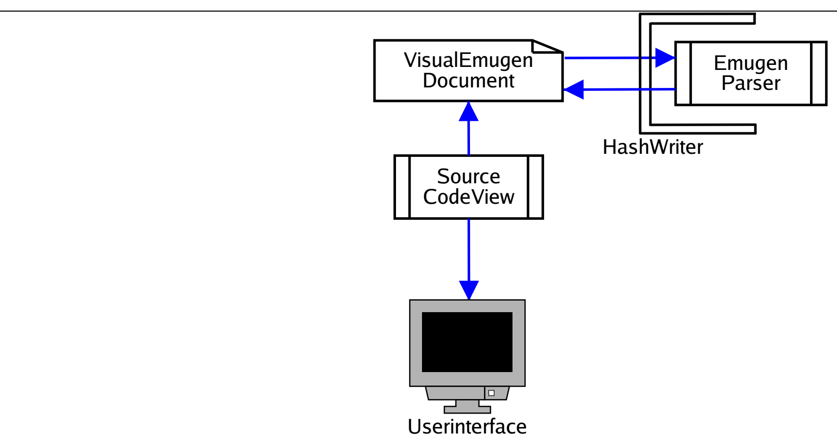

Abbildung 9: Ablauf zur Fehlerkennzeichnung

3.4.9 Antintegration

Im Rahmen der Erstellung von *Visual Emugen* trat zu Tage, dass die Integration des Kommandozeilenwerkzeugs *Emugen* nicht besonders einfach war. Als Folge davon entstand innerhalb von *Visual Emugen* eine Klasse, die die Integration von *Emugen* in *Ant* -Abläufe ermöglicht.

Realisiert wird diese Integration durch eine einzige Klasse aus dem Paket `emugen.anttask`, nämlich `EmugenTask`. Diese Klasse dient für *Ant* als Einstiegspunkt in das Geflecht von Parsern und Datenstrukturen, was *Emugen* zur Verfügung stellt. Ihre Methode `execute` imitiert dabei die Abläufe, die im Orginal *Emugen* in der `main` Methode auftreten.

3.4.10 Emugen-Weiterleitung

Eine weitere Eigenschaft von *Visual Emugen* besteht in der Kommandozeilenschnittstelle. Wenn *Visual Emugen* ohne jeden Parameter, oder mit dem Parameter `-gui` aufgerufen wird, so wird die gewohnte grafische Oberfläche gestartet werden. Sobald jedoch ein anderer Parameter verwendet wird, so wird der komplette Parameterstring an *Emugen* weitergereicht,

damit dieses aufgerufen werden kann. Auf diese Art kann *Visual Emugen* komplett als Ersatz für *Emugen* verwendet werden.

4 Dokumentation

Der folgende Part soll hauptsächlich als Anleitung dienen, wie *Visual Emugen* hergenommen werden kann, dabei beschreibt der erste Teil die Verwendung der grafischen Benutzerschnittstelle, und der zweite Teil die Benutzung der Bibliotheken, die im Zusammenhang mit der Programmierung von *Visual Emugen* entstanden sind.

4.1 Visual Emugen Benutzung

Es existieren drei Möglichkeiten, *Visual Emugen* zu benutzen. Zum Ersten kann *Visual Emugen* direkt als *Emugen* - Ersatz, also als Kommandozeilenwerkzeug, hergenommen werden. Weiterhin bietet *Visual Emugen* eine Schnittstelle um in einen Umwandlungsvorgang mit dem Werkzeug *Ant* zu unterstützen. Schliesslich kann man *Visual Emugen* auch direkt ausführen und mit Hilfe der grafischen Schnittstelle *Emugen* -Spezifikationen entwickeln.

4.1.1 Aufruf von der Kommandozeile

Die Möglichkeit von *Emugen* , das Programm von der Kommandozeile aus aufzurufen wurde bei *Visual Emugen* direkt übernommen. Intern wird der Aufruf meistens sowieso an *Emugen* weitergereicht. Das Grundgerüst für den Aufruf läuft folgendermassen ab:

```
java -jar VisualEmugen-static-VERSION.jar -f -fo emugenfile.emu
```

Bei Bedarf können *Visual Emugen* , und damit *Emugen* weitere Parameter angehängt werden. Diese Parameter kann man durch Eingabe von

```
java -jar VisualEmugen-static-VERSION.jar -help
```

sehr leicht selbst herausfinden.

4.1.2 Aufruf aus Ant

Dank *Visual Emugen* können *Emugen* -Dateien jetzt auch in einen *Ant* -Umwandlungsprozess eingebunden werden. Ein Beispiel finden Sie hier:

```
  <!-- a very minimal build file -->
  <project name="EmugenProjekt">

    <!-- defining a path to the VisualEmugen jar file -->
5   <path id="binaries">
      <pathelement location="bin/VisualEmugen-static-VERSION.jar" />
    </path>
```

```
    <!-- declaring the additional task "emugen" -->
10  <taskdef name="emugen"
             classname="emugen.anttask.EmugenTask"
             classrefpath="binaries"
     />

15  <!-- calling the new task from a target -->
    <target name="generate">
      <emugen file="emuinput.emu"
              destdir="generated"
              overwrite="true"
20            formsonly="true"
       />
    </target>
  </project>
```

Das gewünschte Ziel sieht man hier im dritten Block (Zeile 15). Der Benutzer möchte in seinen einzelnen Targets den Task “emugen” hernehmen können. Weil “emugen” nicht Bestandteil der Standardwerkzeuge von *Ant* ist, muss dieses neue Task *Ant* bekannt gegeben werden. Dies passiert im zweiten Block (Zeile 10). Damit *Ant* bescheid weiss, wo es die Klassen suchen muss, auf die in der Taskdefinition Bezug genommen wird, wird hier der Klassenpfad aus Zeile 5 referenziert.

4.1.3 Grafische Oberfläche

Der Hauptvorteil von *Visual Emugen* ist sicherlich seine grafische Oberfläche. Sie erlaubt komfortables Entwickeln von *Emugen* -Quellcodes, bei einfacher Generierung der Ergebnisse. Gestartet werden kann die Oberfläche auf jedem Betriebssystem, wo das Java SDK 1.4.2 installiert ist, durch die Eingabe von

```
java -jar VisualEmugen-static-VERSION.jar
```

Auf Systemen mit Microsoft Windows sollte auch ein Doppelklick auf die `.jar`-Datei die Anwendung starten. Daraufhin sollte sich *Visual Emugen* wie in Abbildung 10 auf Seite 25 präsentieren. Der Arbeitsbereich in *Visual Emugen* gliedert sich in zwei Bereiche, dem Codefenster links und der Baumanzeige rechts. Der Benutzer kann nun anfangen, seinen Quellcode in das Codefenster einzugeben. Gültiger Quellcode wird im Codefenster (wie in Abbildung 11 auf Seite 26) farblich sinnvoll hervorgehoben.

Die Baumanzeige (siehe Abbildung 12 auf Seite 26) verändert sich dabei, um die Datenstruktur des Quellcodes geeignet abzubilden. Wenn Fehler im Quellcode auftreten, die der *Emugen* -Parser erkennt, so wird die Baumansicht verschwinden, und das Fehlerhafte Symbol im Codefenster durch eine rote Unterstreichung (siehe Abbildung 11 auf Seite 26) gekennzeichnet.

Als letzte Benutzerunterstützung existiert noch die Syntaxergänzung (siehe Abbildung 11 auf Seite 26). *Visual Emugen* führt intern Buch über alle bereits eingegebenen Wörter sowie

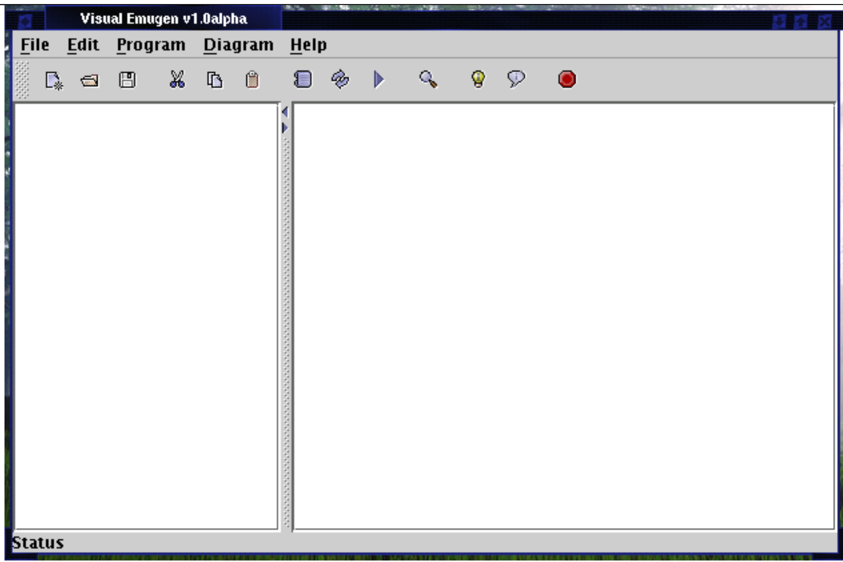

Abbildung 10: Screenshot nach dem Start

über die *Emugen* und *Java* Schlüsselwörter. Fängt nun der Benutzer an, ein Wort einzutippen, so versucht *Visual Emugen* die aktuelle Eingabe durch sinnvolle Endungen zu ergänzen, und bietet diese Auswahl in einem Dropdown-Menü an. Ist der gewünschte Begriff dabei, so kann der Benutzer durch Betätigen der `Pfeil-ab` Taste in dem Menü die richtige Ergänzung anwählen, und mit `Enter` bestätigen. Sollte er sich doch wieder Erwarten anders entschließen, so kann das Menü durch `Esc` abgebrochen oder durch `Strg-Leer` wieder angezeigt werden.

Sobald die Entwicklung der *Emugen* -Quellcodes abgeschlossen ist, kann der Benutzer den Menüpunkt `Program` - `Compile` anwählen. Dies führt dazu, dass *Visual Emugen* automatisch den eingegebenen Quellcode mit Hilfe von *Emugen* in `.java`-Dateien übersetzt, und diese `.java`-Dateien gleich in `.class`-Dateien umwandelt, die dann zu einem fertig ausführbaren `.jar`-Paket zusammengefasst werden. Anschliessend wird der Benutzer aufgefordert (siehe 13 auf Seite 27), einen Ort anzugeben, an dem das fertige Programm abgespeichert werden soll. *Visual Emugen* legt gleichzeitig auch die *Emugen* -Laufzeitumgebung an diesem Ort ab.

Das letzte Merkmal von *Visual Emugen* ist der Export von Grafiken, der in *Visual Emugen* ganz einfach über den Menüpunt “Diagram”/“Screenshot” abläuft.

4.1.4 Start als Applet

Visual Emugen ist so designed, dass es auch direkt aus dem Browser heraus gestartet werden kann. Dadurch kann man *Visual Emugen* sehr leicht auf Webseiten einbinden. Das einzige Problem dabei ist, dass *Visual Emugen* einen Übersetzungsmechanismus beinhaltet. Dadurch muss zum Einen das Applet Zugriff auf die Festplatte des Surfers haben. Dem kann dadurch abgeholfen werden, dass das Applet zertifiziert ausgeliefert wird, und daher die Möglichkeit

```
form layot of Vorlesungen {:
  removeAll();
  setLayout(new BorderLayout());
  add(InfoIPanel,BorderLayout.CENTER);
  add(InfoIIPanel,BorderLayout.WEST);
  add(InfoIIIPanel,BorderLayout.EAST);
:}
```

```
Anmeldung::=  String:Datum
              Student Vorlesungen
Student::=    String:Nachname
              String:Vorname
              S|
```

String
Student
Staatsbürgerschaft
Semesterwochenstunden

Abbildung 11: Syntaxhighlighting mit Syntaxfehler und Syntaxergänzung

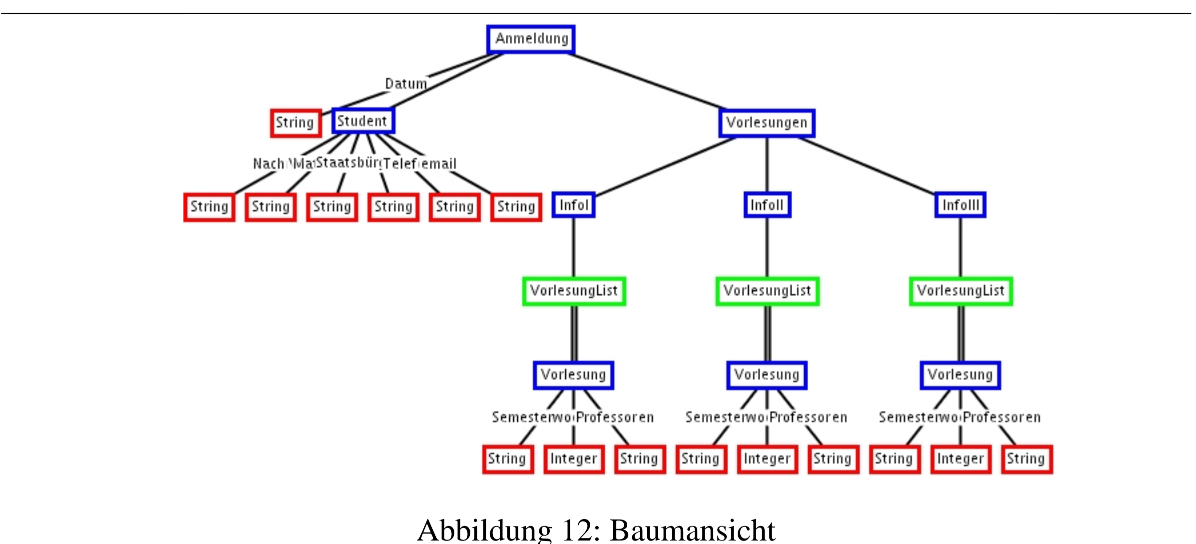

Abbildung 12: Baumansicht

erhält, nach Erlaubnis des Benutzers auf den lokalen Computer zuzugreifen. Zum Anderen muss der Benutzer nicht nur die Java 1.4.2 Runtime Engine installiert haben, sondern das komplette Java 1.4.2 SDK.

Zur Einbindung von *Visual Emugen* in die eigene Site empfiehlt sich folgender Code:

```
<applet code="emugen.visual.gui.VisualEmugenApplet.class"
        archive="VisualEmugen-static-VERSION.jar"
        width="700"
        height="500">
5  <param name="progressbar" value="true" >
  <param name="boxmessage" value="Java␣l&auml;dt...">
</applet>
```

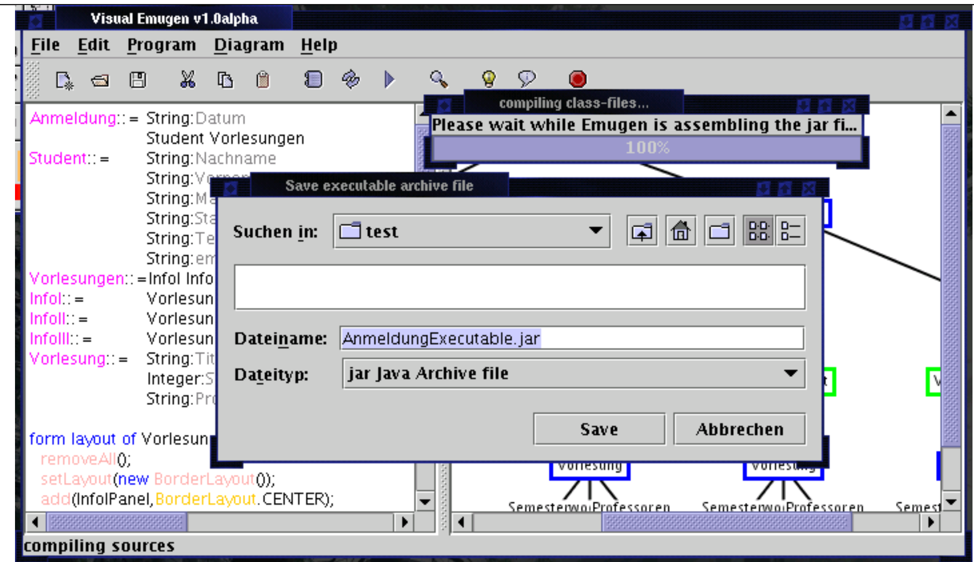

Abbildung 13: Generierung von ausführbaren Dateien

4.2 Entwicklung

Die anschliessenden Abschnitte sollen Anleitung dazu liefern, selbst Bestandteile aus *Visual Emugen* in eigene Programme einzubauen - schliesslich wurde bei der Entwicklung von *Visual Emugen* Wert auf Modularität gelegt... Die Zielgruppe für die nächsten Abschnitte ist also der Programmierer, der sein eigenes Programm mit ähnlichen Mechanismen ausstatten will, wie sie in *Visual Emugen* vorkommen.

4.2.1 Visual Emugen umwandeln

Visual Emugen kann auf zwei Arten erzeugt werden - eine Variante, in der alle nötigen Bibliotheken dynamisch dazu gebunden werden, sowie eine statisch gebundene Variante. Eine Übersicht über die beteiligten Zwischenschritte findet sich in Abbildung 3 auf Seite 13.

static

Ant wird mit dem Parameter `static` aufgerufen. Dadurch entsteht im `dist`-Verzeichnis ein großes `.jar` Archiv mit dem Namen `VisualEmugen-static-VERSION.jar`. Diese Variante wird zudem von *Ant* auch noch signiert.

Der Verwendungszweck dieser Variante ist die Verteilung von *Visual Emugen* auf andere Computer und die Ausführung im WWW über die Applet Technologie. In beiden Fällen ist es wichtig, alle Bibliotheken, die zur Ausführung notwendig sind, gleich mit im selben `.jar`-Archiv auszuliefern. Die Ausführung dieser Variante ist denkbar einfach:

```
java -jar VisualEmugen-static-VERSION.jar
```

shared

Dazu muss *Ant* mit dem Parameter `shared` aufgerufen werden. Diesmal erzeugt *Ant* mehrere `.jar`-Archive im `dist`-Verzeichnis: Es werden sowohl das eigentliche Programmarchiv `VisualEmugen-shared-VERSION.jar` als auch alle benötigten Bibliotheken erzeugt. Ausgeführt wird diese Variante mit

```
java -jar VisualEmugen-shared-VERSION.jar
```

Diesmal ist es jedoch besonders wichtig, dass alle benötigten Bibliotheken im selben Verzeichnis wie `VisualEmugen-shared-VERSION.jar` liegen. Es handelt sich dabei um `emugen.jar` und `grace.jar`.

In dieser Variante entsteht auch `compilerbau-shared-VERSION.jar`, die interne Bibliothek, die alle Werkzeuge enthält, die bei der Entwicklung von *Visual Emugen* entstanden sind, aber auch allgemein für andere grafische IDEs weiterverwendbar sind. Diese Bibliothek kann ganz einfach aus dem `dist`-Verzeichnis entnommen, und in andere Projekte übernommen werden.

4.2.2 Graphendarstellung

Die Integration von *Grace* -Klassen war eigentlich nicht Thema dieses Projekts, jedoch erwiesen sich die von *Grace* gelieferte Dokumentation als sehr karg und das Werkzeug selbst als fehlerhaft. Daher sollen hier noch zusätzlich zur eigenen Dokumentation des *Grace* Projekts einige Hinweise in eigener Sache gegeben werden:

- **Fehlerhafter Editor**

  Die von *Grace* generierte Editorklasse ist so wie sie ist nicht für Grapheditoren brauchbar. Das liegt daran, dass die Informationen über die eigene Größe, also Höhe und Breite, für das Editor-Element durch das Grace-Framework nicht korrekt erfolgt. Bei der Berechnung der benötigten Größe bezieht der von *Grace* zur Verfügung gestellte Editor nur die Graphknoten mit ein, die Kanten bleiben unberücksichtigt. Daher bietet es sich an, im eigentlichen Programm nur eine eigene, von der *Grace* -Editor-Klasse abgeleitete Klasse zu verwenden. Wichtig ist für diese Klasse die überladung der Methoden `getBoundingBox()`,`layoutGraph()` und `requestScrollRepaint()`. Letztere Übernimmt die Korrektur der ererbten Darstellung, in dem die gezeichneten Knoten an die richtige Stelle im Dialogelement verschoben werden:

```
public void requestScrollRepaint(boolean always) {
  JScrollPane scrollPane=(JScrollPane)getParent().getParent();
  Rectangle2D.Double bounds=(Rectangle2D.Double)getBoundingBox();
  Dimension viewport=scrollPane.getViewport().getExtentSize();
5 //test if the viewportsize doesn't fit the needs of our nodes
  //if this happens, move all nodes into the viewport
  //if everything is alright, do no correction to prevent an
  //irritating movement of the nodes
  boolean x=
```

```
10        ((bounds.width +Math.abs(bounds.x)+20)>viewport.getWidth())
          || (bounds.x<0);
      boolean y=
          ((bounds.height+Math.abs(bounds.y)+20)>viewport.getHeight())
          || (bounds.y<0);
15    always=always||(x&&y);
      if (always)          translateAll(-bounds.x,-bounds.y);
      else if (x) always = translateAll(-bounds.x, 0);
      else if (y) always = translateAll(0, -bounds.y);

20    //if there were any corrections necessary set the new width
      //and inform the scrollpan about changes
      if (always) {
          setPreferredSize((int)Math.ceil(bounds.width),
                           (int)Math.ceil(bounds.height));
25        scrollPane.validate();
      }
      //don't forget to request a repaint
      super.requestRepaint();
    }
```

Damit `requestScrollRepaint()` richtig arbeiten kann, muss es wissen, wie groß der zu zeichnende Bereich wirklich ist. Damit das klappt, muss eine korrekte Methode `getBoundingBox()` zur Verfügung gestellt werden:

```
    public Rectangle2D getBoundingBox() {
      Rectangle2D.Double result=new Rectangle2D.Double(0,0,0,0);

      //calculate the bounding box for all nodes
5     Iterator it = getModel().getNodes();
      while (it.hasNext())
          Rectangle2D.union(
              result,
              getJNode((Node)it.next()).getFigure().getBounds(),
10            result);

      //make sure the bounding box also surrounds the edges
      it=getModel().getEdges();
      while (it.hasNext())
15        Rectangle2D.union(
              result,
              getJEdge((Edge)it.next()).getFigure().getBounds(),
              result);

20    //do some correction because of possible numerical failures
      result.width=Math.ceil(result.getWidth())+1;
      result.height=Math.ceil(result.getHeight())+1;
      return result;
    }
```

Schliesslich muss noch die Methode `layoutGraph()` so überladen werden, dass sie nach dem eigenen Layout auch `requestScrollRepaint(true)` aufruft:

```
public void layoutGraph(){
  super.layoutGraph();
  requestScrollRepaint(true);
}
```

- **Ungünstiger Scanner für die Spezifikationsdatei** Bei der Arbeit mit *Grace* ist es nötig, eine Spezifikationsdatei zu erstellen. Das Konzept dahinter ist relativ geschickt gemacht - es ist möglich eigene Klassen zur Darstellung und internen Repräsentation von Knoten und Kanten zu verwenden. Dazu war es wohl notwendig, Import-Statements in einzelnen Bereichen der Spezifikationsdatei zuzulassen. Das wäre an sich eine großartige Sache - wenn nicht aus scheinbar unerklärlichen Gründen das Einbinden von gewissen Klassen zu unerklärlichen Fehlermeldungen beim Umwandeln der Spezifikation in die Klassensammlung auftreten würden.

  Die Lösung dieses ungewöhnlichen Phänomens ist, dass die Verwendung von *Grace* -Schlüsselwörtern wie `node` oder `edge` im Import-Statement den Scanner dazu bringt, auch das entsprechende Token dem Parser zu melden. Dies bringt natürlich den Parser im höchsten Grade durcheinander. Für den Benutzer ist dies allerdings sehr lästig, da er eigentlich wie in objektorientierten Umgebungen Klassen nach Verwendungszweck in entsprechende Pakete gruppieren möchte. Wenn nun allerdings Paketnamen wie `node` oder `edge` zu Parse-Fehlern führen, dann ist das schade. Das ist übrigens auch ein Grund dafür, warum *Visual Emugen* auch Paketnamen wie `emugen.visual.grace.model.e` führt, statt `emugen.visual.grace.model.edges`, was eigentlich naheliegend gewesen wäre. Eine Verwendung von Scanner-Zuständen im *Grace* -Scanner hätte dieses Problem umgangen...

- **Algorithmus für Baumlayout** Ein wenig Schade ist die geringe Auswahl an Standard-Layout-Algorithmen in *Grace* . So wurde bei der Entwicklung von *Visual Emugen* auf den bereits in *Visual Classgen* verwendeten Layout-Algorithmus zurückgegriffen. Dieser findet sich nun im Paket `emugen.visual.grace.layout` wieder.

4.2.3 Java-Compiler verwenden

Das `compilerbau`-Toolkit stellt unter anderem eine Methode bereit, über die ein Programmierer eine sehr einfach zu bedienende Schnittstelle zum SUN-Java-Compiler erhält. Durch Verwendung dieser Methode kann er in einer beliebigen selbstgeschriebenen Java-Klasse eine Textdatei mit Java-Code an den Übersetzer übergeben, um den daraus generierten Code sofort dynamisch in das momentan aktive Programm nachzuladen.

Ein Anwendungsbeispiel für eine Verwendung des SUN-Java-Compilers ist die Unterstützung von beliebig eingebbaren arithmetischen Ausdrücken durch den Anwender. So ist es beispielshalber auf einmal nicht mehr nötig, selbst einen Parser für arithmetische Ausdrücke mit

allen Schikanen zu entwerfen. Man übergibt den gewünschten Ausdruck einfach an den Java-Compiler mit seinen mächtigen Bibliotheken und lässt sich den gewünschten Ausdruck (zu allem Überfluss auch noch optimiert) in eine Java-Klasse umwandeln, die man anschliessend verwenden kann.

Ein weiteres denkbares Anwendungsszenario wäre die Bereitstellung einer Art "Makrosprache", über die der Benutzer das geschriebene Programm automatisieren kann....

Um in den Genuss des Compilers zu kommen ist es lediglich notwendig, das Bibliothekspaket `compilerbau` in das eigene Program mit aufzunehmen. Im Quellcode kann der Compiler dann so aufgerufen werden:

```
import compilerbau.util.CompileKit;
public class Example{
  public static void main(String[] args){
    String classpath = "lib/MyLibrary.jar";
    File     destdir = new File("classes");
    File      srcDir = new File("java");
    String  javaFile = "MyClass.java";
    CompileKit.javac(classpath,destdir,srcDir,javaFile);
  }
}
```

Zur Laufzeit muss nun nur noch darauf geachtet werden, dass der SUN-Java-Compiler auch von der `compilerbau`-Bibliothek gefunden und verwendet wird! Das geschieht vornehmlich dann, wenn auch die Virtual-Machine aus dem selben Verzeichnis wie dem des Java-Übersetzers verwendet wird...

4.2.4 Syntaxhighlighting

Syntaxhighlighting und Fehlerkennzeichnung sind bei *Visual Emugen* in eine Komponente integriert - die Klasse `compilerbau.gui.SourceCodeEditorKit`. Diese Klasse erwartet als Parameter einerseits Informationen darüber, welche Klasse zur Syntaxerkennung hergenommen werden soll, als auch eine Klasse, die über eventuell aufgetretene Fehler bescheid weiß. Wie diese Komponenten zusammengebracht werden, wird im nächsten Abschnitt behandelt.

Das Syntaxhighlighting in *Visual Emugen* wurde über einen einfachen *JFlex* - Scanner realisiert. In eigenen IDEs kann dieses Prinzip auch sehr einfach verwendet werden, da die Logik für das Syntaxhighlighting in das Paket `compilerbau` ausgelagert ist. Das Einzige, was zu tun bleibt, ist, einen Scanner zu schreiben, der die gewünschten Schlüsselworte erkennt. Dieser Scanner muss dann nur noch vom Interface `compilerbau.gui.Syntaxhighlighter` abgeleitet sein, um für das Syntaxhighlighting verwendet zu werden.

Als Rückgabeobjekte dienen Instanzen der Klasse `compilerbau.gui.Token`. Ich möchte an dieser Stelle davon abraten, zustandsbasiertes Scannen zu verwenden, da die Darstellungsroutine leider den Quelltext nur zeilenweise durchscannt, und dadurch Zustandsinformationen verliert...

```
import compilerbau.gui.*;
```

```
import java.awt.Color;
%%

%class HighlightLexer
%implements compilerbau.gui.SyntaxHighlighter
%public
%8bit
%caseless
%char
%type Token
%{
  private int startOffset;
  public HighlightLexer(java.io.Reader in, Integer startOffset){
        this(in,startOffset.intValue());
        }
  public HighlightLexer(java.io.Reader in, int startOffset){
    this.yy_reader = in;
    this.startOffset = startOffset;
  }
%}
NL  = \n | \r | \r\n
MyKeyword = "goto" | "loop" | "exit"
MyComment = / / [^\r\n]*{NL}
%%

<YYINITIAL> {

  {MyKeyword}     { return new Token(
    yychar+startOffset,
    yylength(),
    Color.blue);  }
  {MyComment}     { return new Token(
    yychar+startOffset,
    yylength(),
    Color.green); }
}

// error fallback
.|\n                { return new Token(
    yychar+startOffset,
    yylength(),
    Color.black); }
```

4.2.5 Fehlerkennzeichnung

Um das `SourceCodeEditorKit` verwenden zu können,muß man eine Klasse (am besten die Dokumentklasse) so umschreiben, dass sie `compilerbau.gui.SourceCodeErrorEmitter` implementiert, und dadurch der anzeigenden Klasse über eventuell in dem nachgefragten Be-

reich aufgetretene Fehler bescheid geben kann. Zusammen mit dem oben generierten Scanner kann so ein gültiges `SourceCodeEditorKit` erzeugt werden:

```
import compilerbau.gui.*;
import javax.swing.*;
import javax.swing.text.*;
public class MyFrame extends JFrame{
 public MyFrame(){
  JEditorPane          jep  = new JEditorPane();
  PlainDocument        doc  = new VisualEmugenDocument();
  SourceCodeEditorKit  scek = new SourceCodeEditorKit(
                        (Class) HighlightLexer.class,
           (SourceCodeEmitter) doc);
  jep.setEditorKit(scek);
  jep.setDocument (doc);
  getContentPane().add(      new JScrollPane(jep));
 }
}
```

4.2.6 Syntaxvervollständigung

Syntaxvervollständigung wie in *Visual Emugen* sollte mit Hilfe des `compilerbau`-Pakets in Zukunft kein Problem mehr darstellen - Es bedarf nur wenigen Zeilen, einem bereits bestehendem Editor die Syntaxergänzungskomponente hinzuzufügen:

```
import javax.swing.*;
import javax.swing.text.*;
import compilerbau.gui.SuffixPopup;
public class Beispiel extends JFrame{
  public Beispiel(){
    String     keywords = "goto loop exit";
    JEditorPane     jep = new JEditorPane();
    PlainDocument   doc = new PlainDocument();
    SuffixPopup   popup = new SuffixPopup(doc,jep,keywords);
    editorpane.setDocument(mydoc);
    getContentPane().add(new JScrollPane(jep));
  }
}
```

In diesem Beispiel wird ein einfacher Editor erzeugt, der ausser allen eingegebenen Wörtern auch die Schlüsselwörter “goto”, “loop” und “exit” zur Ergänzung anbietet.

4.2.7 SplashScreen

Ein Splash-Screen ist ein Merkmal moderner Programme, mit deren Hilfe sie die eventuell längere Wartezeit bis zum Start des Hauptprogrammes überbrücken. Oftmals werden diese SplashScreens auch benutzt, um dem Benutzer das Programmlogo zusammen mit der aktuellen Versionsnummer des verwendeten Programms nahezubringen.

Das compilerbau-Paket enthält nun eine Klasse, mit deren Hilfe ganz einfach Splash-Screens erzeugt werden können:

```
import compilerbau.gui.SplashScreen;
public class SplashDemo{
  public static void main(String[] args){
    SplashScreen ss = new SplashScreen(
          new ImageIcon(SplashDemo.class.getResource("logo.gif")),
          null,
          5,
          "Version␣1.0");
    // Hier nun das eigentliche Programm
    System.out.println("Programmende␣:-)");
  }
}
```

4.2.8 Installer

Ein letzter Bestandteil des compilerbau Pakets, der bisher völlig unerwähnt geblieben ist, ist der Installer. Diese Klasse erlaubt eine einfache Erstellung von Installationsdialogen für Java-Programmpakete, die unter verschiedenen Betriebssystemen installiert werden sollen. Die Verteilung von *Visual Emugen* wird im Rahmen des TUM-Praktikums “Generierung von Grafischen Oberflächen” mit Hilfe dieser Klasse geregelt.

Die Verwendung von compilerbau.gui.Installer ist denkbar einfach. Der Installer sucht standardmäßig nach einer Datei mit dem Pfad ./resources/config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <string>Mein tolles Programm</string>
 <string>MtP</string>
 <string>1.0</string>
 <string>\bin\setup.js</string>
 <string>/bin/setup.unix</string>
 <string>MeinTollesLogo.gif</string>
 <string>Installationsarchiv.zip</string>
 <string>ReleaseNotes.html</string>
</java>
```

In dieser Datei können alle möglichen Einstellungen konfiguriert werden. Dabei handelt es sich um folgende Eigenschaften:

Zeile 3 Der ausführliche Programmname

Zeile 4 Ein Kürzel für das Programm

Zeile 5 Die Versionsnummer

Zeile 6 Der Pfad im Installationsverzeichnis zum Windows-spezifischen Installations-Skript

Zeile 7 Der Pfad im Installationsverzeichnis zum Unix-spezifischen Installation-Skript

Zeile 8 Das Program-Logo im `./images/` Verzeichnis

Zeile 9 Das zu installierende Zip-Archiv im `./resources/` Verzeichnis

Zeile 10 Die nach der Installation anzuzeigende Readme-HTML-Datei

Die Verwendbarkeit des Installers wird daher hauptsächlich von den verwendeten Shell-Skripten bestimmt. Unter Windows 2000/XP hat sich die Verwendung von JavaScript zum Anlegen von Registry-Einträgen und Verknüpfungen auf dem Desktop bewährt, unter Unix wären zum Beispiel bash-Skripte zum Anlegen von Pfad-Variablen

Eine brauchbare Installationsroutine kann dadurch erzeugt werden, dass aus dem Projektverzeichnis nun ein `.jar`-Archiv erzeugt wird, das dank seinem Manifest standardmäßig die Klasse `compilerbau.gui.Installer` ausführt. Diese wird im nächsten Schritt aus der Konfigurationsdatei alle nötigen Informationen beziehen und einen benutzerdefiniderten Installationsdialog durchführen.

5 Fazit

5.1 Bewertung

An dieser Stelle soll nun eine abschliessende Bewertung des Projekts folgen, um herauszufinden, ob die Anforderungen und Ziele für *Visual Emugen* erreicht wurden, oder ob das Projekt ein Fehlschlag war.

5.1.1 Erfüllung der Ziele

Hier soll überprüft werden, wie weit die Ziele von Seite 5 erfüllt wurden.

- Farbliche Hervorhebung von Syntaxelementen erfolgt in *Visual Emugen*
- Syntaxfehler werden in *Visual Emugen* gekennzeichnet
- Durch ein Popup-Menü wird Syntaxergänzung in *Visual Emugen* bereitgestellt
- Die Umwandlung von *Emugen* -Quellcode in ein ausführbares `.jar`-Archiv wird von *Visual Emugen* geleistet, was sowohl zur einfachen Umwandlung als auch zur einfachen Weitergabe dient.
- Die Integration von *Emugen* in *Visual Emugen* erfolgt sehr flexibel, da sehr klare und dokumentierte Schnittstellen zu *Emugen* bestehen, als auch die technische Einbindung durch einen einzigen Kopiervorgang sehr einfach erfolgt.

- Die Anforderung, einen Oberflächenkonstruktionskit zur Verfügung zu stellen, wird durch das `compilerbau`-Paket erfüllt.
- Die Quellcodedokumentation erfolgte durch den Einsatz von Javadoc, und kann sehr leicht mit *Ant* generiert werden. Zudem stellt diese Arbeit selbst einen Großteil der Dokumentation dar.

Es kann also mit Fug und Recht behauptet werden, dass *Visual Emugen* die an das Werkzeug gestellten Anforderungen voll und ganz erfüllt.

5.1.2 Anwendbarkeit

Zusätzlich kann der Erfolg eines Werkzeugs in seiner Anwendung in der Praxis gemessen werden. Hier kann *Visual Emugen* mit folgenden Einsatzgebieten aufwarten:

- *Visual Emugen* soll ab dem Wintersemester 2003/2004 für das Praktikum "Generierung von Benutzeroberflächen" eingesetzt werden
- *Visual Emugen* ist mittlerweile auch beim Entwickler von *Emugen* im Einsatz und erzielt beachtliches Lob
- Ein Einsatz von *Visual Emugen* im Rahmen der Realschul-Informatik-Ausbildung wird erwogen.

5.2 Ausblick

Visual Emugen verspricht also, ein voller Erfolg zu werden. Trotz allen Vorteilen, die die Verwendung von *Visual Emugen* mit sich bringt, dürfen jedoch einige offene Punkte nicht verschwiegen werden:

- Es ist auch in Zukunft darauf zu achten, dass *Visual Emugen* auf dem Laufenden gehalten wird, und bei Bedarf an die entsprechenden neuen *Emugen* -Versionen angepasst wird
- *Visual Emugen* hat kein passendes Hilfesystem, dass auf die Bedürfnisse der Benutzer eingeht: So gibt es weder eine Online-Hilfe zur Bedienung von *Visual Emugen* noch eine Hilfe zur Verwendung der *Emugen* -Syntax. Dies wird noch dazu dadurch erschwert, dass sowieso keine aktuelle Spezifikation des *Emugen* -Leistungsumfangs existiert.
- *Emugen* parst nur *Emugen* -Quelltext nach Fehlern, der Java-Code bleibt syntaktisch ungeprüft. Eine vollständige Syntaxüberprüfung kann leider erst zur Übersetzungszeit durch den Java-Compiler erfolgen, nachdem *Emugen* die Eingabedatei in Java-Code übersetzt hat. Denkbar wäre jedoch, um die Benutzerfreundlichkeit zu erhöhen, ein zusätzliches Scanner/Parser-Gebilde, dass die Java-Fragmente, die in den Java-Blöcken von *Emugen* vorkommen auf die an dieser Stelle erkenbaren Fehler hin überprüft. Dabei

ist zum Beispiel die korrekte Klammerung der auftretenden Ausdrücke ein überprüfbares Korrektheitsmerkmal. Auch der korrekte Aufbau von Zuweisungen kann überprüft werden.

- *Emugen* stoppt bereits nach dem ersten Syntaxfehler und gibt auch nur diesen aus. Der Grund darin liegt beim verwendeten Parser-Generator CUP. Dieser beherrscht kein fortsetzungsfähiges Parsen. Dadurch entsteht in *Visual Emugen* eventuell ein unvollständiges Bild von Eingabe-Fehlern, und der Benutzer wird nicht besonders gut auf die Fehlerquelle hingewiesen. Dieser Mangel ist nur sehr schwer behebbar, und nur ein Austausch des zugrundeliegenden Parsers könnte Abhilfe leisten.