JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



# Ein visueller Editor für die Programmiersprache Monaco

BAKKALAUREATSARBEIT

(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus der technischen Wissenschaften

in der Studienrichtung

INFORMATIK

Eingereicht von:

*Christian Wirth, 0355354*

Angefertigt am:

*Institut für Systemsoftware*

Betreuung:

*o.Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck*
*Dipl.-Ing. Dr. Herbert Prähofer*

*Pasching, April 2007*

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bakkalaureatsarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Pasching, April 2007 Christian Wirth

# Kurzfassung

Die vorliegende Bakkalaureatsarbeit behandelt einen visuellen Editor für die domänenspezifische Programmiersprache Monaco (Modular Notation for Automation Control). Diese Sprache wurde am Christian Doppler Labor für Automated Software Engineering an der Johannes Kepler Universität Linz entwickelt. Ihr Einsatzgebiet ist die Maschinensteuerung. Der visuelle Editor ist Teil einer integrierten Entwicklungsumgebung, genannt Monaco-IDE, die auf Basis der Eclipse Rich Client Platform entwickelt wurde. Neben dem visuellen Editor werden in dieser Arbeit auch die dafür relevanten Bereiche der Monaco-IDE behandelt.

Ziel bei der Entwicklung der Monaco-IDE war es, ein Programmiersystem für Domänenexperten im Bereich der Maschinensteuerung zu schaffen. Dieses soll die Programme während der Implementierung auf visuelle Weise darstellen, einfach und intuitiv zu bedienen sein und möglichst wenige Programmierkenntnisse voraussetzen. Im Rahmen des Projektes waren bereits ein Compiler und weitere Werkzeuge für Monaco vorhanden. Aufbauend auf diese wurde vom Autor dieser Arbeit ein Prototyp des visuellen Editors entwickelt. Dieser wurde mit dem Eclipse Graphical Editing Framework realisiert.

Diese schriftliche Projektdokumentation in Form einer Bakkalaureatsarbeit soll einen Überblick über den aktuellen Projektstand bieten. Nach einer kurzen Einführung in die Sprache Monaco werden die verwendeten visuellen Notationen beschrieben. Im Detail wird auf die technische Implementierung des visuellen Editors eingegangen. Im Anschluss werden die Funktionen des Editors beschrieben, die auch anhand eines Beispielprojektes vorgeführt werden.

# Abstract

This bachelor thesis covers a visual editor for the domain specific programming language Monaco (Modular Notation for Automation Control). This language has been developed at the Christian Doppler Laboratory for Automated Software Engineering at the Johannes Kepler University in Linz, Austria. The field of application of this language is the domain of machine control. The visual editor is part of an integrated development environment called Monaco-IDE. This IDE is based on the Eclipse Rich Client Platform. The thesis deals with the visual editor in detail and also covers those parts of the Monaco-IDE that are relevant to the visual editor.

During development of the Monaco-IDE the main intention was to create a programming environment to be used by domain experts in the field of machine control. It should present Monaco source code in a visual notation that is easily comprehendible. The IDE shall enable domain experts with little knowledge of software engineering to write and maintain programs. Based on an already existing compiler and additional tools the author of this thesis developed a prototype of a visual editor to fulfil those requirements. The system is based on the Eclipse Graphical Editing Framework.

This project documentation in form of a bachelor thesis should give an overview of the current state of the ongoing project. After a short introduction of Monaco its visual notation is discussed. The technical implementation of the editor will be covered in detail. Finally, the features of the visual editor will be presented both in theory and applied to a sample project.

# Inhaltsverzeichnis

# 1 Einleitung und Zielsetzung

Diese Bakkalaureatsarbeit entstand am *Christian Doppler Labor für Automated Software Engineering* der Johannes Kepler Universität Linz im Projekt *Domain-specific Languages for Industrial Automation*. Industriepartner und Auftraggeber dieses Projektes ist das Linzer Unternehmen KEBA AG, das im Bereich der Automatisierungstechnik tätig ist.

Ziel des Projektes ist die Entwicklung einer domänenspezifischen Programmiersprache (*Monaco*) [Prae07] für den Bereich der Maschinensteuerung. Die Sprache ist darauf ausgelegt, auch von Domänenexperten ohne fortgeschrittene Programmierkenntnisse leicht erlernt und verwendet werden zu können.

Neben der Programmiersprache wird auch der Prototyp einer Programmierumgebung entwickelt. Dies umfasst einen Compiler, einen Interpreter mit einer Virtuellen Maschine sowie eine integrierte Entwicklungsumgebung (IDE). Der Kern dieser IDE ist ein visueller Editor. Während die erstgenannten Komponenten von den Kollegen am Institut entwickelt wurden, hat der Autor dieser Bakkalaureatsarbeit den Prototypen des visuellen Editors und große Teile der IDE entwickelt.

## 1.1 Aufgabenstellung

Die Kernkomponente bei der Entwicklung einer IDE für Monaco ist ein visueller Editor. Dieser soll die Bearbeitung von Programmcode in möglichst intuitiver, graphischer Form ermöglichen. Der visuelle Editor soll nahtlos in die Eclipse-basierte Entwicklungsumgebung eingebunden sein.

Diese Bakkalaureatsarbeit beschreibt eine prototypische Umsetzung des visuellen Programmiersystems für Monaco-Programme. Insbesondere wird die visuelle Darstellung von Monaco-Steuerungsroutinen und die Umsetzung in der Programmierumgebung behandelt. Es werden dabei auch die notwendigen Konzepte

der Sprache Monaco, die zur Umsetzung verwendeten Technologien und die Funktionen der entwickelten Benutzeroberfläche vorgestellt.

## 1.2 Struktur

Diese Bakkalaureatsarbeit ist wie folgt aufgebaut:

- In Kapitel 2 wird eine kurze Einführung in die Programmiersprache Monaco und den zu entwickelnden visuellen Editor gegeben.
- Kapitel 3 beschreibt die im Editor verwendete visuelle Notation von Monaco
- In Kapitel 4 werden die verwendeten Technologien wie Eclipse RCP oder GEF behandelt.
- Kapitel 5 befasst sich mit dem visuellen Editor, seiner Benutzeroberfläche und den angebotenen Funktionen.
- In Kapitel 6 wird mit Hilfe des visuellen Editors ein umfassendes Beispielprogramm entwickelt, um die Funktionen des Editors zu präsentieren.

# 2 Monaco

Die Sprache Monaco (*Modular Notation for Automation Control*) ist eine domänenspezifische Programmiersprache zur Steuerung von Maschinen und Robotern. Sie wurde am Christian Doppler Labor für Automated Software Engineering entwickelt. Die Sprache lehnt sich an die Ausdrucksstärke von *Statecharts* [Hare87] an, ist syntaktisch jedoch an die imperative Programmiersprache Pascal angelehnt. Auch Elemente weiterer Sprachen sind in der Sprachdefinition enthalten.

Die Ziele bei der Sprachdefinition von Monaco waren hierarchische Strukturierung, Erweiterbarkeit, Ereignisorientierung, Ausnahmebehandlung, Verifizierbarkeit und Sicherheit. Berücksichtigt wurden in der Maschinensteuerung notwendigen Konzepte wie etwa Parallelität oder asynchrone Ereignisse [Hurn06].

## 2.1 Aufbau von Monaco-Programmen

In der derzeitigen Compiler-Version besteht ein Monaco-Programm immer aus einer Datei. Darin müssen sämtliche benötigte Datentypen, Komponenten und Interfaces definiert sein.

Ein Programm besteht aus folgenden Elementen:

- Komponenten (COMPONENT)
- Interfaces für Komponenten (INTERFACE)
- Datentypen (TYPES)
- Konstanten (CONSTANTS)
- Maschineninitialisierung (SETUP)

Eine Komponente repräsentiert üblicherweise ein konkretes Maschinenbauteil, etwa einen Motor oder einen Sensor. Die Komponente kapselt alle von dieser Einheit

benötigten, verwendeten und ausgegebenen Daten und Programmteile. Sie bestehen aus:

- Funktionen ohne Nebeneffekt (FUNCTION)
- Steuerungsroutinen (ROUTINE)
- Lokalen Variablen (VARS)
- Ereignissen (EVENTS)
- Parametern (PARAMETER)
- Unterkomponenten (SUBCOMPONENT)

Während natürlich alle in den obigen beiden Listen angeführten Daten in geeigneter Weise in der IDE angezeigt werden, sind für diese Arbeit insbesondere Routinen und Funktionen von Bedeutung. Diese enthalten die eigentliche Steuerungslogik in Form von Anweisungen. Eben diese Anweisungen werden vom visuellen Editor dargestellt und sind daher für diese Arbeit von zentraler Bedeutung.

Das Linken von Sourcecodes aus mehreren unterschiedlichen Quellcode-Dateien zu einer ausführbaren Einheit ist derzeit nicht implementiert. Ebenso können in Monaco keine Bibliotheken geschrieben oder eingebunden werden. Beides ist aber durch den aktuellen Entwicklungsstand bedingt und stellt keine generelle Einschränkung von Monaco dar.

## 2.2 Visueller Editor

Herkömmliche Programmierumgebungen erlauben meist nur das Bearbeiten des Quelltextes. Diese Art der Programmierung erfordert jedoch ein umfassendes Wissen über Programmierung im Allgemeinen und die Syntax der Sprache im Speziellen. Zwar ist die Sprache Monaco bereits für eine leichte Lesbarkeit durch Domänenexperten ausgelegt. Der visuelle Editor soll die Benutzerfreundlichkeit jedoch weiter erhöhen, die Bedienung vereinfachen und Fehler verringern.

Dazu soll es möglich sein, Monaco-Programme in einer graphischen Notation darzustellen. Diese Notation soll sich am Verständnis des Domänenexperten orientieren. Dieser soll auf einen Blick erkennen können, was vom abgebildeten Codestück bewirkt wird.

Programme in der graphischen Notation sollen direkt bearbeitet werden können. Hierzu ist die Nutzung von *Drag & Drop* sowie *Copy & Paste* von graphischen Elementen vorgesehen. Beim Anlegen neuer Objekte sollen jeweils alle Möglichkeiten vorgeschlagen werden (*Autovervollständigung*). Das Arbeiten mit dieser graphischen Darstellung soll möglichst intuitiv sein.

Aufbauend auf die *Eclipse Rich Client Platform* [Mcaf05] soll eine Entwicklungsumgebung erstellt werden, die den erwarteten Komfort bietet. Diese soll den von modernen IDEs gewohnten Funktionsumfang bieten: Projektverwaltung, automatisches Kompilieren, Starten von generierten Programmen, benutzerfreundlicher Sourcecode-Editor mit Syntaxhervorhebung und Autovervollständigung, im Endausbau auch eine Debugging-Umgebung. Ebenso soll diese den graphischen Editor optimal integrieren. So soll der Benutzer Codestücke sowohl textuell als auch graphisch editieren können, diese Ansichten müssen wechselseitig jeweils am aktuellen Stand gehalten werden. Für den Endbenutzer dieser Entwicklungsumgebung stellt der visuelle Editor die Kernkomponente dar.

# 3 Visuelle Notation von Monaco

Die wesentliche Herausforderung der Arbeit am visuellen Editor war die Entwicklung einer simplen, aber ausdrucksstarken graphischen Notation für Monaco. Diese muss natürlich alle syntaktisch korrekten Monaco-Programme darstellen können. Im Sinne des Projektziels soll diese Darstellung aber auch für Domänenexperten möglichst einfach zu verstehen sein. Programmierkenntnisse sollten für ein Arbeiten mit dem visuellen Editor möglichst nicht Voraussetzung sein.

In der bisherigen Ausbaustufe des Prototypen können Unterprogramme (ROUTINE und FUNCTION) angezeigt werden. Dies umfasst insbesondere atomare Anweisungen (etwa Ein- und Ausgaben, Zuweisungen, Routinenaufrufe, WAIT, RETURN), Kontrollstrukturen (wie etwa IF, LOOP, WHILE, PARALLEL) und die Ereignisbehandlung (in Form von ON-Handlern). Ebenso werden auch lokale Variablen und Parameter dargestellt.

Ebenfalls bereits implementiert ist die Darstellung des Konfigurations- und Initialisierungsabschnittes (SETUP). Dieser Code wird beim Start der Programmausführung abgearbeitet. Es werden dabei die einzelnen Komponenten statisch miteinander verbunden, parametrisiert und der Einstiegspunkt in die Programmausführung festgelegt. Zur Laufzeit können die während des SETUP getroffenen Einstellungen nicht mehr verändert werden.

## 3.1 Konzepte

Die visuelle Notation von Monaco basiert auf einigen grundlegenden Konzepten. Einige davon ergeben sich unmittelbar aus der Sprachdefinition. Andere wiederum

wurden speziell für die graphische Darstellung gewählt, um diese einfach und leicht erkennbar anzeigen zu können.

- **Blockorientierung**: Monaco verwendet ein Blockkonzept, wie es bei gängigen Programmiersprachen üblich ist. Dieser werden in der visuellen Darstellung durch unterschiedlich gefärbte Rechtecke repräsentiert.
- **Anweisungssequenzen**: Innerhalb von Blöcken werden Sequenzen von Anweisungen vertikal untereinander angeordnet. Dies symbolisiert den sequentiellen Kontrollfluss.
- **Parallelitäten**: parallele Handlungsstränge werden horizontal nebeneinander dargestellt.
- **Ereignisbehandlung**: die Ereignisbehandlung erfolgt in der zweiten Dimension. Ereignisbehandlungsroutinen sind rechts außen, an Blöcke gekoppelt, dargestellt.
- **Einklappbarkeit**: alle komplexeren Elemente lassen sich einklappen.

### 3.1.1 Blöcke

Die visuelle Notation von Monaco ist direkt aus der textuellen Syntax abgeleitet. Da die Blockstruktur zentrales Element der Routinen ist, sind Blöcke auch das Grundkonzept der visuellen Darstellung. Da jeder Block eine gültige Anweisung ist, können sie auch geschachtelt werden, wodurch ein Baum von Blöcken aufgespannt wird. Alle weiteren Anweisungen müssen sich innerhalb eines Blockes befinden. Unterprogramme definieren implizit einen Block, in dem sich die Anweisungen des Unterprogramms befinden.

Um Blöcke auf unterschiedlichen Ebenen leichter unterscheidbar zu machen und damit die Blockstruktur zu betonen, werden diese mit unterschiedlichen Farbtönen darstellt.

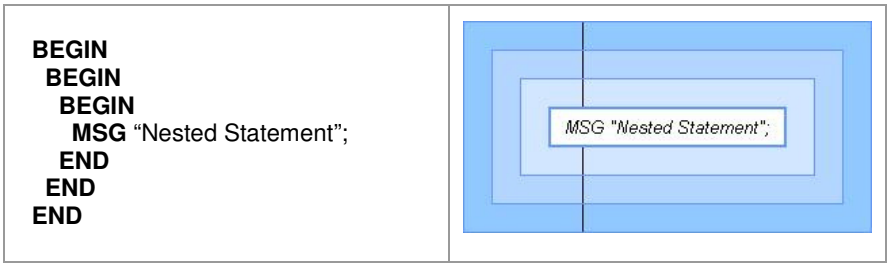


Abbildung 1: unterschiedlich gefärbte Blöcke

### 3.1.2 Anweisungssequenzen

Wichtigstes Konzept innerhalb der Blöcke sind Sequenzen von Anweisungen. Diese werden untereinander in der Reihenfolge ihrer Ausführung angeordnet. Die Ausführungssequenz wird durch eine Linie verdeutlicht, die die Anweisungen verbindet. Optional kann diese Linie um Pfeil-Symbole ergänzt werden.

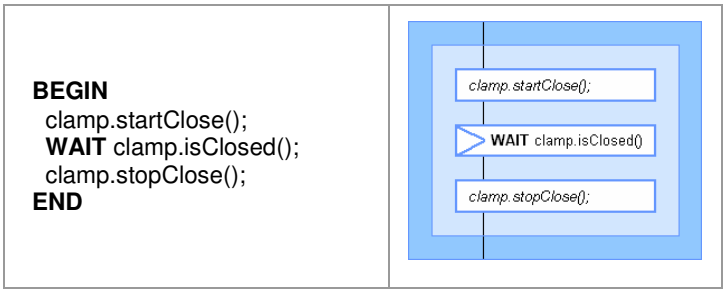


Abbildung 2: Sequenz von Anweisungen

Blöcke sind ebenfalls Anweisungen. Sie können sich daher ebenso in Anweisungssequenzen befinden. Auf diese Weise können Blöcke auch beliebig geschachtelt werden.

Während wir die Pfeilchen zwischen den Anweisungen während der Entwicklung zuerst als notwendig angesehen hatten, haben wir später erkannt, dass sie den natürlichen Lesefluss mehr behindern als fördern. Sie sind daher nun optional und können ein- und ausgeschaltet werden.

### 3.1.3 Ereignisbehandlung

Eine wichtige Funktion von Blöcken ist die Ereignisbehandlung für darin enthaltene Anweisungen. Jedem Block kann eine beliebige Anzahl an ON-Handlern zugewiesen werden. Den Handlern ist jeweils eine boolesche Bedingung zugeordnet.

Befindet sich das Programm bei der Ausführung innerhalb des Blockes wird an klar definierten Haltepunkten (etwa WAIT-Anweisungen) überprüft, ob die Bedingung eines der Handler TRUE ergibt. Ist dies der Fall, so wird der normale Programmfluss angehalten und stattdessen der Handler ausgeführt. Dieser kann nun entsprechend der eingetretenen Bedingung reagieren. Anschließend kann er entscheiden, ob die Ausführung des Blockes fortgeführt werden kann (an der Stelle, wo unterbrochen wurde) oder ob der Block beendet wird. In diesem Fall wird mit der Programmausführung mit der ersten Anweisung nach dem Block fortgefahren.

Es werden nicht nur die jeweils innersten Handler ausgewertet. Tatsächlich werden alle Handler geprüft, die zu einem Block gehören, der die aktuelle Anweisung dynamisch umgibt. Es werden also alle ON-Handler aller Blöcke des aktuellen Call stacks ausgewertet. Dies ermöglicht eine einfachere, ebenfalls hierarchisch angeordnete Fehlerbehandlung. Generelle, an vielen Stellen mögliche Fehler können so möglichst allgemein, an einigen wenigen Stellen behandelt werden. Spezielle Fehler hingegen werden genau dort behandelt, wo sie auftreten, um möglichst fallspezifisch auf diese reagieren zu können.

Das Konzept der ON-Handler ist hauptsächlich zur Fehlerbehandlung gedacht. Es ähnelt insofern dem bekannten *Exception*-Konzept in Java. Da jedoch nicht nur Exceptions, sondern beliebige boolesche Bedingungen und Ereignisse überwacht werden können, kann es viel allgemeiner genutzt werden. Semantisch sind ON-Handler daher mächtiger als Exceptions. So lassen sich damit etwa unmittelbar Zeitüberschreitungen (Timeouts) modellieren oder komplexere Fehlerbedingungen prüfen. Im Gegensatz zu Exceptions werden Handler in definierten Programmzuständen überprüft, während Exceptions aktiv ausgelöst werden.

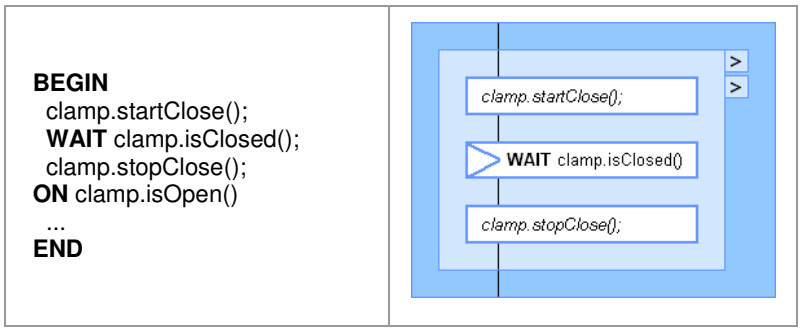


Abbildung 3: ON-Handler, eingeklappt

Visuell dargestellt werden die Handler außerhalb des normalen Ablaufes, auf der rechten Seiten an Blöcke angehängt. Im Normalfall sind die Handler eingeklappt. Sie nehmen also weder Platz weg noch beeinträchtigen sie den Lesefluss. Erst durch Aufklappen kann der Code betrachtet und manipuliert werden. Dies entspricht dem Gedanken, dass die Handler Code enthalten, der nur in Ausnahmesituationen (Fehler, Ereigniseintritt) ausgeführt wird und für den normalen Ablauf nicht relevant ist.

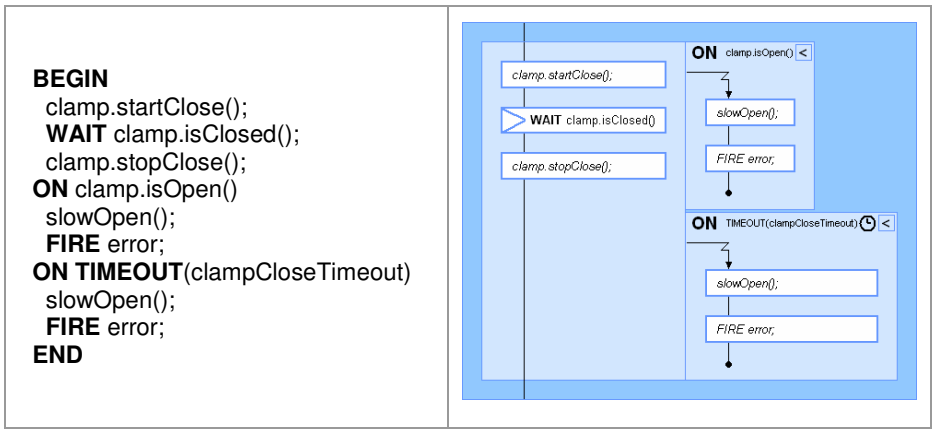


Abbildung 4: ausgeklappte ON-Handler

### 3.1.4 Einfache Anweisungen

Viele Anweisungen können im Editor auf sehr einfache Weise dargestellt werden. Es sind dies etwa Ein- oder Ausgabeanweisungen, Wertzuweisungen, Unterprogrammaufrufe oder simple Anweisungen wie WAIT oder RETURN. Diese stellen im Syntaxbaum zwar nicht unbedingt die Blätter dar, können aber selber keine weiteren Anweisungen beinhalten.

Deren Funktion ist bei sauberer Datenabstraktion und gutem Programmierstil sehr leicht aus dem Sourcecode ablesbar. Sie können daher auch unmittelbar in Form ihres Sourcecodes dargestellt werden. Manche wichtige Anweisungen wie etwa RETURN oder WAIT werden zusätzlich mittels Fettschreibung oder Icons besonders hervorgehoben.

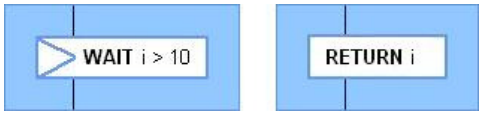


Abbildung 5: Blöcke mit einer einzelnen WAIT- und RETURN-Anweisung

### 3.1.5 Kontrollstrukturen

Kontrollstrukturen sind zusammengesetzte Anweisungen wie bedingte Anweisungen (IF), Schleifen (LOOP, WHILE) oder die Parallelausführung mehrerer Anweisungen (PARALLEL). Im Gegensatz zu einfachen Anweisungen enthalten diese weitere Anweisungen. Für ihre visuelle Repräsentation sind spezielle Darstellungsformen

vorgesehen. Die im Projekt gewählte visuelle Notation dieser Elemente wurde hinsichtlich einer intuitiven Erkennbarkeit ihrer Semantik gewählt.

### Parallelausführung

Die PARALLEL-Anweisungen spalten den sequentiellen Programmfluss logisch in zwei oder mehr Stränge auf. Im Gegensatz zur vertikal angeordneten Anweisungslisten werden die Anweisungen innerhalb von PARALLEL horizontal nebeneinander angeordnet. Dies repräsentiert die gleichzeitige Ausführung aller parallelen Anweisungsstränge.

```
PARALLEL
  BEGIN
    clamp.startClose();
    WAIT clamp.isClosed();
    clamp.stopClose();
  END
||
  BEGIN
    WAIT clamp.position() >= IPOS;
    core.insert();
  END
END
```

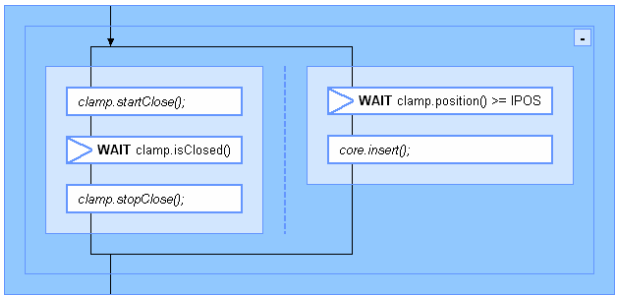


Abbildung 6: Eine PARALLEL-Anweisung mit zwei parallelen Blöcken

### Schleifen

Die Darstellung von Schleifen ist einfach und intuitiv. Ein Rechteck mit abgeschrägten Ecken symbolisiert den Schleifenkopf. Dem sequentiellen Schleifenrumpf folgt eine Kante, die zum Schleifenkopf zurück führt und dabei den Rücksprung symbolisiert.

Wird die Ausführung der Schleife hingegen beendet, wird mit den auf die Schleife folgenden Anweisungen fortgesetzt.

```
WHILE i<10
  BEGIN
    MSG i;
    i := i+1
  END
```

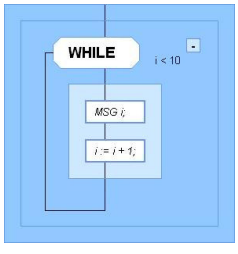


Abbildung 7: Einfache WHILE-Schleife

### Bedingte Anweisung

Problematisch bei der Wahl der Darstellung waren Verzweigungen in Form bedingter Anweisungen (IF-Anweisung). Dabei wird von mehreren möglichen (einer bis theoretisch beliebig vielen) Handlungssträngen höchstens einer ausgeführt. Anschließend wird mit der auf das IF-Statement folgenden Anweisung fortgefahren. Diese Anweisung stellt also keinesfalls eine Sequenz, sondern eine Auswahl dar.

Entsprechend der eingangs formulierten Logik müssten die möglichen Handlungsstränge also in der zweiten (horizontalen) Dimension nebeneinandergelegt werden, da eben nicht alle Anweisungen ausgeführt werden wie bei einer Liste. Dabei ist aber zur befürchten, dass die Nutzung sogenannter IF-Kaskaden zu einer sehr breiten Darstellung führen würde. Außerdem wäre eine horizontale Darstellung der IF-Anweisung sehr leicht mit der ebenfalls horizontalen PARALLEL-Anweisung zu verwechseln gewesen.

Visuell werden die Alternativen daher vertikal untereinander dargestellt. Die Handlungsstränge sind zum Hauptstrang jedoch vertikal etwas eingerückt, um die Aufspaltung in mehrere Möglichkeiten zu verdeutlichen. Im Hauptstrang liegen lediglich die Symbolisierungen der Bedingungen, die, im Falle einer Auswertung auf TRUE, zur jeweiligen Anweisung verzweigen.

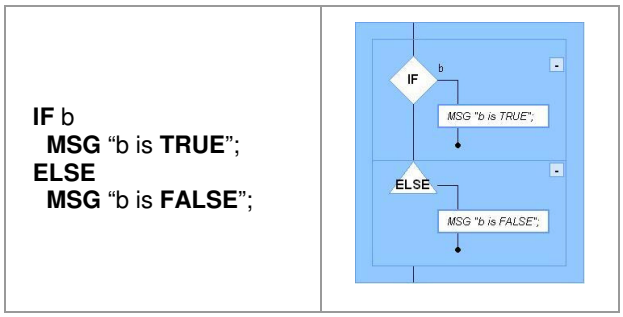


Abbildung 8: IF-Anweisung mit boolescher Variable als Bedingung

# 4 Verwendete Technologien

Die Monaco-IDE wurde in der Programmiersprache Java entwickelt. Als Entwicklungsumgebung kam Eclipse (www.eclipse.org) zum Einsatz. Eclipse wurde gleichzeitig auch als Basis für die Entwicklung der IDE genutzt, wie im folgenden Kapitel beschrieben wird.

Eclipse hat sich in den letzten Jahren zu einer der beliebtesten IDEs für die Java-Programmierung entwickelt. Eclipse ist aber mehr als eine Java-IDE. Tatsächlich stellt es ein erweiterbares Framework für die Entwicklung von Programmen, insbesondere von Entwicklungsumgebungen, zur Verfügung.

Eclipse basiert auf einem Plug-In-Modell. Aufbauend auf einem Kern, der auf das Nötigste beschränkt wurde, ist die übrige Funktionalität der mächtigen Plattform in Plug-ins [Lude07] verpackt. Diese lassen sich beliebig austauschen und erweitern. Vorhandene Funktionalität kann problemlos in eigenen Plug-ins genutzt und so bei Bedarf um eigene Programmteile erweitert werden.

Auf dieser Grundlage existiert heute nicht nur die bekannte Java-IDE, sondern auch eine Vielzahl weiterer Entwicklungsumgebungen, etwa für C/C++, Perl oder PHP. All diese greifen auf einen Satz von Plug-ins zu, was die Erstellung einer Entwicklungsumgebung deutlich vereinfacht. Doch Eclipse kann auch ohne diese speziellen IDE-Plug-ins verwendet werden. Es stellt dann eine allgemeine Applikations-Plattform dar, auf Basis derer sich eine Vielzahl von Rich-Client-Applikationen entwickeln lassen.

## 4.1 Rich-Client-Platform

Ein *Rich Client* [Mcaf05] ist ein Programm, das eine komplexe graphische Benutzerschnittstelle zur Verfügung stellt. Diese soll dem vom Benutzer erwarteten Verhalten ähnlicher Programme entsprechen. Funktionen wie Drag & Drop oder eine

Zwischenablage sollen nach Möglichkeit von einem Rich Client angeboten werden. Dies alles soll das Arbeiten mit dem Programm so einfach wie möglich gestalten. Der Benutzer soll sich auf die zu bearbeitenden Daten konzentrieren können, anstatt auf das System.

Das Gegenteil eines Rich Clients ist ein S*imple client* (oder auch *Thin client*), etwa ein Terminal-Dienst auf Kommandozeilenbasis oder ein browserbasierter Client. Diese sind hinsichtlich der Benutzerinteraktion oft stark eingeschränkt.

Problematisch bei der Entwicklung von Rich Clients ist die Tatsache, dass viele der dabei benutzten Technologien auf verschiedenen Betriebssystemen unterschiedlich implementiert sind. Jede moderne graphische Betriebssystemoberfläche bietet eine Fensterverwaltung und eine Vielzahl an GUI-Elementen an. Funktionen wie etwa Zwischenablage (Copy & Paste) oder Drag & Drop werden unterstützt. Die Programmierung dieser Techniken unterscheidet sich jedoch je nach eingesetztem Betriebssystem. Dies führt dazu, dass ein erheblicher Mehraufwand betrieben werden muss, um ein Programm auf unterschiedlichen Systemen mit zumindest ähnlicher Benutzerschnittstelle anbieten zu können.

Als Abhilfe dafür gibt es Rich-Client-Plattformen. Diese stellen - wie sonst ein Betriebssystem - gewisse Dienste zur Verfügung. Die Plattform kann dabei auch die vom Betriebssystem angebotenen Funktionen nutzen und diese lediglich kapseln. Dies geschieht aber jedenfalls betriebssystemunabhängig. Ein auf der Plattform basierendes Programm muss im Allgemeinen nicht wissen, auf welchem Betriebssystem es konkret ausgeführt wird. Trotzdem kann es alle von der Plattform angebotenen Funktionen nutzen.

## 4.2 Eclipse Rich Client Platform

Die *Eclipse Rich Client Platform* (RCP) ist eine derartige Plattform. Auf ihr basierende Programme können mit nahezu identischer Benutzerschnittstelle auf vielen verschiedenen Betriebssystemen ausgeführt werden (Windows, Linux und andere UNIX-Varianten, Mac OSX, QNX, Windows CE und so weiter). Neben dem bekanntesten Beispiel der bereits erwähnten Java-IDE, wird die RCP mittlerweile von etlichen Projekten genutzt. Das wohl bekannteste Programm ist der populäre BitTorrent-Client *Azureus*, aber auch spezialisierte Anwendungen wie das Steuerungs-

und Kontrollsystem für den Mars-Roboter Spirit, genannt *Maestro*, verwenden die Eclipse RCP. Weitere Beispiele sind auf der Homepage der Platform angeführt (http://www.eclipse.org/community/rcpcp.php).

### 4.2.1 Konzepte

Die wichtigsten Konzepte der RCP sind:

- Komponentenkonzept
- Einheitliches Design der Benutzeroberfläche
- Portabilität
- Bibliotheken

#### Komponentenkonzept

Eclipse folgt dem Grundsatz „everything is a plug-in“. Fast die gesamte Funktionalität des Frameworks ist in Plug-ins verpackt. Dies entspricht einem komponentenbasierten Konzept. Die Begriffe Plug-In, Komponente oder Modul werden hier synonym verstanden.

Die Platform kann jederzeit um Plug-ins erweitert werden. Diese sind versioniert, auch unterschiedliche Versionen eines Plug-ins können gleichzeitig installiert sein.

#### Einheitliches Design der Benutzeroberfläche

Für Eclipse wurde das *Standard Widget Toolkit* (SWT) entwickelt. Dieses ermöglicht einen einfachen und effizienten Zugriff auf die vom jeweiligen Betriebssystem angebotenen GUI-Elemente. So sieht jeder Benutzer die Anwendung in dem ihm gewohnten Bild. Wird das Programm unter Windows ausgeführt, so wird die dort übliche Darstellung von GUI-Elementen verwendet. Unter Linux werden die Eigenschaften des jeweiligen „GUI-Toolkits“ verwendet. Der Programmierer braucht sich aber nicht um die Besonderheiten des jeweiligen Betriebssystems kümmern. Das Framework abstrahiert das System und vereinfacht so die Nutzung.

#### Portabilität

RPC-Programme lassen sich überall dort ausführen, wo eine Java-Umgebung und eine SWT-Implementierung zur Verfügung stehen. Dadurch wird ein großer Nachteil herkömmlicher Rich Clients umgangen, die üblicherweise auf ein bestimmtes Betriebssystem beschränkt waren. Java und das SWT-System sind hingegen auf praktisch allen Systemen verfügbar.

#### Bibliotheken

Mit der Eclipse RCP wird ein umfangreicher Satz an Funktionalität mitgeliefert. Vieles davon kann in eigenen Projekten wiederverwendet werden. So lässt sich etwa eine IDE relativ einfach selber erstellen, da viele der bereits für die Java-IDE entwickelten Funktionen genutzt werden können.

### 4.2.2 Architektur von Eclipse

Die in viele einzelne Plug-ins aufgeteilte Funktionalität von Eclipse lässt sich in mehrere Schichten einteilen. Dies wird hier am Beispiel der Entwicklungsumgebung für Java-Plug-ins erläutert. Werden andere Komponenten auf oberster Ebene verwendet, so ändern sich dadurch eventuell auch weiter unten liegende Schichten. Lediglich die unterste, die RCP, wird immer benötigt.

Die unterste Schicht ist die *Rich Client Platform*. Diese stellt wie bereits dargestellt grundlegende Dienste zur Verfügung, die in vielen Applikationen nützlich sind. Typische Module dieser Schicht sind *SWT*, *JFace* oder die *Workbench*.

Die zweite Schicht ist die *IDE Platform*. Diese stellt sprachunabhängige Dienste zur Entwicklung einer IDE zur Verfügung. Es sind dies etwa die Bereitstellung eines Workspace, Debugging, Refactoring und ähnliches.

Die dritte Schicht stellen die *Java Development Tools* (JDT) dar. Diese modellieren eine Java-spezifische IDE. Hier werden die nur für Java benötigten Funktionen angeboten (etwa *JUnit*) oder die bestehenden Dienste der IDE-Platform um Java-Funktionalitäten erweitert (etwa Debugging).

Die oberste Ebene sind die auf die JDTs aufbauenden *PlugIn Development Tools*. Diese erweitern wiederum die generellen Java-Fähigkeiten der Platform um diejenigen, die für die Programmierung eines Java-Eclipse-Plug-ins notwendig sind. Es sind dies etwa

spezielle Dialoge und Funktionen zur Implementierung und Konfigurierung eines Plug-ins [Wimm06].

## 4.3 Graphical Editing Framework (GEF)

Das *Graphical Editing Framework* (GEF) wurde und wird von IBM entwickelt. Es ist ein Plug-in und soll das Implementieren von graphischen Editoren vereinfachen und stellt dafür die notwendigen Mechanismen und Konstrukte zur Verfügung. Aufbauend auf der Eclipse RCP stellt es unter anderem folgende Features zur Verfügung:

- die Graphik-Bibliothek *draw2D*
- ein Framework, um eine MVC-Architektur in hierarchischer (baumartiger) oder allgemein graphenartiger Form generieren zu können
- Verbindungslinien mit automatischer Wegfindung zwischen Elementen (etwa in UML-Diagrammen oder Baum-/Graphendarstellungen hilfreich)

Die von einem Parser generierte Datenstruktur, ein sogenannter abstrakter Syntaxbaum, stellt zumindest im Groben einen Baum dar. Aufgrund der gebotenen Konzepte eignet sich GEF daher sehr gut für die Erstellung eines graphischen Editors für eine solche Datenstruktur. Der Syntaxbaum lässt sich sehr leicht hierarchisch darstellen. Auch lässt sich die Blockstruktur der imperativen Programmiersprachen (wie etwa C, Pascal, Java oder auch Monaco) auch bereits mit einfachen Mitteln visuell umsetzen. Schon mit der Darstellung von ineinander geschachtelten Rechtecken lässt sich die Blockstruktur intuitiv dem Benutzer anzeigen.

GEF wird meist dazu eingesetzt, Eclipse-Plug-ins zu entwickeln. Es ist aber nicht auf diesen Anwendungsbereich beschränkt und kann auch außerhalb von Eclipse benutzt werden. Um das Framework nutzen zu können, muss der Programmierer lediglich einen *SWT-Viewer* anlegen, auf dem GEF seine graphische Darstellung anbringen kann. Auch dieser wird üblicherweise von GEF bereit gestellt. Für den visuellen Editor für Monaco wurde etwa ein ScrollingGraphicalViewer verwendet. Daneben sind nur einige wenige weitere Schritte notwendig, um die ersten Daten anzeigen zu können.

### 4.3.1 Model-View-Controller-Architektur

Die *Model-View-Controller*-Architektur (MVC) [KP88] wird eingesetzt, wenn Daten angezeigt und editiert werden sollen. Es stellt daher auch für einen graphischen Editor die entsprechende Basisarchitektur dar.

Die Idee hinter MVC ist, den Code des Datenmodells von dem der visuellen Darstellung der Daten zu trennen. Um die Daten zwischen Model und Anzeige konsistent zu halten ist ein drittes Modul, der Controller, notwendig. Diese drei Teile sind fundamentale Komponenten eines jeden Systems, welches Daten speichern, anzeigen und bearbeiten kann.

GEF verwendet diese Architektur in leicht abgewandelter Form:

- **Model**: Das *Model* umfasst die Datenstrukturen, die die eigentlichen Daten kapseln. Diese regeln den Zugriff auf die Daten und stellen dadurch deren Konsistenz sicher. Auch das Benachrichtigungsmodell, das bei Datenänderungen zum Einsatz kommt, ist in diesem Modul implementiert.
- **View**: Die *View* ist für die Darstellung der Daten auf dem Bildschirm verantwortlich. Für die Anzeige der Daten hat es Zugriff auf das jeweils korrespondierende Model-Objekt. Meistens existiert für jedes dargestellte Objekt des Models genau ein View-Objekt, dies muss aber nicht unbedingt sein.
- **Controller**: Der *Controller* ist der für die Steuerung des Systems zuständige Code. Er kommt etwa beim Ändern von Daten oder bei der Reaktion auf Benutzereingaben zum Tragen. Vom Model wird er über Änderungen benachrichtigt und beauftragt anschließend die View, sich entsprechend zu aktualisieren. Dies ist der größte Unterschied zur ursprünglichen MVC-Architektur, bei der auch zwischen View und Model eine Verbindung besteht [Lude07]. Bei Datenänderungen im Model würde die View direkt benachrichtigt werden, ohne Zwischenschritt über den Controller.

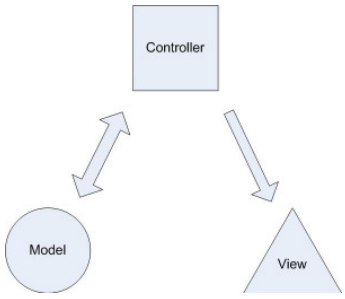


Abbildung 9: MVC-Architektur

Die Trennung in drei unabhängige Schichten hat mehrere Vorteile. Nicht nur ist das Programm durch die klare Aufgabenverteilung übersichtlicher strukturiert, auch lassen sich die einzelnen Module leichter austauschen. So kann etwa das Model von einem arbeitsspeicherbasierten Konzept in eines, dass die Daten auch während des Betriebes in einer Datenbank speichert, geändert werden, ohne dass dies eine Auswirkung auf die View hätte. Ebenso kann die Darstellung der Daten in der View grundlegend geändert werden, ohne dass sich dadurch das Model ändern müsste. Dies ist etwa der Fall, wenn das Programm von einem Endgerät auf ein anderes übertragen wird, beispielsweise von einem PC auf ein Mobiltelefon mit höchst unterschiedlichen Darstellungsmöglichkeiten. Auch können die Model-Objekte problemlos gleichzeitig von mehreren Views dargestellt werden.

Die wichtigste Aufgabe kommt dem Controller zu. Dieser regelt den Datenfluss zwischen View und Model. Er bestimmt damit auch, welche Daten von der View angezeigt werden. Auch der Controller ist aber letztlich von den Benutzereingaben oder einer anderen Kontrollinstanz abhängig und führt nur die so erhaltenen Befehle aus.

### Umsetzung in GEF

GEF setzt das MVC-Konzept unmittelbar um. Der Programmierer stellt dem Framework eine Struktur zur Verfügung, die die Daten kapselt. Diese Datenstruktur stellt das Model dar. Üblicherweise besteht dieses aus mehreren Objekten, oft unterschiedlichen Typs. Die Model-Objekte sind untereinander meist in Form eines Baumes oder eines Graphen verbunden. Mit Hilfe des *Factory*-Entwurfsmusters [GOF95] werden für alle diese Objekte sogenannte *EditParts* erzeugt. Diese stellen die Controller der Objekte dar. Hierbei ist zu beachten, dass jedes Model-Element einen eigenen EditPart und somit einen eigenen Controller hat.

Vom EditPart kann bei Bedarf ein View-Objekt angelegt werden, das am Bildschirm angezeigt wird. Sowohl die EditParts als auch die Views sind auf die gleiche Weise wie die Model-Objekte hierarchisch miteinander verbunden oder ineinander geschachtelt. Im Normalfall führt dies dazu, dass für jedes Model-Objekt genau ein EditPart und genau ein View-Objekt existieren. Andere Konstellationen sind möglich, werden aber nur selten eingesetzt.

Diese klar definierte Beziehung zwischen Model, View und Controller erleichtert den Umgang mit Benutzereingaben: das Framework stellt etwa bei einem Mausklick fest,

welches das hierarchisch tiefste Anzeige-Element ist, auf das geklickt wurde. Dessen EditPart wird über diese Aktion informiert und erhält die Gelegenheit, sein Model-Objekt zu ändern. Im Folgenden wird dieses Konzept für die Behandlung von Benutzereingaben im Detail beschrieben.

### 4.3.2 Behandlung von Benutzereingaben in GEF

Die Behandlung von Benutzereingaben in GEF ist relativ komplex. Hier ist der größte Lernaufwand notwendig, wenn man neu mit GEF zu arbeiten beginnt. Im Endeffekt ist das System jedoch logisch aufgebaut und überaus mächtig. Das Framework nimmt dem Programmierer eine Menge Arbeit ab. Nach einer entsprechenden Lernkurve ist man bei der Entwicklung mit GEF wesentlich produktiver [Wimm06].

Entsprechend dem MVC-Konzept ist bei Benutzereingaben das Model gemäß den Angaben des Benutzers zu verändern. Dies ist die Aufgabe des Controllers. Dieser ist auch für die anschließende Aktualisierung der Anzeige verantwortlich. Auch GEF funktioniert im Groben nach diesem Prinzip, es sind allerdings mehr Zwischenschritte notwendig.

#### Ablauf

Die Manipulation beginnt damit, dass der Benutzer ein Tool in der Palette auswählt. Meist ist ein *Selection-Tool* vorausgewählt. Der Mauszeiger kann sich je nach Werkzeug anpassen, um den aktuellen Zustand zu symbolisieren.

Mit diesem Werkzeug klickt der Benutzer im Editor oder führt eine komplexere Aktion durch (etwa Drag & Drop). Der Editor wird über die Aktion informiert und erzeugt mit Hilfe des Werkzeuges einen entsprechende *Request*. Dieser wird von einem von der Klasse Request abgeleiteten Objekt repräsentiert.

Vom Framework wird dieser Request an den betroffenen EditPart weitergeleitet. Üblicherweise ist dies der oberste EditPart, dessen View sich an der angeklickten Stelle befindet. Dieser behandelt den Request aber nicht selber, sondern leitet ihn an die mit ihm verbundenen *EditPolicies* weiter.

Die EditPolicies prüfen nach dem Prinzip der „*Chain of Responsibility*“ [GOF95] der Reihe nach, ob ihnen der Request bekannt ist. Ist dies bei einer EditPolicy der Fall, so erzeugt es ein *Command*-Objekt.

Das Command-Objekt wird ausgeführt und verändert dabei das Model. Dies setzt den Benachrichtigungsapparat in Gang, der darauf hin zumindest die EditParts der veränderten Model-Elemente über die Änderung benachrichtigt. Diese aktualisieren nun typischerweise ihre View entsprechend den geänderten Daten, um den aktuellen Zustand anzuzeigen.

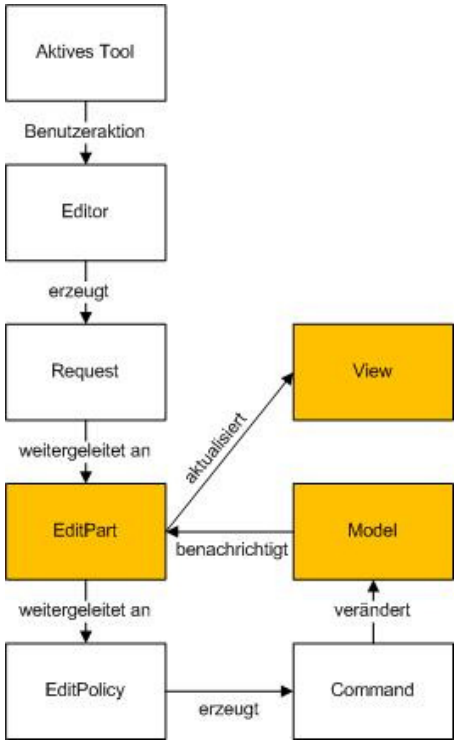


Abbildung 10: Typischer Ablauf bei der Behandlung einer Benutzereingabe

Im Folgenden werden die beim Ablauf involvierten Objekte näher beschrieben.

### Tools

Ein *Tool* definiert einen Zustand, in dem sich der Mauscursor befindet. Je nach aktivem Tool bewirkt eine Mausaktion wie etwa Klick, Bewegung oder Ziehen eine andere Reaktion. Die verfügbaren Tools sind im Editor auf der Palette angeordnet und können dort ausgewählt werden. Typische Beispiele für Tools sind Werkzeuge zum Selektieren, zum Einfügen neuer Objekte oder zum Anlegen von Verbindungslinien.

Nach dem Setzen einer Benutzeraktion ist das aktive Tool dafür verantwortlich, einen entsprechenden *Request* zu erstellen.

### Requests

*Requests* kapseln Benutzereingaben. Es handelt sich um Objekte, deren Klasse von der Klasse Request abgeleitet ist. Gespeichert wird aber nicht unmittelbar die Benutzereingabe, sondern bereits eine Bitte um Reaktion darauf. Das Tool hat aus der ursprünglichen Eingabe bereits abgeleitet, was der Benutzer wollte, etwa ein Objekt anlegen, löschen oder verschieben. Diese Information wird in einem Datenobjekt gemeinsam mit den notwendigen Parametern gespeichert.

### EditPolicies

Jedem EditPart ist eine beliebige Anzahl an sogenannten *EditPolicies* zugeordnet. Wie oben ausgeführt sind die EditParts zwar die Kontroller der Datenobjekte. Üblicherweise entscheiden aber nicht die EditParts direkt, wie auf Benutzereingaben reagiert wird. Diese Verarbeitung eines Requests wird von den EditPolicies vorgenommen.

Eine EditPolicy ist ein von der Klasse EditPolicy abgeleitetes Objekt. Es beschreibt, auf welche Requests der EditPart reagieren soll und durch welches Command es zu manipulieren ist. Jedem EditPart können beliebig viele Policies zugeordnet sein. Dadurch können ähnliche Aufgaben in einer Policy gekapselt und einfach wiederverwendet werden.

Typische EditPolicies sind etwa SelectionEditPolicy, DirectEditPolicy oder ConnectionEditPolicy. Die SelectionEditPolicy beschreibt, wie auf den Versuch einer Selektierung zu reagieren ist. Die DirectEditPolicy erlaubt das textuelle Bearbeiten von Objekten direkt im Editor, die ConnectionEditPolicy ist für die Manipulation von Verbindungslinien zwischen Objekten zuständig.

Soll eine Aktion behandelt werden, so wird der in der Hierarchie am weitesten unten liegenden EditPart informiert. Dessen EditPolicies werden überprüft, ob sie den gestellten Request behandeln können. Ist dies nicht der Fall, wird mit dem Vaterobjekt des EditParts fortgefahren. Der Vorgang wird wiederholt bis ein EditPart gefunden werden kann, der den Request behandelt. Behandeln bedeutet dabei, dass die EditPolicy den Request in ein Command-Objekt umwandelt. Findet sich kein zuständiger EditPart, löst die Benutzeraktion keinerlei Reaktionen aus.

### Commands

*Commands* entsprechen dem Entwurfsmuster „*Command*“ [GOF95]. Ein Command kapselt den zur Durchführung einer bestimmten Aufgabe notwendigen Code. Ebenso werden vor oder bei der Abwicklung des Commands die dabei manipulierten Daten gespeichert, so dass der Originalzustand wiederhergestellt werden kann.

Durch Verwendung dieses Entwurfsmusters ergeben sich zwei Vorteile: Einerseits wird der Code vom konkreten Ort der Verwendung getrennt und kann einfach an anderer Stelle wiederverwendet werden. Andererseits wird dadurch das Zurücknehmen der Aktion vereinfacht. Dazu wird das Command-Objekt nach der Durchführung gespeichert. Soll es widerrufen werden, so kann es aufgrund der gespeicherten Daten selbständig den alten Stand von vor der Manipulation wiederherstellen.

# 5 Visueller Editor

Im Rahmen des Projektes wurde die in Kapitel 3 beschriebene Notation prototypisch implementiert. Das Ergebnis ist der *Visual Editor* (VE), der in eine umfangreiche Entwicklungsumgebung für Monaco-Programme integriert ist. Diese „Monaco-IDE“ und insbesondere der visuelle Editor werden in diesem Kapitel beschrieben.

Der visuelle Editor ist Kern dieser Bakkalaureatsarbeit. Der Umgang mit diesem wird daher hier umfassend behandelt. Die restlichen Module werden nur so weit beschrieben, als das für das Arbeiten mit dem visuellen Editor notwendig ist.

## 5.1 Monaco-IDE

Die Monaco-IDE wurde auf Basis von Eclipse RCP und GEF entwickelt (siehe Kapitel 4). Dies erleichterte und beschleunigte die Entwicklung rapide, da auf bereits existierenden Code zurückgegriffen werden konnte. Die IDE integriert alle zur Entwicklung von Monaco-Programmen notwendigen Softwarekomponenten.

Wichtigstes Modul ist der visuelle Editor, in dem vom Programmierer die eigentliche Editierarbeit durchgeführt wird. Unterstützt wird der Editor von einer Vielzahl weiterer Fenster der Benutzerschnittstelle. Ebenso sind einige Plug-ins im Hintergrund tätig und haben keine eigene Benutzerschnittstelle. In Zusammenarbeit ergeben all diese Module eine mächtige Entwicklungsumgebung.

Abbildung 11 zeigt die Monaco-IDE in der Standard-Ansicht. In der Mitte befindet sich der visuelle Editor, der Methoden und Funktionen darstellt. In diesem Bereich können auch mehrere Editoren angezeigt werden, seien es weitere visuelle Editoren oder Texteditoren. Um diesen Editor-Bereich herum sind die weiteren Fenster der Benutzerschnittstellte angeordnet. Es sind dies von rechts oben im Uhrzeigersinn:

- **Outline-View**: Zeigt die an der ausgewählten Codeposition gültigen Deklarationen an.
- **Properties-View**: Zeigt die Eigenschaften des gewählten Codeelementes an und erlaubt die Bearbeitung dieser.
- **Problems-View**: Gibt Fehlermeldungen des Compilers aus.
- **Project-View**: Stellt die Typdeklarationen des aktiven Monaco-Projekts dar.
- **Navigator-View**: Zeigt alle im Projekt eingebundenen Dateien.

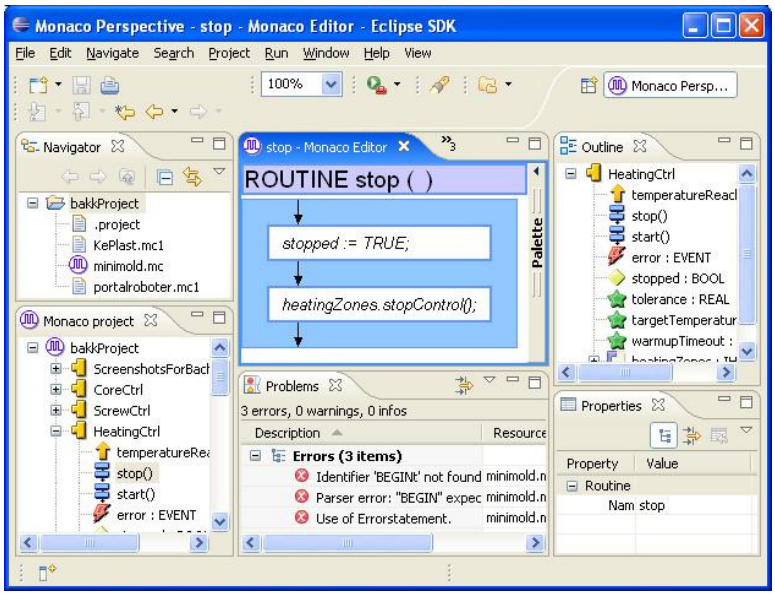


Abbildung 11: Die Monaco-IDE mit den wichtigsten Ansichten

Diese Fenster sind nicht Teil des visuellen Editors. Sie erfüllen aber wichtige Aufgaben, ohne die der visuelle Editor nicht sinnvoll eingesetzt werden kann. Diese sollen daher in den folgenden Abschnitten kurz beschrieben werden.

### 5.1.1 Project-View

Die *Project-View* zeigt die Struktur aller in Eclipse geöffneten Monaco-Projekte an. In Form eines Baumes (einer „Tree-View“) werden zu den Projekten die Komponenten, Typen, Interfaces, Callables (ROUTINE und FUNCTION) und weitere Deklarationen angezeigt.

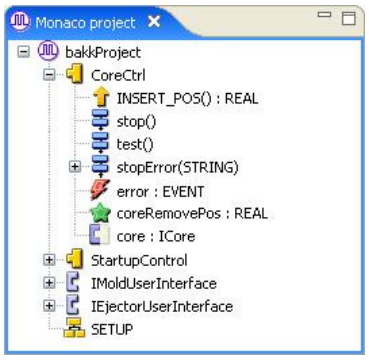


Abbildung 12: Die Project-View, eine Komponente ist ausgeklappt

Durch Doppelklick auf die Kindelemente werden diese in einem neuen Editor-Fenster dargestellt. In der derzeitigen Ausbaustufe ist dies bei Routinen und Funktionen sowie beim Setup möglich.

Auch können in der Project-View neue Elemente angelegt werden. Wird mit der rechten Maustaste auf ein Element geklickt, so öffnet sich ein Kontextmenu mit einer Liste aller an dieser Stelle möglichen Einfügeoperationen. Wird etwa auf eine Komponente geklickt können unter anderem eine neue Routine, eine Funktion oder eine lokale Variable angelegt werden.

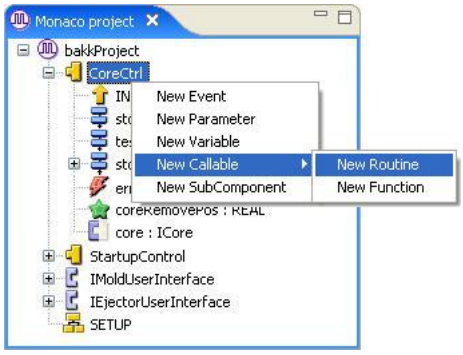


Abbildung 13: Project-View, einer Komponente wird eine Routine hinzugefügt

Die jeweils möglichen Operationen sind nicht statisch im Editor programmiert. Sie werden vom CodeInfo-Paket per *Reflection* aus dem CodeDOM ausgelesen. Dies soll eine leichte Erweiterbarkeit des Editors garantieren. So kann etwa eine neue Art von Statement in die Sprache eingeführt werden, ohne dadurch notwendigerweise den visuellen Editor verändern zu müssen. Durch die Auswertung des CodeDOM über Reflection kann der visuelle Editor das neue Statement erzeugen.

### 5.1.2 Outline-View

Die *Outline-View* stellt die Menge der im aktuell angezeigten Codestück gültigen und sichtbaren Deklarationen dar. Es werden also alle Elemente dargestellt, auf die das Programm innerhalb dieses Programmelementes zugreifen kann. Es sind dies die Parameter und lokalen Variablen einer Routine, die Member-Variablen der übergeordneten Komponente sowie deren aufrufbare Routinen und Funktionen oder globale Variablen.

Mittels Drag & Drop lassen sich Elemente aus der Outline in den Editor übertragen. So können etwa Aufrufe einer Routine sehr leicht erstellt werden.

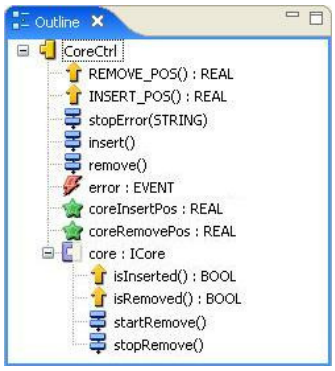


Abbildung 14: Outline-View mit Funktionen, Methoden, Events, Parametern und einer Subkomponente

### 5.1.3 Properties-View

Die *Properties-View* zeigt Eigenschaften des im visuellen Editor aktuell selektierten Elementes an. So können etwa der Sourcecode, der Darstellungszustand (aus- oder eingeklappt) oder eine verknüpfte Bedingung (bei IF oder ON-Handler) angezeigt werden.

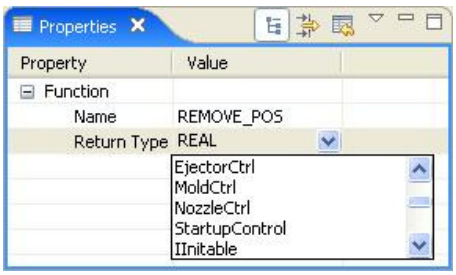


Abbildung 15: Properties-View bei Auswahl einer Funktion, der Rückgabetyp wird bearbeitet

Dieser Teil der IDE ist noch nicht vollständig implementiert, es können nur einige wenige Eigenschaften tatsächlich dargestellt werden. Auch die Informationen über die verfügbaren Properties sollen per Reflection aus dem CodeDOM ausgelesen werden. Dabei sind jedoch rein intern verwendete Properties auszuscheiden und nicht in der Properties-View anzuzeigen. Derzeit ist diese Unterscheidung aber noch nicht möglich, es fehlt ein eindeutiges Unterscheidungskriterium. Eine mögliche zukünftige Lösung des Problems wäre, *Annotations* zur Auszeichnung von im Editor sichtbaren Properties zu verwenden.

### 5.1.4 Problems-View

In der *Problems-View* werden Fehler und Warnungen des Parsers ausgegeben. Erkennt dieser tatsächliche oder mögliche Probleme während des Parsens, so führt er den Parse-Vorgang nach Möglichkeit trotzdem zu Ende. Der Benutzer wird auf alle dabei erkannten Probleme in der Problems-View hingewiesen. Bei Doppelklick auf ein derartiges Problem gelangt man zu einem Texteditor, in dem das betroffene Codestück ebenfalls durch einen Marker hervorgehoben ist. Wird der Fehler behoben, dann wird auch der Problem-Marker wieder entfernt.

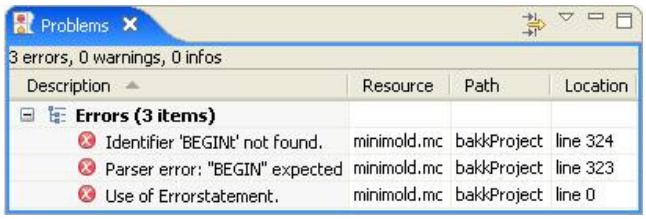


Abbildung 16: Die Problems-View zeigt einen Tippfehler an

### 5.1.5 Texteditor

Ebenfalls Teil der IDE ist ein auf Monaco angepasster Texteditor. Dieser bietet unter anderem Syntaxhervorhebungen, indem Schlüsselwörter und andere spezielle Codeteile besonders formatiert oder farblich markiert werden. Für die Zukunft ist auch eine Autovervollständigungs-Funktion geplant. Diese wird während des Tippens Vorschläge unterbreiten, welche Eingaben an der aktuellen Stelle Sinn ergeben.

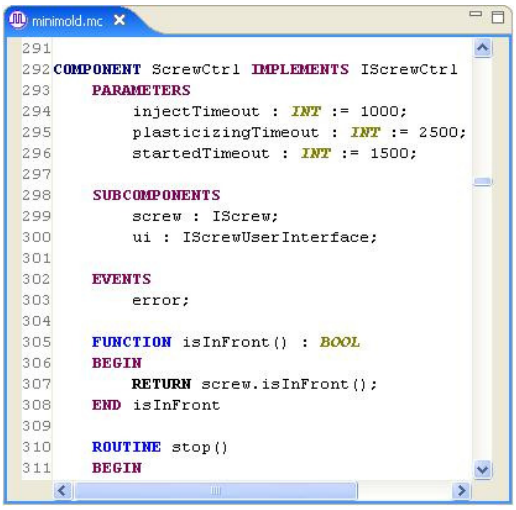


Abbildung 17: Texteditor mit Syntaxhervorhebung

## 5.2 Visueller Editor

Der visuelle Editor (im Folgenden *Monaco VE*) nutzt die Daten, die der Monaco-Compiler in Form eines abstrakten Syntaxbaumes zur Verfügung stellt. Dieser Baum wird im Rahmen dieses Projektes als *CodeDOM* bezeichnet. Er wird vom Parser aus dem Sourcecode eines Monaco-Programms in objektorientierter Form als baumähnlicher Graph generiert. Diesen Graphen kann der Monaco VE darstellen. Dem Benutzer ist es aber auch möglich, mit dem visuellen Editor den CodeDOM zu manipulieren und damit das Programm zu verändern. Er programmiert also Monaco-Sourcecode auf graphische Weise. Der *Monaco-Writer* kann anschließend den veränderten CodeDOM wieder in für Menschen lesbaren Monaco-Quelltext transformieren.

Der visuelle Editor nutzt das *Graphical Editing Framework* (GEF) zur Datenanzeige (siehe Kapitel 4 „Verwendete Technologien“). Im Hauptfenster des Editors wird jeweils eine Routine oder Funktion angezeigt (zur Notation der Darstellung, siehe Kapitel 3 „Visuelle Notation von Monaco“).

### 5.2.1 Überblick

Im Monaco VE werden die Unterprogramme, wie in Kapitel 3 bereits erläutert, in der graphischen Notation angezeigt. Neben den in Struktogramm-ähnlicher Form

angeordneten Statements einer Methode oder Funktion werden auch deren Parameter und lokale Variablen dargestellt (siehe Abbildung 18).

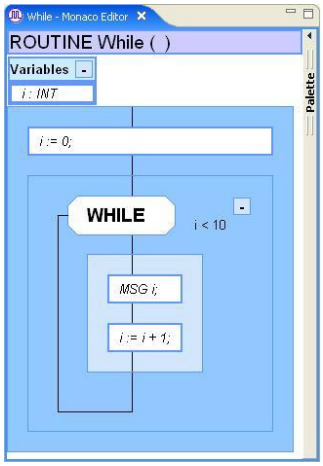


Abbildung 18: Eine Routine im visuellen Editor. i ist eine lokale Variable, darunter sind die Anweisungen dargestellt. Auf der rechten Seite ist die Palette angebracht, hier eingeklappt.

Ein weiteres wichtiges Element des Hauptfensters ist die Palette. Auf dieser werden die im aktuellen Zustand sinnvollen Werkzeuge (*Tools*) angezeigt. Das wichtigste davon ist das *Selection-Tool* mit dem die meisten Manipulationen im visuellen Editor durchgeführt werden können. Es können damit Objekte markiert, per Drag & Drop verschoben, expandiert und zusammengeklappt werden. Weitere Tools stehen etwa für das Anlegen neuer Objekte zur Verfügung. Intensiv genutzt wird die Palette im Setup. Dabei werden alle zur Verfügung stehenden Komponenten angezeigt und können so schnell im Graphen eingefügt werden. Mit dem *Connection-Creation-Tools* können sie anschließend in die Hierarchie eingebunden werden.

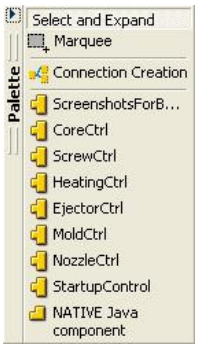


Abbildung 19: Palette, hier für das SETUP

Wichtige Manipulationen der Daten können über ein Kontextmenu erledigt werden. Dazu klickt der Benutzer mit der rechten Maustaste auf das zu manipulierende Objekt.

Dadurch erscheint ein Kontextmenu, in dem etwa neue Kind-Objekte des angeklickten Objektes erzeugt werden können.

Das Hauptfenster kann nicht nur den Inhalt von Methoden und Funktionen anzeigen, sondern auch das sogenannte *Setup*. Damit ist die Konfiguration und Initialisierung der Maschine gemeint. Es werden dabei die einzelnen Monaco-Komponenten miteinander verbunden, Initialisierungswerte gesetzt und Parameter angepasst. Dieser Teil ist ebenfalls in den visuellen Editor eingebunden, wurde jedoch nicht vom Autor dieser Arbeit entwickelt und soll daher hier nicht näher beschrieben werden.

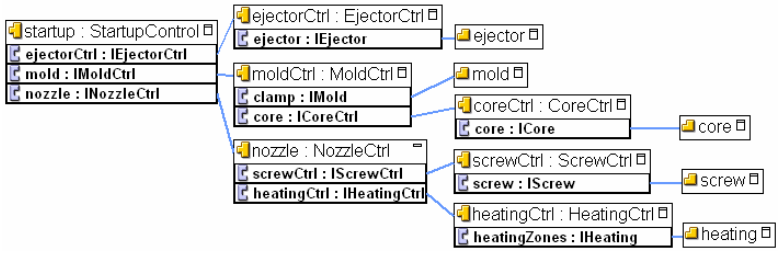


Abbildung 20: Darstellung des SETUP im Monaco VE

### 5.2.2 Editieren im visuellen Editor

Das Bearbeiten von Code erfolgt primär im visuellen Editor. Dort kann der Sourcecode auf umfassende Weise manipuliert werden. Fast alle im Texteditor auf den Sourcecode anwendbaren Operationen können auch im visuellen Editor durchgeführt werden.

Unterstützt wird dieser Prozess durch die in den weiteren Ansichten der IDE möglichen Operationen. So werden Eigenschaften des selektierten Objektes in der Properties-View übersichtlich gruppiert und können dort auch direkt verändert werden. In der Project-View ist es möglich, neue Deklarationen anzulegen.

#### Einfügen von Anweisungen

Eine grundlegende Funktion des visuellen Editors ist es, neue Anweisungen an das Ende bestehender Blöcke anzufügen. Dazu kann der Benutzer mit einem Rechtsklick in den freien Bereich eines Blocks ein Kontextmenu öffnen. In diesem werden alle in diesem Block möglichen Einfügeoperationen angezeigt. Auf gleiche Weise können auch Kontrollanweisungen wie PARALLEL oder IF um Kind-Objekte erweitert werden.

Welche Operationen an einer Position möglich sind, ist nicht statisch festgelegt. Diese Information wird generisch aus dem CodeDOM ausgelesen. Der Editor wird dadurch flexibler und passt sich automatisch an Änderungen im CodeDOM an.

An den meisten Stellen lassen sich so einfache Statements, Kontrollstrukturen oder Blöcke anlegen. Innerhalb von Blöcken können ON-Handler hinzugefügt werden, PARALLEL-Statements lassen sich um neue parallele Anweisungsfolgen erweitern, IF-Statements um neue IfThen-Abschnitte.


Abbildung 21: Kontextmenu, um Komponenten zu bearbeiten

Die so eingefügten Statements sind vorerst leer. Wird ein Block, ein PARALLEL- oder ein LOOP-Statement neu eingefügt, so ergeben diese lediglich Hüllen für noch einzufügende Funktionalität. Erst durch das nachfolgende Einfügen der Statements ergibt dies gültige Konstrukte. Besonders deutlich wird das beim Einfügen eines neuen IF-Statements. Dieses alleine trägt keinerlei Funktion in sich. Erst durch das Einfügen von IfThen-Abschnitten in das IF-Statement ergibt sich die gewohnte Semantik einer bedingten Anweisung.

### Copy & Paste

Der visuelle Editor erlaubt Copy & Paste. Jedes Statement kann markiert und mittels Tastenkombination oder Menueintrag kopiert werden. An anderer Stelle lässt es sich dann auf gleiche Weise einfügen. Dies funktioniert sowohl innerhalb eines visuellen Editors als auch zwischen mehreren Editoren. Es ist nicht einmal notwendig, dass die Quelle und das Ziel der visuelle Editor ist. So kann auch Sourcecode im visuellen Editor kopiert und in einen Texteditor eingefügt werden. Auf gleiche Weise kann auch Sourcecode aus Drittprogrammen kopiert und in den visuellen Editor eingefügt werden.

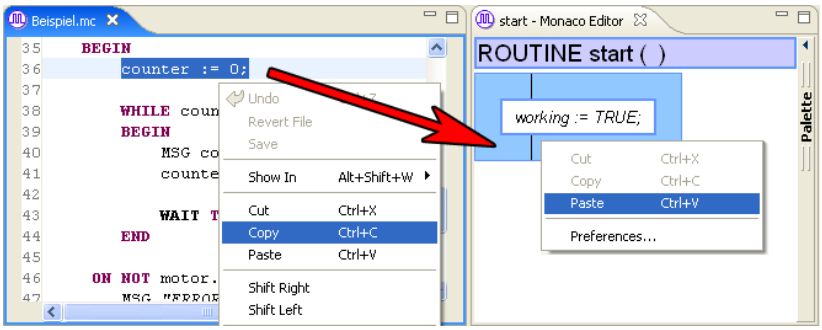


Abbildung 22: Copy & Paste aus dem Texteditor in den visuellen Editor

### Drag & Drop

*Drag & Drop* bezeichnet die Technik, Elemente bei gedrückt gehaltener Maustaste am Bildschirm zu bewegen. Dabei kann das Objekt entweder verschoben werden oder eine Kopie des Objektes am Ziel neu angelegt werden. Diese zwei prinzipiell unterschiedlichen Möglichkeiten werden auch von der IDE unterstützt.

Werden Objekte innerhalb des Editors gezogen, so handelt es sich um ein Verschieben. Man kann auf diese Weise bereits bestehende Statements umgruppieren, also etwa in der Reihenfolge verändern oder von einem IF- in den zugehörigen ELSE-Block verschieben. Würde man dies im textuellen Sourcecode machen, wären dazu etliche Mausklicks oder Tastenanschläge notwendig. Im visuellen Editor reicht dafür ein Klick samt Mausbewegung an das Ziel.

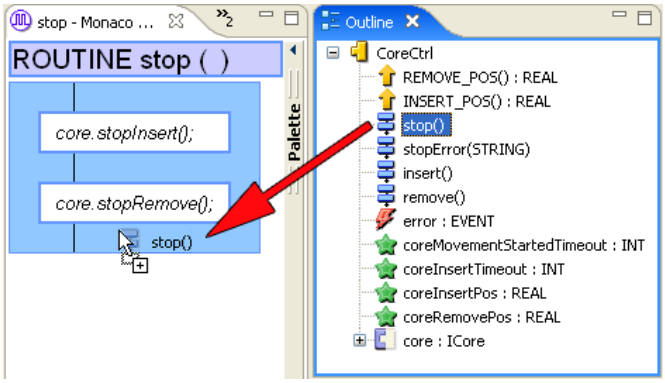


Abbildung 23: Drag & Drop aus der Outline-View, es wird ein Routine-Call erzeugt.

Eine weitere Anwendung von Drag & Drop ergibt sich in Verbindung mit der Outline-View. Dort wird der Scope des aktuell geöffneten Codeteiles angezeigt. Dieser umfasst beispielsweise alle Variablen, auf die zugegriffen werden kann, und alle aufrufbaren Methoden. All diese lassen sich per Drag & Drop aus der Outline in den visuellen

Editor ziehen. Routinenaufrufe lassen sich so bequem in einer einzigen Operation erstellen.

Prinzipiell lässt sich die letztgenannte Technik auch auf Variablenzugriffe oder Funktionsauswertungen anwenden. In der aktuellen Ausbaustufe kann der visuelle Editor allerdings nur vollständige Statements verarbeiten. Ein Herunterbrechen von Statements auf Ausdrücke oder noch weiter ist für eine zukünftige Ausbaustufe vorgesehen. Dann ist auch das Einfügen eines Funktionsaufrufes oder eines Variablenzugriffes per Drag & Drop möglich. Derzeit führt eine derartige Operation zu keinem sinnvollen Statement im visuellen Editor.

### DirectEdit

Eine schnelle und bequeme Methode der Bearbeitung bietet sich durch das Verknüpfen von textuellem und visuellem Editieren. Dazu wird etwa durch Doppelklick auf ein Objekt ein Textfeld geöffnet, in dem der Quelltext des Objektes editiert werden kann. Nach Abschluss der Editier-Operation wird nur der veränderte Sourcecode neu übersetzt und das Objekt in aktualisierter Form dargestellt. Ergibt der eingegebene Sourcecode kein gültiges Statement, so wird stattdessen ein ErrorStatement in auffälligem Rot als Fehlermarkierung dargestellt.

Dies wird durch GEF unmittelbar unterstützt. Es werden hierzu spezielle EditPolicies, so genannte „DirectEdit“-Policies verwendet. Im Editor kommen diese unter anderem zum Bearbeiten von booleschen Bedingungen zum Einsatz, etwa bei IF-Statements, in WHILE-Schleifen oder in ON-Handlern.


Abbildung 24: Bearbeiten einer Bedingung

# 6 Beispielprogramm

Dieses Kapitel soll die Funktionen der Monaco-IDE und insbesondere des visuellen Editors an einem praktischen Beispiel demonstrieren. Dazu wird ein kurzes Programm in der Monaco IDE entwickelt. Es wird dabei wo immer möglich auf die Funktionalität des visuellen Editors oder anderer IDE-Elemente zurück gegriffen.

Folgende Operationen sollen dabei demonstriert werden:

- Anlegen eines neuen Projektes mit einer Monaco-Datei
- Verwenden von Project- und Properties-View
- Einfügen des Setup
- Anlegen von Komponenten und Interfaces
- Hinzufügen von Variablen, Routinen und Funktionen
- Bearbeiten des Sourcecodes der Unterprogramme
- Drag & Drop sowie Copy & Paste

## 6.1 Zu entwickelndes Programm

Als Beispielszenario verwenden wir eine sehr einfache Maschine mit zwei Komponenten. Die Steuerungskomponente Main hat genau eine Subkomponente Motor, die das Interface IMotor erfüllt. IMotor enthält die Routinen start und stop sowie die Funktion isWorking. Main enthält lediglich die Routine main, der die Steuerung der Maschine obliegt. Aufgabe der Maschine ist es, den Motor zu aktivieren, zehn Zählschritte lang laufen zu lassen und anschließend wieder aus zu schalten. Dabei möglicherweise auftretende Fehler sollen korrekt behandelt werden.

Folgender Quelltext implementiert den beschriebenen Ablauf. In weiterer Folge versuchen wir, mit dem visuellen Editor diesen Code zu entwickeln.

```
INTERFACE IMotor
  ROUTINE start();
  ROUTINE stop();
  FUNCTION isWorking() : BOOL;
END IMotor

COMPONENT Motor IMPLEMENTS IMotor
  VARS
    working : BOOL;

  ROUTINE start()
  BEGIN
    working := TRUE;
  END start

  ROUTINE stop()
  BEGIN
    working := FALSE;
  END stop

  FUNCTION isWorking() : BOOL
  BEGIN
    RETURN working;
  END isWorking
END Motor

COMPONENT Main
  SUBCOMPONENTS
    motor : IMotor;

  VARS
    counter : INT;

  ROUTINE main()
  BEGIN
    counter := 0;
    SELF.motor.start();
    WHILE counter < 10
    BEGIN
      MSG counter;
      counter := counter + 1;
      WAIT 1000;
    END
    SELF.motor.stop();

  ON TIMEOUT(1000)
    MSG "ERROR: Timeout";
    SELF.motor.stop();

  ON NOT motor.isWorking()
    MSG "ERROR: Unexpected stop of motor";
    SELF.motor.stop();

  END main

END Main

SETUP
BEGIN
        //Code des Setups
END SETUP
```

## 6.2 Vorgehensweise

Nach dem Start des Editors wird die vorgegebene Monaco-Ansicht angezeigt. Bei dieser sind die wichtigsten Fenster der Monaco-IDE rund um den visuellen Editor angeordnet.

### 6.2.1 Anlegen des Projektes und einer Monaco-Datei

Zu Beginn müssen wir ein Projekt anlegen. Dazu wählen wir im Hauptmenü den Punkt „File – New – Project“. Mit Hilfe des „New Project“-Dialoges legen wir ein neues, generisches Projekt an. Wir wählen also aus der Kategorie „General“ den Eintrag „Project“ (Abbildung 25). Als Titel vergeben wir beispielsweise „Demonstration“ (Abbildung 26). Dadurch wird ein neues, leeres Projekt angelegt und in der Navigator-View angezeigt. Dieses enthält lediglich eine Datei „.project“, in der Eclipse die wichtigsten Projekteinstellungen speichert (Abbildung 27).

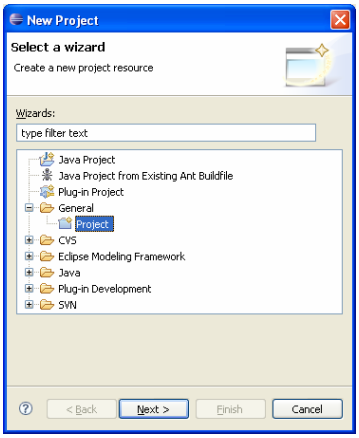


Abbildung 25: Neues allgemeines Projekt anlegen

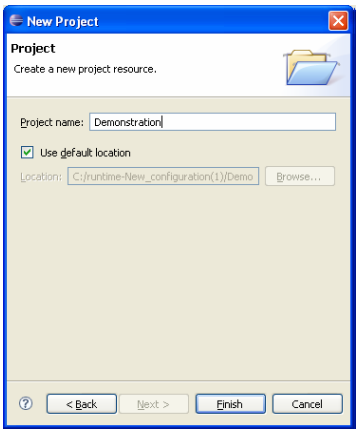


Abbildung 26: Titel des Projektes eingeben

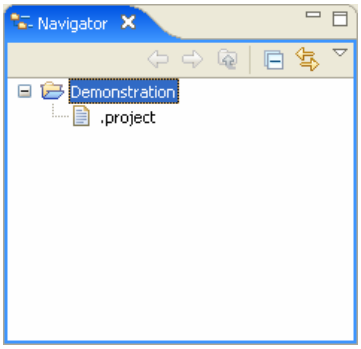


Abbildung 27: Ein leeres Projekt wurde angelegt

Nun legen wir eine Monaco-Quellcodedatei an. Dazu wählen wir abermals den Menupunkt „File – New“ aus, diesmal aber den Punkt „Other“. In der Liste wählen wir „Monaco – Monaco Editor File“ aus. Auf der nächsten Seite geben wir einen Dateinamen (etwa „Beispiel.mc“) ein (Abbildung 28). Durch Klick auf „Finish“ bestätigen wir unsere Eingabe. Die Datei wird nun von Eclipse angelegt und in der Navigator-View angezeigt (Abbildung 29). Gleichzeitig erscheint ein Eintrag des noch leeren Projektes in der Project-View (Abbildung 30).

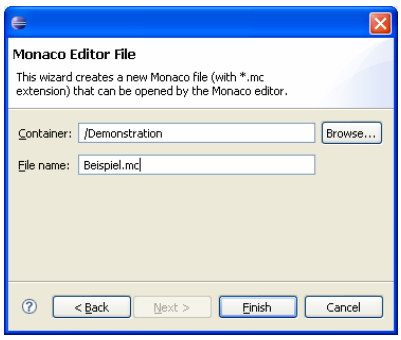


Abbildung 28: Neue Monaco-Sourcecode-Datei anlegen

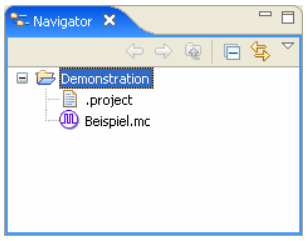


Abbildung 29: Projekt mit Monaco-Sourcecode-Datei

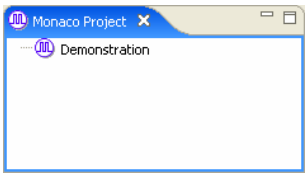


Abbildung 30: Project-View mit leerem Projekt

### 6.2.2 Erzeugen der Komponenten und des Interfaces

Durch einen Rechtsklick mit der Maus auf das Projekt in der Project-View öffnet sich ein Kontextmenu. Bei Auswahl von „Setup“ wird das selbige angelegt (Abbildung 31). Auf die gleiche Weise erzeugen wird wir ein Interface und zwei Komponenten.

Das Interface und die beiden Komponenten werden mit Standardnamen generiert. Falls notwendig werden an den Namen fortlaufenden Nummern angehängt. Durch oder Selektieren des Objekts und einen Tastendruck auf F2 können wir den Namen der Datensätze bearbeiten (Abbildung 32). Das Interface benennen wir „IMotor“, die Komponenten „Main“ und „Motor“. Das Setup belassen wir ohne Änderung.

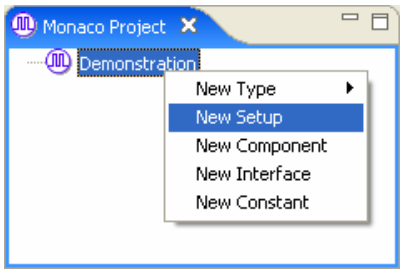


Abbildung 31: Einfügen des SETUP-Abschnitts

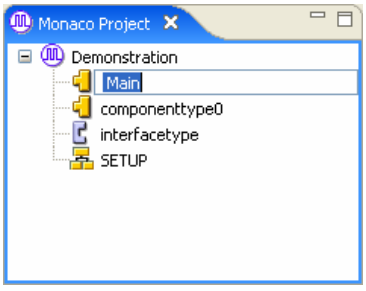


Abbildung 32: Umbenennen einer Deklaration

### 6.2.3 Anlegen von Variablen, Routinen und Funktionen

#### Variablen

In den Komponenten werden wir jeweils eine Variable benötigen. Auf die bereits bekannte Weise fügen wir diese Objekte den Komponenten hinzu. Der Typ der Variablen wird standardmäßig auf ERROR gesetzt. Um dies zu korrigieren, selektieren wir eine der Variablen. Dabei ändert sich auch die Properties-View. Dort werden zu jeder Variablen deren Name und der Typ angezeigt. Wir ändern nun beides. Die Variable in Motor ändern wir auf den Namen working und setzen den Typ auf BOOL (Abbildung 33). Anschließend bearbeiten wir die Variable in Main. Wir ändern deren Namen auf counter und stellen den Typ auf INT.

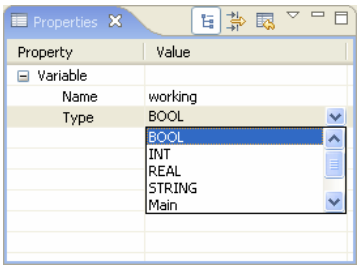


Abbildung 33: Ändern des Datentyps einer Variablen in der Properties-View

#### Routinen und Funktionen

Die wichtigste Routine des Programms soll sich in der Komponente Main befinden. Wir legen dort auf die bekannte Weise eine Routine an. Im Menupunkt „Callables“ haben wir dazu die Wahl zwischen ROUTINE und FUNCTION. An dieser Stelle wählen wir ROUTINE. Das angelegte Unterprogramm nennen wir in bereits bewährter Weise auf main um.

Als nächstes legen wir die benötigten Programmelemente im Interface IMotor an. Wir benötigen dort zwei Routinen sowie eine Funktion. Die Routinen nennen wir start und stop, die Funktion erhält den Namen isWorking. In der Properties-View ändern wir den Rückgabetyp (*Return Type*) der Funktion auf BOOL.

Die Komponente Motor soll das Interface IMotor erfüllen. Leider ist es derzeit noch nicht möglich, dies zu automatisieren. Wir müssen nun also die gerade eben für IMotor durchgeführten Schritte nochmals für Motor wiederholen.

Damit sind nun alle Deklarationen korrekt angelegt (Abbildung 34). In den nächsten Schritten werden wir dieses Gerüst nun mit Anweisungen füllen.

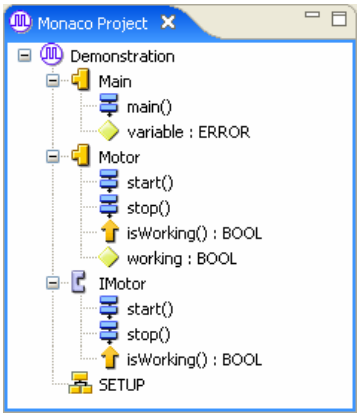


Abbildung 34: Project-View mit allen benötigten Deklarationen

### 6.2.4 Anweisungen in Motor

Um die Routinen und Funktionen der Komponente Motor mit Anweisungen zu füllen, wollen wir nun erstmals den visuellen Editor verwenden.

#### Routine Motor.start

Um den visuellen Editor zu öffnen klicken wir mit der linken Maustaste in der Project-View doppelt auf die Methode start. Darauf hin öffnet sich der visuelle Editor im zentralen Editor-Bereich und stellt die noch leere Methode dar. Wir sehen nur ein kleines blaues Rechteck, den Block der Methode (Abbildung 35).

Da unser Beispielprogramm keine echte Maschine steuert, sondern eine solche lediglich simuliert, wollen wir in der Routine start lediglich die Variable working auf TRUE setzen. Dazu klicken wir mit der rechten Maustaste in den Bereich des Blockes. Wiederum erscheint ein Kontextmenu, das uns das Anlegen von Anweisungen erlaubt. Variablen-Zuweisungen können im aktuellen Entwicklungsstand leider nicht automatisch generiert werden, wir wählen ersatzweise ein DebugStatement aus („Add Statement – New DebugStatement“, siehe Abbildung 35). Anschließend markieren wir die Anweisung und drücken auf der Tastatur die Taste F2. Dies erlaubt uns, den Sourcecode der Anweisung direkt im visuellen Editor zu bearbeiten (Abbildung 36). Hier im Visuellen Editor ist es auch möglich, dies durch einen Doppelklick auf das entsprechende Objekt durchzuführen. Wir wollen der Variablen working den Wert TRUE zuweisen, also geben wir ein:

```
working := TRUE;
```

Durch einen Mausklick an eine andere Stelle des Editors wird der Vorgang abgeschlossen. Sofern unsere Eingabe korrekt war, wird die gewünschte Anweisung generiert und angezeigt.

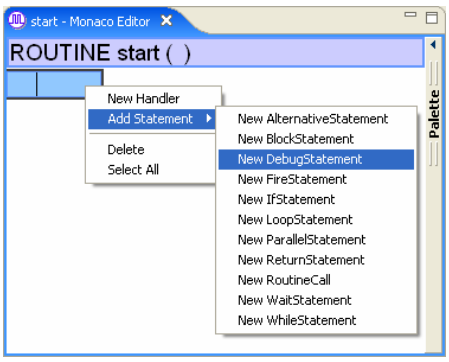


Abbildung 35: Einfügen eines DebugStatement in eine leere Routine

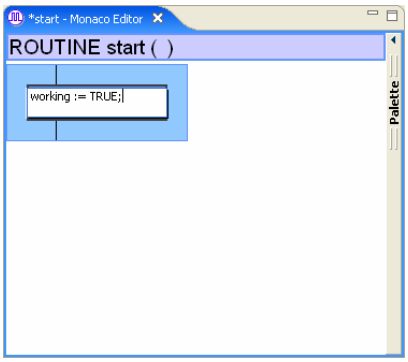


Abbildung 36: Bearbeiten des Statements in eine Variablen-Zuweisung

Wir wählen nun den Menupunkt „File – Save“ an, um die Datei zu speichern. Abgekürzt kann man dies auch durch einen Klick auf das Speichern-Symbol (Diskette) in der Symbolleiste oder die Tastenkombination STRG+S erledigen. Da wir die Routine start nicht mehr weiter bearbeiten möchten, können wir den Editor nun schließen.

### Routine Motor.stop

Auf die gleiche Weise wie oben ausgeführt erstellen wir eine Variablen-Zuweisung in der Routine stop. Wieder öffnen wir den Editor durch Doppelklick. Mit Hilfe des Kontextmenus legen wir ein Statement an und ändern dieses in eine Variablen-Zuweisung. Diesmal wollen wir die Maschine anhalten. Wir geben als Zuweisung also ein:

```
working := FALSE;
```

Anschließend können wir wieder speichern und den Editor schließen.

### Funktion Motor.isWorking

Den Vorgang wiederholen wir nochmals für die Funktion isWorking. Einzige Aufgabe dieser ist es, den Wert der durch die Komponente gekapselten Variable working öffentlich sichtbar zu machen. Da sich diese Funktion im Interface befindet, ist sie nach außen hin sichtbar. Die Variable hingegen ist dies nicht, auf sie kann nur von Code

innerhalb der Komponente zugegriffen werden. Daher ist diese Zugriffsfunktion notwendig.

Als Sourcecode wird in der Routine ebenfalls nur eine Anweisung benötigt:

```
RETURN working;
```

Dadurch wird der Wert der Variablen working von der Routine zurück gegeben.

### 6.2.5 Anweisungen in Main.main

Wir kommen nun zur komplexeren Methode dieses Beispiels. In Main.main soll der Kern der Steuerung implementiert werden. Ihre Aufgabe ist es, den Motor zu starten und in einer Schleife zehn Durchgänge lang laufen zu lassen. In jedem Durchlauf wird eine Sekunde lang pausiert. Nach Abschluss dieser Tätigkeit wird der Motor wieder abgeschaltet. Während der gesamten Operation soll der Motor überwacht und eventuell auftretende Fehler korrekt behandelt werden.

#### Motor als Subkomponente von Main

Hier müssen wir nun leider einmal auf den Texteditor zurückgreifen. Es ist derzeit noch nicht möglich, mittels der IDE zu spezifizieren, welches Interface eine Komponente erfüllen soll. Daher klicken wir im Navigator links oben mit der rechten Maustaste auf unsere Monaco-Datei (Beispiel.mc) und wählen aus dem Menu „Open with – Monaco Text Editor“ aus. Der Texteditor wird geöffnet und zeigt das Ergebnis unserer bisherigen Arbeit an. Wir suchen uns die Definition der Komponente Motor und fügen an diese eine IMPLEMENTS Klausel an:

```
COMPONENT Motor IMPLEMENTS IMotor
```

Anschließend speichern wir unsere Änderungen, worauf hin die Quelltextdatei automatisch neu kompiliert wird.

Nun müssen wir noch festlegen, dass Motor eine Subkomponente von Main ist. Dazu fügen wir in der Project-View zu Main eine SUBCOMPONENT hinzu. In der Properties-View ändern wir den Namen der Subkomponente auf motor und den Datentyp auf IMotor.

### Anweisungen

Nun können wir die Routine Main.main im visuellen Editor öffnen. In der gewohnten Weise legen wir zuerst eine Zuweisung an und setzen damit den Wert der Variablen counter auf 0.

Als nächstes fügen wir eine Schleife ein. Wir wählen dazu aus dem Kontextmenu den Eintrag „Add Statement – New WhileStatement“, um eine WHILE-Schleife anzulegen. Diese Schleife hat zu Beginn keinen Rumpf und als Abbruchbedingung den Ausdruck FALSE, wird also niemals durchlaufen.

Zuerst ändern wir die Abbruchbedingung. Wie auch bei anderen Elementen bereits beschrieben lässt sich dies per Druck auf die Taste F2, während die Bedingung selektiert ist, oder einem Doppelklick auf die Bedingung durchführen. Wir ändern diese auf:

```
counter < 10
```

Damit wird die Schleife zehn Mal durchlaufen, vom 0. bis zum 9. Mal.

Anschließend klicken wir mit der rechten Maustaste auf den freien Bereich im rechten, unteren Bereich der WHILE-Schleife. Im bekannten Kontextmenu wählen wir „SetStatement – New BlockStatement“. Damit wird als Rumpf der Schleife ein Block festgelegt (Abbildung 37). In diesen fügen wir nun drei Anweisungen ein. Mit einem DebugStatement wollen wir vom Programm in jedem Durchlauf den Wert der Variablen counter ausgeben lassen. Das zweite Statement ändern wird in eine Zuweisung, in der wir counter um eins erhöhen. Durch ein WAIT-Statement führen wir noch eine erzwungene Pause in den Ablauf ein. Dazu geben wir, auf die drei Statements verteilt, folgenden Sourcecode ein:

```
MSG counter;
counter := counter + 1;
WAIT 1000;
```

Der Ablauf der Schleife ist nun fertig (Abbildung 38). Es fehlen jedoch noch die Anweisungen zur Ansteuerung des Motors.

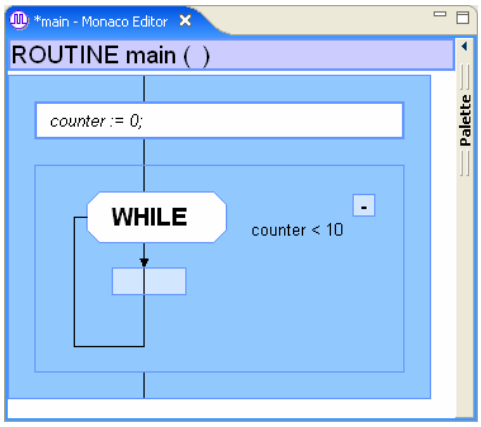


Abbildung 37: WHILE-Schleife mit Abbruchbedingung und leerem Block

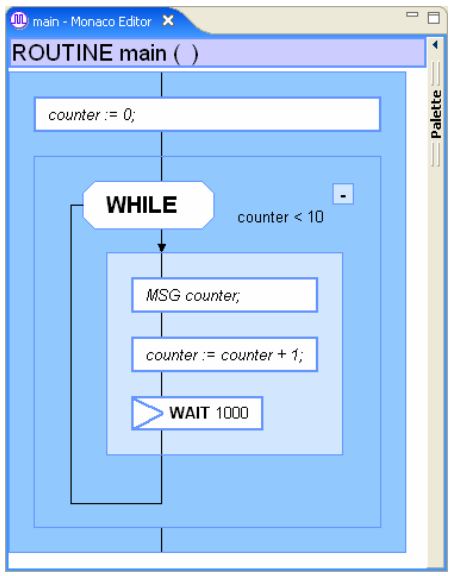


Abbildung 38: Fertig programmierte Schleife

### Motoransteuerung

Der Motor muss vor Beginn der Schleife gestartet werden. Vor dem Ende der Routine main soll der Motor auf jeden Fall wieder gestoppt sein.

Die Komponente Motor, konkret deren Interface IMotor, haben wir bereits als Subkomponente mit dem Variablennamen motor eingebunden. Wir könnten nun also wie gewohnt Anweisungen einfügen und die Routinen des Interface aufrufen. Das Starten des Motors wäre etwa mit dem Sourcecode motor.start() möglich.

Es gibt jedoch noch eine einfachere, schnellere Methode dazu. In der Outline-View, rechts oben am Bildschirm, werden die aktuell sichtbaren Symbole angezeigt, etwa lokale Variablen und Unterprogramme aber auch Subkomponenten. Wir sehen dort die aktuell bearbeitete Methode main, die Variable counter und die Subkomponente motor. Wenn wir die Subkomponente aufklappen, dann sehen wir ihre drei Unterprogramme. Diese können per Drag & Drop in das Editor-Fenster gezogen werden und dort an der gewünschten Stelle eingefügt werden. Auch innerhalb des Editors können die Statements so verschoben werden, etwa um sie in der Reihenfolge zu verändern oder in die Schleife hinein zu verschieben. Auf diese Weise ziehen wir einen Methodenaufruf von start vor den Beginn der Schleife sowie einen Aufruf der Methode stop nach das Ende der Schleife.

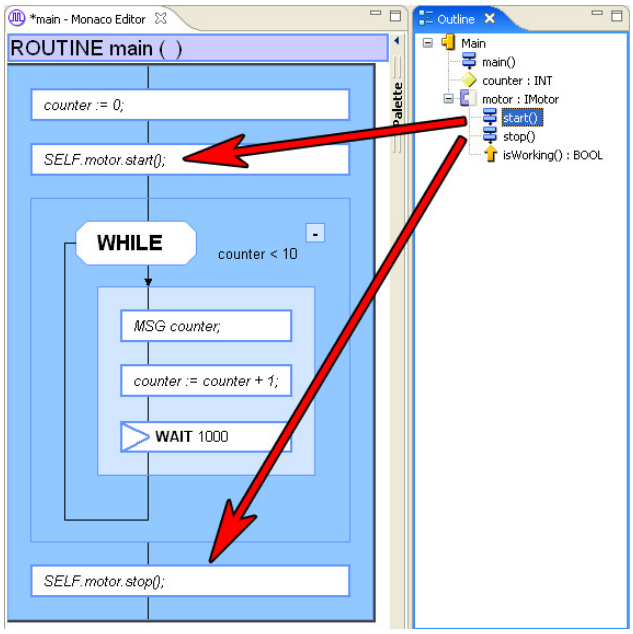


Abbildung 39: Drag & Drop von Calls aus der Outline-View in den Editor

### Fehlerbehandlung

Nun fehlen uns lediglich die Codestücke zur Fehlerbehandlung. Wir wollen in Main.main zwei Fehler überprüfen. Eine mögliche Fehlersituation wäre, dass der Motor unerwarteterweise zu arbeiten aufhört. Dies können wir durch Abfragen der Funktion motor.isWorking() feststellen. Ein weiterer denkbarer Fehlerfall wäre, dass der Motor die von ihm geforderte Aktion nicht schnell genug durchführt. In unserem einfachen Beispiel kann dies nicht geschehen. Würde die Komponente Motor aber tatsächlich die Hardware ansteuern, wäre eine Zeitverzögerung durchaus denkbar. Wir wollen den Fehler an dieser Stelle also überprüfen.

Wir fügen den Code zur Fehlerbehandlung an den Block der Routine main an. Um einen ON-Handler einzufügen, öffnen wir das Kontextmenu für den Block und wählen „New Handler“. Das Symbol eines neuen Handlers wird rechts oben an den Block angefügt. Wenn wir auf dieses Symbol klicken, wird der Handler aufgeklappt und sein Inhalt angezeigt.

Den ersten Fehler prüfen wir, indem wir die Bedingung des Handlers auf folgenden Code setzen:

```
NOT motor.isWorking()
```

Der Motor wäre dann nicht aktiviert. Dies ist als Fehlerfall zu betrachten, da der Motor während des gesamten Ablaufs der WHILE-Schleife eingeschaltet sein sollte. Tritt der Fehler ein, dann soll zur Sicherheit der Motor gestoppt werden sowie die Methode und

damit das Programm beendet werden. Wir fügen in den Handler also noch folgenden Code ein:

```
MSG „ERROR: Unexpected stop of motor“;
motor.stop();
```

Die Programmausführung wird danach mit der Anweisung fortgesetzt, die auf den Block folgt, an dem der ON-Handler hängt. Da dies die Hauptmethode unseres Beispielprogramms ist, bedeutet dies, dass das Programm im Fehlerfalle automatisch beendet wird. Es sind also keine weiteren Anweisungen zur Programmbeendigung notwendig.

Der zweite Fehler ist auf ähnliche Weise zu beheben. Als Bedingung nutzen wir die in Monaco vordefinierte Funktion TIMEOUT:

```
TIMEOUT(1000)
```

Als Reaktion auf den Fehler geben wir wiederum eine Nachricht aus und stoppen den Motor:

```
MSG „ERROR: Timeout“;
motor.stop();
```

Der Code der Routine ist damit komplett (Abbildung 40).

Wir können die bearbeitete Datei nun speichern. Betrachten wir den generierten Quelltext nun wieder im Texteditor, so müsste dieser in etwa dem eingangs beschriebenen entsprechen.

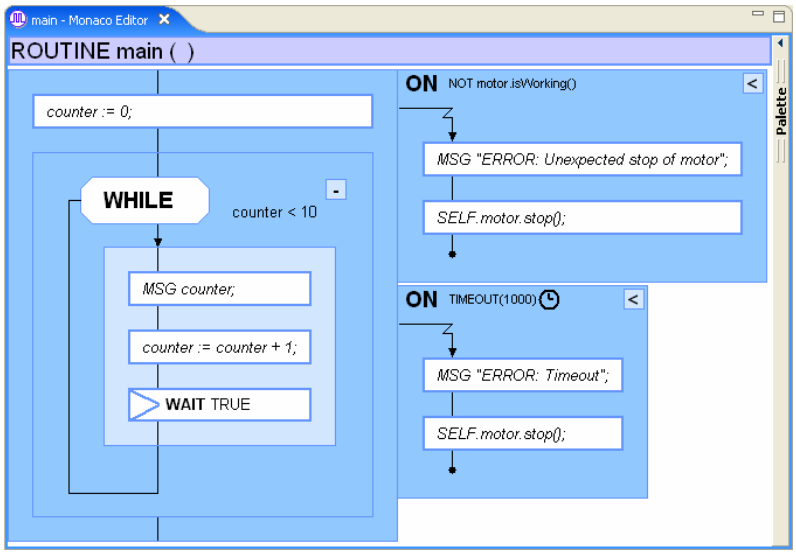


Abbildung 40: Fertiger Code der Routine Main.main

# 7 Zusammenfassung

Im Rahmen dieser Arbeit wurde der visuelle Editor für die Programmiersprache Monaco vorgestellt. Es wurde ein Überblick über den aktuellen Projektstand der prototypischen Implementierung des visuellen Editors im Rahmen der Monaco-IDE geboten. Dabei wurde die Sprache Monaco kurz vorgestellt, es wurden die visuellen Notationen beschrieben und die technische Umsetzung des Editors erklärt. Abschließend wurden die Funktionen des visuellen Editors sowie assistierender Teile der Monaco-IDE erläutert und in einem Beispielprojekt vorgeführt.

Die Nutzung der betriebssystemunabhängigen Java-Plattform ermöglicht die Nutzung der entwickelten Werkzeuge auf unterschiedlichen Systemen. Wie dargestellt wurde, hat die Eclipse RCP als Grundlage für die Implementierung zu einer vergleichsweise kurzen Implementierungszeit beigetragen. Es konnte ein voll funktionsfähiger Prototyp erstellt werden, der bereits nahe an die Funktion eines Texteditors herankommt oder in gewissen Bereichen bereits übertrifft.

Die entwickelte visuelle Notation erlaubt Domänenexperten ohne umfassende Programmier-Kenntnisse, Monaco-Programme zu lesen und zu verstehen. Eine wichtige Aufgabe für die Zukunft wird es aber sein, diese Notation weiter zu optimieren und an die Bedürfnisse der zukünftigen Benutzer anzupassen. Der aktuelle Prototyp stellt lediglich den ersten Schritt des Projektes dar, dessen Endziel ein Endbenutzer-Programmiersystem ist.

# Literaturverzeichnis

[Arth04] J. Arthorne, C. Laffra: *Official Eclipse 3.0 FAQs*. Pearson Education, Boston, 2004

[Daum05] B. Daum: *Rich-Client-Entwicklung mit Eclipse 3.1*. dpunkt Verlag, Heidelberg, 2005

[GOF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, 1995

[Hare87] D. Harel: *Statecharts: A Visual Formalism for Complex Systems*. In: Science of Computer Programming. 8/1987, North Holland, S. 231-274

[Hurn06] D. Hurnaus: *Eine domänenspezifische Programmiersprache für Maschinensteuerungen: Entwurf eines Compilers und einer Ausführungsumgebung*. Diplomarbeit, Fachhochschule Hagenberg, 2006

[KP88] G. Krasner, S. Pope: *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*. Journals of Object-Oriented Programming, 1(3):26-49, August/September 1988

[Lude07] J. Ludewig, H. Lichter : *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken*. dpunkt Verlag, Heidelberg, 2005

[Mcaf05] J. McAffer, J. Lemieux: *Eclipse Rich Client Platform*. Addison Wesley, 2005

[Moor04] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, P. Vanderheyden: *Eclipse Devleopment using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, First Edition, 2004

[Prae07] H. Prähofer, D. Hurnaus, R. Schatz, H. Mössenböck: *The Domain-Specific Language Monaco*. Technical Report, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, Austria, 2007.

[Wimm06] C. Wimmer: *Komponententechnologie*. Vorlesungsunterlagen, Universität Linz, WS 2006/07

[Wint05] M. Winter: *Methodische objektorientierte Softwareentwicklung*. Dpunkt Verlag, 2005

# Abbildungsverzeichnis