TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Institut für Software- und Multimediatechnik, AG Didaktik der Informatik

Strukturelle Untersuchung einer IDE mit dem Ziel einer möglichst frei skalierbaren Anpassung der IDE von Lazarus

Diplomarbeit

zur Erlangung des akademischen Grades eines Diplom-Medieninformatikers der Fakultät Informatik der Technischen Universität Dresden

Vorgelegt von: Michael Kuhardt

Matrikel-Nr.: 2936155

Hochschullehrer: Prof. Dr. paed. habil. Steffen Friedrich

Betreuer: Dr. rer. nat. Holger Rohland

Bearbeitungszeitraum: 01.09.2009-28.02.2010

Dresden, 23. Februar 2010

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API ..... Application Programming Interface

CASE ..... Computer-Aided Software Engineering

DIN ..... Deutsches Institut für Normung

EJB ..... Enterprise Java Beans

et al. ..... et alii (lat.: und andere)

etc. ..... et cetera (lat.: und so weiter)

GIF ..... Graphics Interchange Format

HTML ..... Hypertext Markup Language

IBM ..... International Business Machines Corp.

IDE ..... Integrated Development Environment

ISO ..... International Organization for Standardization

JAR ..... Java Archive

JSP ..... Java Server Pages

LCL ..... Lazarus Component Library

MP3 ..... MPEG-1 Audio Layer 3

OSGi ..... Open Services Gateway initiative

OTI ..... Object Technology International

PET ..... Programm-Entwicklungs-Terminal

PHP ..... PHP: Hypertext Preprocessor

SDK ..... Software Development Kit

SQL ..... Structured Query Language

SWT ..... Standard Widget Toolkit

UML ..... Unified Modeling Language

URL ..... Uniform Resource Locator

USB ..... Universal Serial Bus

XML ..... Extensible Markup Language

Auflistungen

1 Einleitung

1.1 Einführung und Motivation

Das Gebiet der imperativen Programmierung stellt einen Bereich der Informatikausbildung an sächsischen Gymnasien dar. Um im Felde der zahlreichen umfangreichen Entwicklungsumgebungen den Schülern sowie Lehrern die Konzentration auf die wesentlichen Aspekte der Programmierung zu ermöglichen, wurde vom Autor in einer bereits vorgelegten Belegarbeit eine prototypische Version der Entwicklungsumgebung Lazarus vorgestellt. Hierbei werden Komponenten sowie Eigenschaften, welche aus didaktischer Sicht unnötig sind, ausgeblendet. Diese Arbeit stellt somit einen ersten Schritt zu einer schulischen Entwicklungsumgebung dar.

1.2 Zielstellung

Das Ziel dieser Diplomarbeit besteht darin, ausgehend von der prototypischen Version, eine möglichst frei skalierbare Version der Lazarus-IDE zu entwickeln. Diese soll es dem versierten Lehrer ermöglichen, eine auf seine didaktischen Intentionen und die Leistungsfähigkeit seiner Schüler angepasste Lazarus-Version zu installieren und im Unterricht zu verwenden. Hierfür ist zunächst eine theoretische Analyse der Entwicklungsumgebung Eclipse vorzunehmen. Weiterhin ist zu untersuchen, welche didaktischen Anforderungen prinzipiell an eine schulische Entwicklungsumgebung zu stellen sind. Auf Basis dieser Untersuchungen sowie durch gezielte Befragung von Fachlehrern sowie Experten sind die Anforderungen an das Werkzeug zu verifizieren und zu realisieren.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in drei Schwerpunkte gegliedert. Im Kapitel *„Analyse der IDE Eclipse“* werden Entwicklungsumgebungen sowie deren Bedeutung für die Softwaretechnologie zunächst historisch eingeordnet. Es folgt eine Beschreibung des modularen Aufbaus der Entwicklungsumgebung Eclipse. Im Zentrum dieser Untersuchung steht die Frage, wie die IDE mit unterschiedlichen Programmiersprachen genutzt werden kann. Auf Basis dieser Untersuchung

werden Lösungsansätze für die Integration eigener Weiterentwicklungen bei Aktualisierung der Gesamtumgebung vorgestellt. Das Kapitel endet mit der Untersuchung der Gebrauchstauglichkeit gemäß *ISO 9241*.

In einem weiteren Kapitel werden Anforderungen formuliert, die aus didaktischer Sicht an eine schulische Entwicklungsumgebung zu stellen sind. Die Grundlage hierzu bildet eine theoretische Untersuchung sowie die Befragung von Fachlehrern.

Die Realisierung der frei skalierbaren LAZARUS-Version wird in Kapitel 4 vorgestellt. Zunächst werden hierfür die relevanten Teile der Lehrerbefragung ausgewertet. Daraus werden unterschiedliche, für die IDE benötigte, Konfigurationsstufen abgeleitet. Den Abschluss des Kapitels bilden die Implementierung der vorgenommenen Änderungen sowie die Evaluierung der Ergebnisse mit der Zielgruppe. Die Arbeit endet mit einem Fazit sowie einem Ausblick auf mögliche weiterführende Arbeiten.

2 Analyse der IDE Eclipse

2.1 Historische Entwicklung von Entwicklungsumgebungen

„Sind statt Schablone, Zettel und Bleistift nunmehr Funktionstasten am Bildschirm die künftigen Werkzeuge der Programmierer?“ Diese Frage stellte die Fachzeitschrift Computerwoche im November des Jahres 1975 nach der Vorstellung des Programm-Entwicklungs-Terminal-System PET aus dem Hause Softlab (vgl. Computerwoche (1975)). Diese Frage war berechtigt, denn der Alltag des Programmierens gestaltete sich bis zu diesem Zeitpunkt wie folgt:

> „Noch kritzeln die meisten von ihnen die Befehle an den Rechner, die in sinnreicher Abfolge schließlich das Programm ergeben, erst mal mit Bleistift auf unzählige Blätter. Mit großflächigen Diagrammen an den Wänden versuchen sie, den Überblick zu behalten. Mehr als zwei Drittel ihrer Zeit vertun sie nach Ansicht von Rationalisierungsexperten ziemlich sinnlos: Sie wühlen in Handbüchern der Computer-Hersteller und in Aktenordnern, die ihre Vorgänger hinterlassen haben. Und sie vertun Zeit, darauf zu warten, dass ihr neues Programm im Rechenzentrum getestet worden ist.“ (SpiegelWissen (1983))

Eingeführt im Jahr 1975, gilt das **PET als einer der ersten Vorläufer heutiger Entwicklungsumgebungen**. Zielstellung war es, mit diesem System die Effektivität der Programmierung zu steigern. Dies sollte zunächst mit neuen Methoden erreicht werden. So etwa durch **hierarchische Modularisierung sowie strukturierte Programmierung**, die mit dem PET-System Einzug hielt. Das Konzept der strukturierten Programmierung geht dabei auf eine Theorie von Boehm und Jacopini aus dem Jahr 1966 zurückt. Sie zeigten damals auf, dass Anwendungen mit Hilfe einer begrenzten Anzahl von Steuerungskonstrukten entwickelt werden können. Durch entsprechende Konstrukte sollte die Anzahl an Fehlern verringert und folglich die Kosten der Entwicklung reduziert werden (vgl. Thurner (1995b)). Das PET-System bot hierzu entsprechende Formalismen-Vorgaben über ein Bildschirm-Terminal an. In der Folge waren somit nur noch Statements einzugeben. Hierfür standen Funktionen wie **das Editieren, eine Suche sowie Formatierungshilfen** zur Verfügung. Als Programmiersprachen wurden zunächst Sprachen der zweiten (*Assembler*) sowie dritten (*COBOL*) Generation unterstützt (vgl.

Boehme (1996)). Des Weiteren stand eine eigens entwickelte Strukturspache zur Verfügung, die automatisch relevante Strukturausschnitte anzeigt. Die Programmentwicklung wurde durch **Funktionstasten** (*W* für *„While“* oder *I* für *„If-Then-Else“*) gestützt und vereinfacht. Mit Hilfe des PET-Systems konnten so neue Programme am Bildschirm-Terminal entwickelt werden. Über ein Band oder eine DFÜ-Verbindung übermittelten die Programmierer die fertigen Programmteile in der korrekten Reihenfolge dann zur Kompilierung an den Großrechner.

In der Folge bekam zunehmend der Gedanke der maschinellen Unterstützung von Entwicklungsmethoden Zuspruch. Die kommerzielle Verbreitung erster so genannter CASE-Systeme verlief jedoch zunächst schleppend (vgl. Thurner (1995b)).

Die Verbreitung von Programmier- und später auch Entwicklungsumgebungen ist zugleich jedoch auch sehr eng mit der Verbreitung und Entwicklung der Programmiersprachen verbunden. Standen in frühen Sprachgenerationen die verfügbaren Operationen eines Rechners noch im Fokus, so orientierten sich höhere Programmiersprachen zunehmend an dem zu lösenden Problem. Das wurde über abstrakte Formulierung des Lösungsweges erreicht, der die Eigenarten der Hardware, auf der das Programm ausgeführt werden soll, nicht mehr in Betracht zieht (vgl. Gumm & Sommer (2008)). Als Beispiele können hierfür *BASIC* oder auch *Pascal* angeführt werden. Das prinzipielle Vorgehen beim Programmieren sah hier bis zur Mitte der 80er Jahre gleich aus:

- Verfassen und Speichern des Quelltexts mit Hilfe eines Editors
- Übersetzung des Quelltextes unter zu Hilfenahme eines Compilers[1] in Maschinensprache
- Editor und Compiler wurden nun abwechselnd solange verwendet, bis der Quelltext fehlerfrei war
- Anschließend erfolgte das Hinzubinden einer Laufzeitbibliothek mit Hilfe eines Linkers
- Anwendungsstart

[1] Abhängig von der Sprache konnte statt des Compilers auch ein Interpreter zum Einsatz kommen. Hierbei wurde nicht der komplette Quelltext übersetzt, sondern jeweils eine einzelne Programmieranweisung in ein Unterprogramm aus Maschinenbefehlen. Diese Unterprogramme werden sofort ausgeführt. Ein Befehl der mehrfach auszuführen ist, wird somit mehrfach übersetzt (vgl. Gumm & Sommer (2008)).

Unter anderem dieses konventionelle Vorgehen führte zu Problemen, die Engels & Schäfer (1989) als „Softwarekrise“ beschrieben. **Durch den ständigen Wechsel der einzelnen Werkzeuge, stieg der damit verbundene Aufwand und verursachte eine Kostenexplosion für qualitativ hochwertige Anwendungen.** Diese Softwarekrise wurde letztendlich jedoch durch neue Methoden sowie durch ingenieurmäßiges Vorgehen überwunden. **Einen Ansatzpunkt beschreiben Engels & Schäfer (1989) durch die Idee, entwickelte Methoden wieder als Software zu realisieren.** Ähnlich des bereits erwähnten CASE-Ansatzes waren also erneut Werkzeuge zu entwickeln, die den Entwickler unterstützen. **Erste Vertreter derartiger Programmiersysteme vereinten hierbei Werkzeuge wie Editor, Compiler oder auch Linker unter einer Oberfläche.** Diese Zusammenführung einzelner Tools charakterisieren Beide jedoch als noch unzureichend. Probleme sahen sie hier vor allem bei der Benutzerschnittstelle der einzelnen Werkzeuge:

- Die einzelnen Benutzerschnittstellen waren nicht oder kaum aufeinander abgestimmt.
- Erschwerter Wechsel der einzelnen Werkzeuge. So unterschied sich zum Beispiel die Art und Weise der Eingabe der verwendeten Kommandos.
- Einzelne Werkzeuge überlappten sich teilweise mit Funktionalitäten.
- Unterschiedliche Ausgaben der einzelnen Werkzeuge. So erzeugten manche Werkzeuge Ausgaben, die von anderen Nutzern gar nicht oder nur durch umständliche, aufwendige zusätzliche Eingriffe wieder verwendbar waren.

Die genannten Probleme führten lt. Engels & Schäfer (1989) in der Folge zu den ersten **Programmentwicklungsumgebungen**. Diese grenzten sich gegenüber den bisherigen Programmiersystemen durch einen **Satz integrierter Werkzeuge** ab. Deren Integration verlief im Weiteren so, dass sich **alle Werkzeuge dem Softwareentwickler global mit einer einheitlichen Benutzerschnittstelle präsentierten**. Weiterhin arbeiteten die Programmentwicklungsumgebungen intern auf hohen, d.h. weitgehend maschinenunabhängigen, Datenstrukturen. Diese repräsentierten alle bei der Entwicklung nötigen Informationen und garantierten so eine hohe Leistungsfähigkeit der einzelnen Werkzeuge. Weiterhin konnte der **werkzeugübergreifende Austausch von Informationen** so automatisiert und folglich unabhängig vom

Nutzer realisiert werden. Aus diesen Punkten folgern sie somit einen großen Vorteil für den Entwickler, der nun **innerhalb einer Umgebung wirklich alle Aktivitäten der Entwicklung miteinander verzahnt ausführen kann.**

Einen ersten Vertreter dieser Kategorie stellt die Programmentwicklungsumgebung TURBO PASCAL von BORLAND im Jahre 1983 dar. Nach GUMM & SOMMER (2008) zeigte es als Erstes gleichzeitig einen Mittelweg zwischen den erwähnten interpretierenden und kompilierenden Systemen auf. Der in das System integrierte Compiler erreichte ähnliche Geschwindigkeiten wie ein Interpeter. Weiterhin war der Compiler mit dem Linker verbunden. Traten Fehler auf, erfolgte ein Sprung in die entsprechende Quelltextzeile um den Fehler zu markieren (vgl. LEITENBERGER). In Version 7 aus dem Jahr 1987 erweiterte BORLAND die Umgebung um Syntaxhervorhebung sowie einen Debugger.

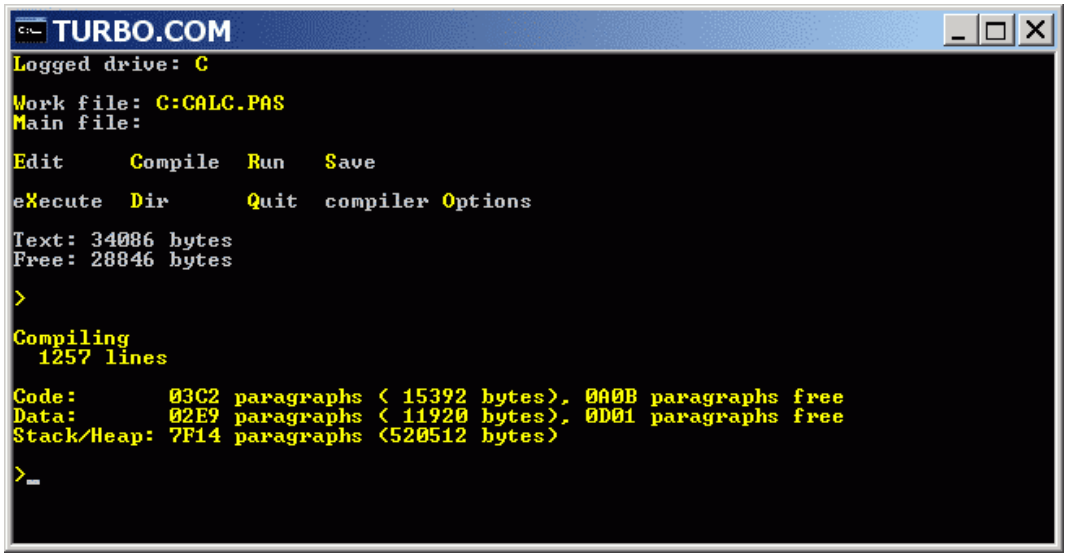

Abbildung 1: Turbo Pascal 1.0 nach LEITENBERGER

Als Gegenstück für die Sprache *Basic* kann hier QUICKBASIC von MICROSOFT angeführt werden. Die erste Version wurde bereits im Jahre 1985 vorgestellt. Jedoch wurde ein Editor zum Bearbeiten des Quellcode erst ein Jahr später innerhalb der Version 2.0 bereitgestellt. Gleichzeitig hielten Block-Statements wie *If-Then-Else* Einzug in die Umgebung. Mit Version 3.0 wurde, ähnlich wie bei TURBO PASCAL im Jahre 1987, ein einfacher Debugger mit Haltepunkten integriert.

Darüber hinaus hielten Statements wie *Select-Case*, *Do-Loop* oder *Const* sowie die Überwachung von Variablen Einzug (vgl. Microsoft Corporation (2001)). Die Abbildung 2 zeigt Version 4.5 aus dem Jahr 1988.

Abbildung 2: QuickBasic 4.5 nach Gerald T. Aitken *(2007)*

Nach dem Erfolg von Turbo Pascal stellte Borland weitere Turbo-Umgebungen für die Sprachen *C*, *Basic* sowie *Prolog* bereit. Bereits seit dem Jahre 1983 wurde von Microsoft die Umgebung Microsoft C (ehemals der C-Compiler Lattice C) veröffentlicht.

Engels und Schäfer kritisierten jedoch an diesen Systemen, dass die **Bandbreite an Funktionen längst nicht ausreichend** war. Softwareentwicklung geht weit über das Bearbeiten und Übersetzen des Quellcode hinaus. So forderten sie z.B. Werkzeugunterstützung für die Erstellung und Bearbeitung der im Prozessverlauf anfallenden Dokumente, wie Pflichtenheft, Modulararchitekturen oder auch Projektplänen. Das wurde von den Programmierentwicklungsumgebungen zu diesem Zeitpunkt nicht unterstützt.

Folgt man den Ausführungen Thurners (vgl. Thurner (1995a)), dann nahmen auch die Verbreitung neuer Methoden der Softwaretechnologie sowie der starke Zuwachs an PCs einen spürbaren Einfluss auf die Entwicklung und Verbreitung von Entwicklungsumgebungen. So ermöglichte der PC Mitte der 80er Jahre Dank der **Grafikfähigkeiten**, des **erschwinglichen Preises** und der

damit verbundenen **Flexibilität**, einzelnen Softwareentwicklern die kundennahe Arbeit durch rasche Ad-hoc-Lösungen. Durch Großrechneremulatoren für den PC entstanden zahlreiche neue CASE-Tools. CASE bezeichnet dabei Software, die den Entwickler bei Tätigkeiten des Entwicklungsprozesses wie

- Anforderungsanalyse,
- Entwurf,
- Programmentwicklung sowie
- Tests

unterstützt. Sommerville (2007) unterscheidet hierbei in *CASE*-Werkzeuge, *CASE*-Arbeitsplätze sowie *CASE*-Umgebungen. *Werkzeuge* unterstützen nur eine einzelne Aktivität. Als Beispiele führt er Entwurfseditoren, Data Dictionaries, Compiler oder auch Debugger an. Die *Umgebungen* hingegen unterstützen die meisten oder alle Aktivitäten des Entwicklungsprozesses. Da beide gleichartige Werkzeuge anbieten und diese unter einer Oberfläche wie beschrieben integrieren, werden die Begriffe *CASE*-Tool sowie IDE oft synonym in der Literatur verwendet (vgl. Balzert (2008), Nödler (2005)).

Mit den ersten Tools der 80er Jahre war nach Thurner (1995a) auch eine **Renaissance der strukturierten Methoden der Programmierung** verbunden. Diese waren zwischenzeitlich beispielsweise zu Gunsten des Wasserfallmodells[2] oder des Rapid-Prototypings eingestellt worden. Diese Umgebungen verbesserten zunächst nicht die Qualität des Codes, führten jedoch **zu höherer Produktivität und zu kundengerechten Systemen**. Als negativen Aspekt merkte Thurner an, dass es einer **langen Ausbildungs- und Anlaufzeit** zum Erwerb der Methodenkenntnisse bedurfte. Auch „einfachere“ Systeme bedeuteten hier keinen Gewinn. Zwar konnten schnell Diagramme „zusammengeklickt“ werden, jedoch blieb der Erfolg ohne methodisches Wissen aus. Die *CASE*-Euphorie der 80er Jahre (vgl. Herzwurm et al. (1994)) ebbte ab. So resümierte auch Sommerville, dass die **ursprünglich erwarteten massiven Kostenersparnisse ausblieben**. Herzwurm et al. (1994) wiesen jedoch darauf hin, dass sich schlicht

[2] „Ein Wasserfallmodell unterteilt die Softwareentwicklung in einzelne Phasen, wobei die Ergebnisse einer Phase in die Folgephasen „fallen“, d.h. dort als Eingabe benutzt werden.“(vgl. Schneider & Werner (2001)).

die Aufgaben gewandelt hatten:

> „Neben der klassischen Unterstützung der Softwareentwicklung durch die Bereitstellung oberflächen- und datenintegrierter Werkzeuge werden CASE-Werkzeuge heute vielfach für Unterstützungsfunktionen auf der fachlichen Ebene - Stichwort Geschäftsprozessmodellierung - und für Querschnittsfunktionen wie Projekt- und Prozessmanagement eingesetzt. Auch die Charakteristika der Zielsysteme haben sich gewandelt: Statt zeichenorientierter Hostanwendungen gilt es immer mehr, methodisch und technisch anspruchsvollere Client-Server-Applikationen mit grafischen Oberflächen für verteilte Umgebungen zu realisieren.“

Zu einer phasenübergreifenden, werkzeuggestützten Softwareentwicklung gibt es nach ihrer Auffassung zu *CASE* keine Alternative.

Sommerville erwähnt hierzu einschränkend, dass es sich bei dem Prozess der Softwareentwicklung noch immer um eine auf kreativem Denken basierende Entwurfsaktivität handelt. Vorhandene Tools automatisieren letztlich jedoch nur entsprechende Routinen. Weiterhin kritisiert er eine bislang schlecht unterstützte Teamarbeit.

Zusammenfassend kann gesagt werden, dass mit Einführung von integrierenden Systemen zur Entwicklung von Softwareprodukten der **Softwareentwicklungsprozess** tatsächlich **deutlich vereinfacht und effizienter gestaltet** werden konnte. Ferner wurde vor allem sowohl die **Produktivität der Entwickler als auch die Qualität der Anwendungen gesteigert**. Anhand des PET-Systems wurde deutlich, wie mit Hilfe von Editierfunktionen, einer Strukturspracheowie vorhandenen Funktionstasten die Entwicklung von Anwendungen erleichtert werden konnte. Während mit Hilfe des Systems die Kosten für den Unterhalt eines Systems auf 10 % der Personalkosten reduziert wurden, berichteten Kunden des Systems darüber hinaus von einer durchschnittlichen Steigerung der Produktivität um 30 bis 50% (vgl. Computerwoche (1979)). Erste **Programmentwicklungsumgebungen** wie Turbo Pascal **brachen den Prozess des ständigen Wechsels einzelner Werkzeuge auf und integrierten diese in einer Anwendung**. Darüber hinaus ermöglichten sie den konformen Datenaustausch der einzelnen Werkzeuge untereinander. Turbo Pascal vereinigte so die leichte Handhabung ei-

nes Interpreters mit der Schnelligkeit kompilierter Programme. Die Testzyklen wurden folglich erheblich kürzer (vgl. Ebner & Klaun (1998)). Vor allem mit Hilfe von *CASE*-Tools konnte erstmals der **gesamte Prozess der Softwarentwicklung abgebildet** werden. Das führte in der Folge zu enormen Steigerungen der Produktivität der Entwickler. Mit Verweis auf Huff nennt Sommerville immerhin einer Verbesserung von ca. 40%. Die Zeiten, in denen Entwickler „mit Bleistift auf unzählige Blätter kritzelten, mit großflächigen Diagrammen an den Wänden versuchten, den Überblick zu behalten...“ und ständig die benötigten Werkzeuge wechseln mussten, waren nunmehr Dank diverser Entwicklungsumgebungen sowie spezieller CASE-Tools endgültig beendet. Die Aussage von *Wolmeringer* bringt sehr anschaulich zum Ausdruck, wie sich die Arbeitsweise dank der integrierten Entwicklungsumgebungen änderte:

> „Bereits vor 30 Jahren, als *C*-Programmierung noch in den Kinderschuhen steckte, merkte man, dass es nicht genügt, den Quellcode mit einem Editor zu erstellen, zu kompilieren und zu testen. Es war auch ganz sinnvoll, eine Übersicht der Befehlssyntax greifbar zu haben. Außerdem konnte man effektiver arbeiten, wenn man Unterlagen zu den Routinen hatte, die man in seinen Programmen nutzen konnte, entweder selbst entwickelte oder zugekaufte. Praktisch war es auch, wenn man den Compiler starten konnte, ohne den Editor zu verlassen. Man konnte dann während des oft langen Kompilierungsvorganges weiterentwickeln.“ (Wolmeringer (2005))

2.2 Die Eclipse-Plattform

2.2.1 Überblick

Was ist Eclipse? Bei Eclipse handelt es sich um eine Plattform, welche vor allem durch die frei-verfügbare *Java*-Entwicklungsumgebung sowie die dahinter stehende große Community Bekanntheit erlangte. Bei näherer Betrachtung handelt es sich jedoch um wesentlich mehr, als nur eine reine IDE zur Entwicklung von *Java*-Anwendungen. Grundlage hierfür ist der dritte Stützpfeiler der Plattform: Die Eclipse-Plattform zur Integration externer Tools.

Nach Thomson (2003) ist die Entwicklung dieser Plattform sehr eng mit der Entwicklung der Kommunikationstechniken in den 90er Jahren verbunden. Mit der vermehrten Verbreitung des

Internet wurde die Abwicklung von Geschäftsprozessen zunehmend ins Internet verlagert, um so den Kontakt zu Geschäftspartnern und den Kunden zu etablieren. Die entsprechenden Anwendungen bauten auf Grund verschiedener Technologien auf unterschiedliche Ressourcen auf. Als Beispiele führt er *Java* oder auch *XML* an. Zur Bearbeitung der verschiedenen Ressourcen wurden demzufolge auch unterschiedliche Anwendungen oder Editoren erforderlich. Als Grund führt Thomson die bis dahin klassische Art der Entwicklung von Anwendungen an. Die Entwickler hatten die vollständige Kontrolle über ihre Anwendungen. Sie alleine entschieden, wie ihre Entwicklungen arbeiteten und wann und wie diese weiterentwickelt werden.

Bei der Entwicklung des Eclipse-Projekts wurde hingegen konsequent ein anderer Weg beschritten. Er lässt sich am Besten durch die Vision: „A new way of looking how we build tools“ beschreiben. **Dave Thomson charakterisiert diese Vision als** nicht mehr werkzeug- oder anwendungszentriert, sondern als **plattformzentriert mit nur grundlegender Funktionalität**:

> „The bare-bones Eclipse-Platform is essentially a universal IDE - an IDE for anything and nothing in particulary.“ (Thomson (2003))

Ähnlich beschreiben das auch die Eclipse-Entwickler in ihrer Projektdokumentation:

> „By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a plug-in model.“ (IBM Corporation (2005))

Als Basis dient die Eclipse-Plattform. Sie wird den Nutzern als Grundgerüst zur Verfügung gestellt und kann durch eigene Erweiterungen (Plug-Ins) über genau definierte Erweiterungspunkte mit weiteren Funktionalitäten versehen werden. Thomson sieht das Ergebnis somit nicht mehr nur als jeweils ein neues Werkzeug sondern eine neue Fähigkeit oder Funktionalität an, die nun durch Eclipse zur Verfügung gestellt werden kann. Gestartet wurde das Projekt in den 90er Jahren von *IBM* sowie *OTI*. Die beschriebene Plattform wird im Weiteren näher betrachtet.

2.2.2 Die Struktur der Installation

In der Folge werden die wichtigsten Elemente einer frischen Eclipse-Installation vor dem ersten Start kurz dargestellt. Bei Produkten auf Basis der Eclipse-Plattform kann zusätzlich ein Unterverzeichnis mit der *Java*-Runtime-Environment enthalten sein (vgl. Shavor et al. (2003)).

Verzeichnis / Datei	**Beschreibung**
Features	Enthält Unterverzeichnisse für die einzelnen, bei der Installation vorhandenen Features. Ein Feature kann ein oder mehrere Plug-Ins referenzieren.
Plugins	Enthält ein Unterverzeichnis für jedes Plug-In.
ReadMe	Enthält Informationen zum aktuellen Release
.eclipseproduct	Diese Datei identifiziert die Anwendung als Eclipse-basiertes Produkt.
EPL-v10.html	Enthält die Eclipse Public License in Version 1.0
Eclipse.exe (in Windows)	Ausführbare Datei zum Start der Anwendung
Install.ini	Datei zur Identifikation von Startoptionen
Notice.html	Eclipse.org-User-Agreement
Startup.jar	Laufzeit-Code für den Start

Tabelle 1: Struktur der Installation nach Shavor et al. (2003)

Für die Erweiterung von Eclipse sind die Verzeichnisse „Plugins“ sowie „Features“ von wesentlicher Bedeutung. Im einfachsten Fall ist es möglich, neue Features und Plug-Ins direkt in die vorgesehenen Verzeichnisse zu kopieren. Im Gegensatz zu den Features werden die Plug-Ins in diesem Fall dann jedoch nicht vom Update Manager verwaltet. Andernfalls muss die Installation über den Update Manager erfolgen.

Plug-Ins

Plug-Ins stellen in Ergänzung zur bestehenden Eclipse-Installation weitere Funktionen bereit. Jedes Plug-In befindet sich dabei in einem eigenen Verzeichnis. Der Name entspricht der *Java*-Namenskonvention mit Namen und der aktuellen Plug-In-Version. Des Weiteren besteht die Möglichkeit, auch Plug-Ins aus anderen Verzeichnissen einzubinden. Im Abschnitt 2.2.3 werden die Plug-Ins einer näheren Betrachtung unterzogen.

Features

Während Plug-Ins den Code und somit die Funktionalität zur Verfügung stellen, können Features als eine Verwaltungseinheit für diese verstanden werden. Sie werden zur Organisierung und Strukturierung von Plug-Ins sowie anderen Features verwendet. Nach Shavor et al. (2003) sind sie:

- Eine installierbare Einheit mit Funktionalität
  Entsprechend ihrer Definition können sie oft problemlos aktiviert und deaktiviert werden.
- Ein Paketkonstrukt
  Auf Basis der Features können Plug-Ins verwaltet oder über den Update Manager installiert bzw. deinstalliert werden.
- Ein logischer Container für Plug-Ins
  Plug-Ins werden über die Features identifiziert und somit vom Update Manager über ihre Features verwaltet.
- Ermöglichen das Einbetten weiterer verbundener Features
  So kann eine Menge von Features für einen bestimmten Dienst/eine Erweiterung verwaltet werden.

Die Namensgebung der Verzeichnisse setzt sich in Analogie zu den Plug-Ins aus der ID des Features sowie der Version zusammen.

2.2.3 Die Architektur

Die Eclipse-Plattform stellt eine offene Architektur zur Verfügung. Somit werden klare Ansatzpunkte definiert, über die Entwickler eigene Plug-Ins und somit Funktionalitäten hinzufügen können, ohne vorhandene Erweiterungen zu beeinflussen.

- Unterstützung der Entwicklung einer Vielzahl von Werkzeugen für die Anwendungsentwicklung.
- Unterstützung einer unbeschränkten Anzahl von Werkzeugentwicklern, einschließlich unabhängiger Softwareverkäufer.

- Unterstützung von Werkzeugen zur Bearbeitung beliebiger Datentypen (*HTML*, *Java*, *C*, *JSP*, *EJB*, *XML*, *GIF*, ...).
- Möglichst nahtlose Integration von Werkzeugen unterschiedlicher Datentypen und Anbieter.
- Unterstützung der Entwicklung von Anwendungen mit und ohne grafische Benutzerschnittstelle.
- Zahlreiche Betriebsysteme wie Windows oder Linux unterstützen.
- Bei der Anwendungsentwicklung der Beliebtheit der Programmiersprache *Java* Rechnung tragen.

Die Plattform selbst besteht aus **mehreren Sub-Systemen**. Diese wiederum sind entweder **als Einzel-Plug-In oder als eine Vielzahl von Plug-Ins realisiert**. Sie setzen dabei direkt auf die Laufzeitumgebung auf. Die nachfolgende Abbildung 3 vermittelt einen ersten Überblick über das so genannte Eclipse SDK.

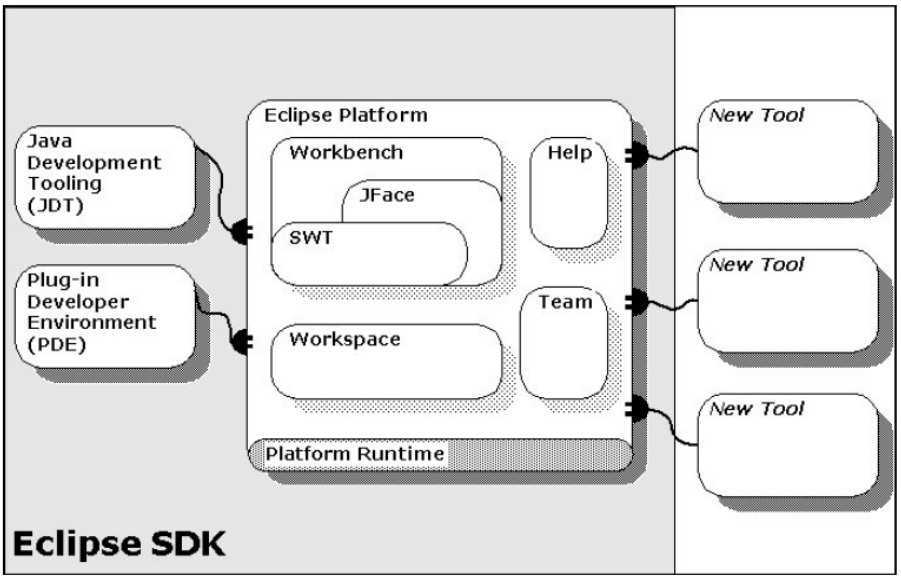

Abbildung 3: Eclipse-*Plattform-Architektur nach* Object Technology International, Inc. *(2003)*

Die weiter unten aufgeführte Tabelle 2 enthält einen Überblick über die einzelnen Komponenten der Plattform und gibt in kurzer Form hierzu entsprechende Erläuterungen.

Komponente	Erläuterung
Platform Runtime	Definiert das Model der Erweiterungspunkte sowie der Plug-Ins. Die Plug-Ins und ihre Erweiterungspunkte werden dynamisch erkannt und in einer Registry gespeichert. Die Laufzeit wurde mit Hilfe des Framework OSGi implementiert (vgl. Daum (2007)).
Resource Management (Workspace)	Definiert eine API zur Erstellung und Verwaltung von Ressourcen wie Projekten, Verzeichnissen und Dateien, welche von den Werkzeugen erstellt werden.
Workbench UI	Die eigentliche Benutzerschnittstelle, mit welcher der Nutzer mit der Plattform agiert. Sie definiert Erweiterungspunkte um UI-Komponenten (Views, Menüs, Aktionen) hinzuzufügen. Darüber hinaus enthält sie die Toolkits *JFace* sowie *SWT* (vgl. 2.2.3). Diese Struktur ermöglicht das Erstellen von eigenständigen Anwendungen, die unabhängig vom Workspace-Modell sind.
Help System	Stellt Erweiterungspunkte für Hilfe sowie Dokumentationen zur Verfügung.
Team Support	Definiert ein Team-Model zur Versionsverwaltung.
Debug Support	Definiert ein sprachunabhängiges Debug-Model sowie UI-Klassen zur Erstellung von Debuggern.
Andere Utilitys	Plug-Ins mit Funktionen zur Suche, Vergleich, dynamisches Update vom Server, ...

Tabelle 2: Die wichtigsten Komponenten der Plattform-Architektur nach IBM Corporation (2005)

Plug-Ins

Wie bereits im Abschnitt 2.2.2 „Die Struktur der Installation“ erläutert, basiert die Plattform auf dem Konzept der Plug-Ins. Hierbei handelt es sich um die **kleinste Plattformeinheit**, die entwickelt und vertrieben werden kann (vgl. Object Technology International, Inc. (2003)). Ein Werkzeug kann dabei als einzelnes Plug-In oder durch mehrere Plug-Ins realisiert werden. Die Runtime-Plattform ist die einzige Komponente, die nicht als Plug-In angelegt ist. **Typischerweise besteht ein Plug-In aus reinem *Java*-Code innerhalb einer *JAR*-Bibliothek sowie aus weiteren Ressourcen (Bildern, Templates, ...).** Die Plattform lässt sich somit in mehrere Bereiche unterteilen. So stellen unterschiedliche Plug-Ins zum Beispiel den Workbench sowie den Workspace zur Verfügung, welche wiederum aus mehreren Plug-Ins bestehen.

Jedes Plug-In hat ein so genanntes „Manifest“. **Dieses Manifest (Datei) beschreibt die Verbindungen mit anderen Plug-Ins.** Im Verbindungsmodell werden die Erweiterungen und entsprechenden Erweiterungspunkte zu anderen Plug-Ins dargestellt. Somit ist diese Datei für Entwickler wichtig, um die korrekten Erweiterungspunkte identifizieren zu können. Innerhalb der *XML*-Datei sind die zwei folgenden Tags besonders wichtig:

- <*extension-point*> - Definiert den neuen Erweiterungspunkt
- <*extension*> - Trägt zu einem bereits existierenden Erweiterungspunkt bei

Über zusätzliche *XML*-Tags können weitere, von der Erweiterung verwendete, Elemente deklariert und somit auch kommuniziert werden. Jeder Erweiterungspunkt hat dabei einen eigenen Identifikator. Dieser ist durch die *Java*-Namenskonvention spezifiziert. Nachfolgend einige Beispiele für Erweiterungspunkte.

Erweiterungspunkt	**Erläuterung**
org.eclipse.ui.actionSets	Ermöglicht das Hinzufügen von Aktionen zu den Menüs oder Toolbars des Workbench
org.eclipse.ui.viewActions	Ermöglicht das Hinzufügen von Elementen zu den Toolbars oder Pull-down-Menüs einer Ansicht

Tabelle 3: Beispiele für Erweiterungspunkte

Ein solcher Erweiterungspunkt kann ferner ein entsprechendes API zur Verfügung stellen, welches von anderen Plug-Ins als Erweiterung implementiert werden kann.

Die Runtime-Plattform ist für das Laden der jeweiligen Plug-Ins verantwortlich. Beim Start der Anwendung werden sämtliche bekannte Plug-Ins inklusive ihrer Manifeste geladen und als Plug-In-Registry im Speicher abgelegt. Treten bei diesem Vorgang Fehler auf (z.B. fehlende Plug-Ins), werden die Fehler protokolliert. Über die Plattform-API kann jedes Plug-In auf diese Registry zugreifen (vgl. Shavor et al. (2003), IBM Corporation (2005)). Nach erfolgreichem Laden der Plug-Ins müssen diese aktiviert werden. Die Aktivierung erfolgt durch Nutzerinteraktion und somit erst dann, wenn das Plug-In tatsächlich benötigt und ausgeführt wird. Bei diesem Vorgang wird nur die Funktionalität dieses Plug-Ins aktiviert. Verwendet es Funktionen weiterer Erweiterungen, werden auch diese wiederum erst bei Bedarf aktiviert. Nach

erfolgreicher Aktivierung bleiben die Plug-Ins bis zum Beenden der Anwendung und somit der Plattform aktiv.

Workspace

Der Workspace eines ECLIPSE-Nutzers stellt die reguläre Datenbasis dar, auf der sämtliche installierte Tools und Plug-Ins arbeiten. Der Workspace besteht dabei aus mehreren, vom Nutzer ausgewählten Projekten. Diese enthalten alle erforderlichen Objekte (Quellcode, Bibliotheken, ...). Deren Verteilung auf das zu Grunde liegende Dateisystem wird somit abstrahiert (vgl. DAUM (2007)).

Als Bestandteil des Workspaces wird ferner der „Mechanismus der Projekt-Naturen“ bereitgestellt. Es handelt sich hierbei um Erweiterungen, mit denen Projekten bestimmte Eigenschaften zugewiesen werden können. So werden zum Beispiel Projekten mit *Java*-Natur der *JavaBuilder* oder ein entsprechendes Symbol zugewiesen (vgl. WALEZAK (2008)). Für jedes Projekt wird die Natur des Projektes im entsprechenden Manifest über den Erweiterungspunkt *org.eclipse.core.resources.natures* definiert (vgl. DAUM (2006)). Einem einzelnen Projekt können so viele Naturen wie erforderlich zugewiesen werden.

Zur Annotierung von Ressourcen stellt der Workspace Markierungen zur Verfügung. Fehlermeldungen des Compilers, Lesezeichen, Suchergebnisse oder auch Debugger-Punkte können hierdurch aufgezeichnet und nachverfolgt werden. Um das Risiko von Dateiverlusten zu minimieren, steht fernerhin eine einfache Workspace-History bereit. Der jeweilige Nutzer kann die entsprechenden Einstellungen nach eigenem Ermessen spezifizieren.

Um Werkzeugen wie dem Compiler die koordinierte Manipulation und Transformationen zu ermöglich, wurde ein Framework mit inkrementellen Project-Buildern integriert. Für ein Projekt können mehrere verschiedene dieser Builder registriert werden.

Workbench, SWT und JFace

Die Benutzerschnittstelle der ECLIPSE-Plattform ist im Wesentlichen auf Basis eines Workbenches realisiert. Dieser stellt die Struktur bereit und stellt dem Nutzer die Benutzerschnittstelle zur Verfügung. Realisiert wurde er mit Hilfe von Werkzeugen, die dem Nutzer ebenso zur Ver-

fügung stehen - *SWT* und *JFace*.

Das Standard Widget Toolkit stellt Widgets sowie eine grafische Bibliothek auf Basis des darunter befindlichen Betriebssystems zur Verfügung. Außerdem ist eine plattformunabhängige *API* integriert. **Hintergrund dieser Architektur war das Ziel, plattformunabhängige und somit portable *Java*-Anwendungen zu schaffen, die sich jedoch in die Benutzerschnittstelle des jeweiligen Betriebssystems integrieren.**

Die JFace-Schnittstelle hingegen stellt Klassen für typische Aufgaben einer Benutzerschnittstelle zur Verfügung. Sie vereinfacht somit das Erstellen einer Benutzerschnittstelle erheblich. Für grafische Objekte werden hierfür *SWT*-Komponenten verwendet. Somit ist die API sowie die Implementierung unabhängig vom jeweiligen Window-Manager.

Der Workbench stellt im Gegensatz zu *SWT* oder *JFace*, welche eher grafische Schnittstellen zur Anwendungsentwicklung bereitstellen, den eigentlichen Kern der Benutzerschnittstelle zur Verfügung. Er basiert auf dem Konzept der Editoren, Views (oder Sichten) sowie der Perspektiven und liefert die Struktur, aus der heraus die einzelnen Werkzeuge mit dem Nutzer interagieren.

Die Editoren ermöglichen das Manipulieren von Objekten. Für Texte ist ein Standard-Editor integriert. Über Plug-Ins sind weitere Editoren integrierbar. **Views hingegen stellen innerhalb des Workbench Informationen über die in Bearbeitung befindlichen Objekte zur Verfügung.** Hierdurch kann z.B. die Arbeit im Editor durch zusätzliche Informationen gestützt werden. Views sowie Editoren können darüber hinaus in **Perspektiven gruppiert** werden. Der Workbench erlaubt mehrere separate Perspektiven, von denen jedoch immer nur eine sichtbar sein kann. Es ist jedoch möglich, innerhalb der Perspektive beliebig viele Editoren sowie Views zu arrangieren. Außerdem ermöglicht das Konzept der Perspektiven den nahtlosen Wechsel zwischen einzelnen Vertretern. **Aufgabenabhängig können so einzelne Perspektiven zusammengestellt und bei Bedarf gewechselt werden.** Nach der Installation stehen bereits Perspektiven, wie etwa die Team-Unterstützung, zur Verfügung.

Die Implementierung des Workbenches basiert auf den vorgestellten *SWT*- sowie *JFace*-Technologien.

2.2.4 Der Update-Manager

> „The Update Manager allows you to find new plug-ins on your machine, your network, or the Internet, compare new plug-ins to your configuration, and install only those that are compatible with your current configuration.“ (Eclipse Foundation (b))

Der Update-Manager der Eclipse-Plattform dient der Verwaltung sowie Installation und Deinstallation von Features. Hierzu erstellt der Update-Manager eine Konfiguration und verwaltet eventuelle Veränderungen, die im Dateisystem auftreten. Jeder verwendete Workspace verfügt dabei über eine eigene Konfiguration. Das ermöglicht das komfortable Wechseln zwischen diversen Workspaces und somit unterschiedlichen Konfigurationen an Erweiterungen. Zur Verwaltung von Features werden Installationsseiten verwendet. Neben den bereits bei der Installation zur Verfügung stehenden, können in der Folge weitere Installationsseiten externer Anbietern hinzugefügt werden. Bei den Installationsseiten gibt es die in Tabelle 4 aufgeführten Typen (vgl. Shavor et al. (2003)).

Seite	**Beschreibung**
Basisseite	Die Installationsseite der Eclipse-Plattform - beinhaltet das Eclipse-Verzeichnis
Verknüpfte Seite	Wird über Link-Dateien zur Eclipse-Plattform hinzugefügt und zur Installation von Features über einen externen Installationsprozess verwendet. So können Features und Plug-Ins mit der Installation gekoppelt, jedoch an einem anderen Ort im Dateisystem abgelegt werden. Zur Identifikation als Erweiterung muss eine *.eclipseextension-Datei* vorhanden sein. Dieses Verfahren wird als „Extension Install“ bezeichnet.
Update-Manager-Installationsseite	Diese Seite wird bei der Installation eines Features über den Update-Manager erstellt.
Updateseite	Es handelt sich hierbei um eine Dateistruktur mit Informationen über Features die installiert oder aktualisiert werden können.

Tabelle 4: Installationsseiten

Der Update-Manager ermöglicht es somit, **einzelne Erweiterungen gezielt zu aktualisieren**. Bis zum Release der Version 3.3 der Eclipse-Plattform war es jedoch noch nicht möglich, die Eclipse-Plattform zu aktualisieren. Mögliche Aktualisierungen beschränkten sich auf die jeweiligen Features (vgl. Eclipse Foundation (a)). **Bei den Releases handelt es sich**

um koordinierte Versionen. Sie bündeln bereits aufgabenspezifisch diverse Plug-Ins in einer aufeinander abgestimmten Konfiguration (vgl. NEUMANN (2009)). Kommt es dabei zu grundlegenden Änderungen an der Technologie, kann es jedoch (noch) zu Inkompatibilitätsproblemen führen. Das kann besonders bei solchen Plug-Ins auftreten, die spezifische Funktionalitäten dieses neuen Releases verwenden. Für diesen Fall ist ein Upgrade auf das neue Release der ECLIPSE-Plattform nötig. Die Workspaces der alten Versionen können jedoch mit dem neuen Release weiter verwendet werden. Zusätzliche Features sollten hingegen neu installiert werden (vgl. IBM CORPORATION (2009)). Inkompatibilitätsprobleme zwischen den letzten beiden Releases sollten sich auf ein Minimum beschränken:

> „One of the goals of ECLIPSE 3.5 was to move ECLIPSE forward while remaining compatible with previous versions to the greatest extent possible. That is, plug-ins written against the ECLIPSE 3.4 APIs should continue to work in 3.5 in spite of any API changes.“ (ECLIPSE FOUNDATION (2009))

2.3 Lösungsansätze für die Problematik der Integration eigener Weiterentwicklungen beim Update des Gesamtpakets

In den vorhergehenden Ausführungen wurden Architektur und Struktur der ECLIPSE-Plattform näher betrachtet. Im Weiteren sollen nunmehr Schlussfolgerungen für eigene Entwicklungen im Sinne der Aufgabenstellung gezogen werden.

Die ECLIPSE-Plattform bietet mit ihrer gut entwickelten Entwicklungsumgebung hierfür bereits eine solide Basis. Auf Grund der Struktur sowie der zu Grunde liegenden Architektur ist es mit entsprechendem Aufwand möglich, eigene Erweiterungen zu entwerfen und hinzuzufügen. Bereits jetzt existiert eine große Anzahl an Projekten und Erweiterungen die deutlich aufzeigen, dass die Integration mehrerer Programmiersprachen sowie weiterer Werkzeuge mit Hilfe des Plug-In Modells möglich ist.

Ein wesentlicher Aspekt dieser Arbeit sowie einer ihr vorausgehenden Belegarbeit des Autors ist es, den Einsatz möglicher Entwicklungsumgebung für die Programmiersprache *Pascal* in den sächsischen Gymnasien zu untersuchen. Neben der an den Gymnasien derzeit vorrangig

verwendeten DELPHI-IDE sowie der auch im weiteren Verlauf dieser Arbeit verwendeten IDE LAZARUS, **erscheint auch der Einsatz der Entwicklungsumgebung Eclipse prinzipiell möglich**. Es sei hierbei jedoch einschränkend erwähnt, dass - entsprechend der Recherchen des Autors - momentan noch kein eigenständiges *Pascal*-Plugin für ECLIPSE existiert.

Im Schwerpunkt der weiteren Betrachtungen steht nun zunächst die Frage, wie eine Entwicklungsumgebung auf Basis der ECLIPSE-Plattform mit Unterstützung mehrerer Programmiersprachen für Schulen realisiert werden könnte? Da eine *Pascal*-Version nicht existiert, wird die Standard-Version als möglicher Ausgangspunkt ausgewählt. Sie bietet bereits eine Unterstützung für die Sprache *Java*. Wird im Weiteren zunächst ausschließlich die Erweiterung der *Pascal*-Funktionalität betrachtet, dann ist in einer ersten Stufe die Integration eines entsprechenden *Pascal*-Plug-Ins vorzunehmen. In erster Konsequenz erfordert das die Implementierung eines eigenen Quelltexteditors. ECLIPSE stellt für potenziell erforderliche Editoren bereits einen Erweiterungspunkt zur Verfügung:

- *org.eclipse.ui.editors*

Über diesen Erweiterungspunkt kann der Entwickler zusätzliche Editoren für Ressourcen - wie zum Beispiel *Pascal*-Quelltext - hinzufügen. Analog zum *Java*-Editor kann dieser dann in der Folge auch über die entsprechenden Menüs in ECLIPSE verwendet werden. Ebenso können über den Erweiterungspunkt

- *org.eclipse.ui.editorActions*

Aktionen für die Toolbar des Editors sowie das entsprechende Menü hinzugefügt werden. Ein zur Deklaration der Erweiterung gehörendes einfaches Manifest (*plugin.xml*) könnte dann wie in Anhang A dargestellt, aussehen. Für die Realisierung eines geeigneten, komfortablen Editors bietet sich die Verwendung des *JFace*-Frameworks an. Dieses stellt standardmäßig bereits Vorlagen und Methoden zur Verfügung um auch Funktionen wie Syntaxhervorhebung zu realisieren (vgl. SHAVOR ET AL. (2003)). In Analogie zu LAZARUS ist in einer endgültigen Version ferner ein visueller Editor denkbar. Als erforderlicher Compiler könnte weiterhin der *FreePascal*-Compiler verwendet und in ECLIPSE integriert werden.

Im Kontext der *Pascal*-Entwicklung ist auch die **weitere Nutzung bereits bestehender Erweiterungen** denkbar. Mögliche Ansatzpunkte bieten hierfür zwei Projekte: Zunächst sei auf das Projekt „EclipseGavab“ (vgl. Gavab Research Group) verwiesen. Hierbei handelt es sich jedoch um eine eigenständige Eclipse-Distribution mit Unterstützung von *Java*, *C/C++*, *Pascal*, *PascalFC*, *Ruby* sowie *Haskell*. Sie kann somit nicht ohne Mehraufwand in eine bestehende Eclipse-Version integriert werden. Einen weiteren Ansatzpunkt sieht der Verfasser im Projekt „Pasclipse“ (vgl. Pasclipse Project). Dessen weitere Entwicklung wurde jedoch inzwischen eingestellt.
Mit o. g. Ausführungen wurde die prinzipielle Verfahrensweise für die Integration einer weiteren Sprache aufgezeigt. Für die Implementierung weiterer Programmiersprachen stehen mehrere Möglichkeiten zur Verfügung:

Zunächst kann, sofern ein entsprechendes Projekt vorhanden ist, dieses über den bereits beschriebenen Update-Manager in die Entwicklungsumgebung integriert werden. **Alternativ hierzu könnte auch ein weiteres, eigenes Plug-In entwickelt und eingebunden werden.** Dieses Verfahren verringert eventuelle Abhängigkeiten mit anderen Erweiterungen und schließt derartige Probleme aus. **Denkbar ist folglich für jede benötigte Sprache ein eigenes Plug-In.** Wie der Autor bereits in seiner Belegarbeit aufzeigte, werden im Informatikunterricht an sächsischen Schulen mehrere Themenblöcke behandelt. So zum Beispiel Themen wie Datenbanken oder Medientypen. Somit sind im Unterricht natürlich auch weitere Programmier- oder Auszeichnungssprachen wie *SQL*, *XML* oder auch *HTML* denkbar. Ein weiterer Ansatz wäre somit auch in einer **spezifische Erweiterung für sächsische Gymnasien** zu sehen. Analog der beschriebenen Vorgehensweise ist dann für jede verwendete Sprache ein Editor und somit ein eigenes Plug-In zu erstellen. Dieses kann dann entsprechend der spezifischen Anforderungen der Schulen angepasst werden. Diese sind dann jedoch gemeinsam in einem einzelnen Plug-In gekoppelt und als eine Erweiterung zu installieren (vgl. Abbildung 4). **Diese Vorgehensweise würde zusätzlich den Aufwand für die betreuenden Lehrer deutlich reduzieren. Nachfolgende Untersuchungen sollten deshalb hier ansetzen und unterschiedliche Varianten entsprechend des Verwendungszweckes tiefergehend analysieren.** Dabei ist zunächst jedoch zu klären, ob für die Lehrer leichtere Wartung durch eine Erweiterung oder

mehrere Erweiterungen, die dann jedoch individuell installierbar oder deinstallierbar sind, erreicht werden kann.

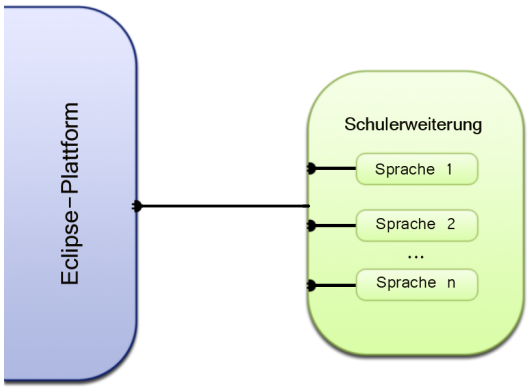

Abbildung 4: Konzept einer möglichen Erweiterung

Wie bereits erläutert, kann für jeden ECLIPSE-Workspace eine eigene Konfiguration verwendet werden. Somit können also auch unterschiedliche Features eingesetzt werden. Das eröffnet den Fachlehrern bei einem eventuellen schulischen Einsatz einen weiteren Spielraum. So ist z.B. **eine separate Konfiguration für jeden Unterrichtskomplex** mit anderen Werkzeugen für die Programmiersprache denkbar. Der entsprechende Workspace kann in ECLIPSE problemlos über *File -> Switch Workspace* ausgewählt werden. Falls die Funktion bei Beginn der Anwendung nicht bereits per Klick ausgeschaltet wurde, steht dem Nutzer außerdem jeweils beim Anwendungsstart ein so genannter „Workspace Launcher“ zur Verfügung. Workspaces können hier dann bequem aus einem Drop-down-Menü ausgewählt werden.

Ferner ist die Integration weiterer Perspektiven als zusätzliches Plug-In in Eclipse denkbar. Für die Bearbeitung der jeweiligen Programmiersprache können Perspektiven mit dem jeweiligen Quelltexteditor sowie den erforderlichen Tools zusammenstellt und zur Nutzung bereitgestellt werden. In diesem Fall ist der nachfolgende Erweiterungspunkt zu verwenden:

- *org.eclipse.ui.perspectives*

Mit dieser Vorgehensweise wird die Separierung in mehrere Workspaces überflüssig und das beliebige Wechseln der jeweiligen Perspektive innerhalb der Anwendung ermöglicht. Analog hierzu

können neue Perspektiven jedoch auch aus der Anwendung heraus erstellt werden.
Für die Integration sowie die Arbeit mit unterschiedlichen Sprachen bietet ECLIPSE wie aufgezeigt, also bereits einige Optionen an. **In diesem Zusammenhang ist die Frage hinsichtlich der Möglichkeit, eigene eingebundene Erweiterungen zu aktualisieren, von besonderer Bedeutung.** Möglich wird das über das bereits vorgestellte Konzept der Features sowie des Update-Managers. Für eventuelle Aktualisierungen der eigenen Erweiterungen ist zusätzlich eine Updateseite nötig (vgl. ECLIPSE FOUNDATION (c)).
Um ein Plug-In (oder mehrere verknüpfte) erstellen und später auch pflegen zu können, sind die in der Folge dargelegten Schritte notwendig.

- **Schritt 1 - Erstellen eines Feature:** Nach Erstellen eines Plug-In-Projektes kann direkt über ECLIPSE ein Feature-Projekt erzeugt werden (*New>Project>Plug-in Development>Feature Project*). In der Folge können zusätzliche Beschreibungen sowie Informationen angegeben werden.
- **Schritt 2 - Erstellen einer Update-Seite:** Um die Aktualisierungen zur Verfügung zu stellen, ist zunächst eine Update-Seite notwendig (*New>Project>Plug-in Development>Update Site Project*).
- **Schritt 3 - Update-URL erstellen und einbinden:** Durch das Erstellen einer Update-URL kann ECLIPSE eigenständig nach neuen Updates suchen und diese dem Nutzer vorschlagen.
- **Schritt 4 - Seite veröffentlichen**
- **Schritt 5 - Aktualisierung erstellen:** Zunächst ist über die *plugin.xml* die Erweiterung um eine Versionsnummer zu inkrementieren. Die erzeugte Seite kann in der Folge neu gefertigt und künftig über den Update-Manager aktualisiert werden.

Tabelle 5: Schritte zur Erstellung und Pflege eines Plug-In

ECLIPSE gestattet dem Entwickler somit in komfortabler Weise die Pflege seiner Plug-Ins, die er bei Bedarf auch jederzeit aktualisieren kann.

Auf Grund der bekannten Stabilität der Plattform, der großen Community des ECLIPSE-Projektes und der beschriebenen Aktualisierungsmechanismen erachtet der Autor einen **langfristigen Einsatz dieser Plattform in sächsischen Schulen prinzipiell für möglich**. Hierfür spricht ferner, dass **erforderliche Änderungen oder neue Versionen problemlos integrierbar**

sind. Sofern unterschiedliche Sprachen als separate Erweiterung realisiert wurden, können diese einzeln aktualisiert werden. An den Schulen ist somit die langfristige Arbeit mit einer stabil arbeitenden ECLIPSE-Version möglich. Der Wechsel auf ein neues Release ist nur dann erforderlich, wenn durch eine Erweiterung spezifische Funktionen der neu zur Verfügung gestellten API unabdingbar werden. Findet dennoch der Umstieg auf eine neue Plattform-Version statt, lassen sich eigene Erweiterungen auf Grund der proklamierten Abwärtskompatibilität der API weiterhin verwenden.

Insgesamt und insbesondere im Vergleich mit der derzeit verwendeten DELPHI-IDE betrachtet, spricht somit vieles für den Einsatz von ECLIPSE an den sächsischen Schulen. Im Gegensatz zu ECLIPSE ist DELPHI weder frei verfügbar noch quelloffen. Somit sind hier eigene Änderungen oder gar Erweiterungen nicht möglich.

Auch im Vergleich mit LAZARUS scheint ECLIPSE zunächst klar im Vorteil zu sein: Das ECLIPSE-Projekt existiert seit vielen Jahren, weist eine große Community auf und wird zudem von zahlreichen namhaften Softwarefirmen unterstützt. Der Plugin-Mechanismus mit seinen klar definierten Schnittstellen ermöglicht prinzipiell die Erweiterung der gesamten Umgebung. LAZARUS hingegen wird nur von einer kleinen Community weiterentwickelt. Dennoch ist LAZARUS ebenso frei verfügbar und steht quelloffen zur Verfügung. **Ähnlich wie Eclipse ist auch hier ein Plugin-Mechanismus integriert.** Dieser wird in der LAZARUS-Terminologie als Package bezeichnet. Diese Packages gestatten nach erfolgreicher Installation (durch Kompilieren des Package) das Hinzufügen von Funktionen zur Entwicklungsumgebung.

Diese Funktionalität würde es - ähnlich wie bei ECLIPSE - letztendlich auch ermöglichen, alte Packages nach Installation einer neuen Version weiterhin zu verwenden. Anders als bei ECLIPSE ist diese Schnittstelle bei LAZARUS jedoch kaum dokumentiert. Darüber hinaus sind für grundlegende Änderungen an LAZARUS weiterhin Arbeiten an der eigentlichen IDE nötig.

Mit dem LAZARUS-Projekt haben sich seine Entwickler jedoch eine Nische gesucht, die mit diesem Projekt optimal ausgefüllt wird. In der finalen Version soll es möglich sein, **Delphi-Projekte adäquat in Lazarus-Projekte zu konvertieren.** Um diesen Anspruch gerecht zu werden, bietet LAZARUS bereits jetzt eine **gute *Pascal*-Unterstützung** an. Demzufolge steht

den Schulen natürlich schon jetzt eine gut entwickelte *Pascal*-Entwicklungsumgebung zur Verfügung. Da diese darüber hinaus den schulischen Belangen prinzipiell gut angepasst werden kann, ist der dortige Einsatz somit bereits möglich. **Für einen optimalen schulischen Einsatz sind jedoch noch einige Probleme bzw. Schwachpunkte abzustellen oder auszugleichen**.

Der Aufwand, Eclipse unter diesen Umständen in Schulen einzuführen, ist hingegen ungleich größer. Zunächst müsste hierfür eine entsprechende Erweiterung entwickelt werden. Diese ist dann in der Folge zu testen und gegebenenfalls weiter zu modifizieren. Trotz der relativ guten Dokumentation ist das nur mit einem erheblichen Mehraufwand an Entwicklungsarbeit zu realisieren. Vor diesem Hintergrund betrachtet der Autor den schulischen Einsatz von Eclipse als eine Gute, aber derzeit eher nur theoretische Option.

2.4 Analyse der Gebrauchstauglichkeit

2.4.1 Überblick zur Gebrauchstauglichkeit

Wie bereits früher dargelegt, wird der Begriff „Gebrauchstauglichkeit“ mit der Norm DIN EN ISO 9241 definiert. Dem zur Folge ist hierunter „das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufrieden stellend zu erreichen“ (vgl. Normenausschuss Ergonomie (NAErg) im DIN, Normenausschuss Informationstechnik und Anwendungen im DIN (2008)) zu verstehen. Für die nachfolgende Untersuchung ist dabei der Teil 110 von ausschlaggebender Bedeutung. Er befasst sich im Wesentlichen mit den Grundsätzen der Dialoggestaltung. Das Ziel der Norm besteht darin, durch entsprechende Vorgaben schon bei der Entwicklung typische Probleme der Benutzung von Benutzerschnittstellen zu erkennen und zu vermeiden. Teil 110 der DIN EN ISO 9241 benennt hierzu sieben Grundsätze, die durch Empfehlungen und Beispiele untersetzt werden:

- Aufgabenangemessenheit (A),
- Selbstbeschreibungsfähigkeit (SB),
- Erwartungskonformität (E),

- Lernförderlichkeit (L),
- Steuerbarkeit (ST),
- Fehlertoleranz (F) sowie
- Individualisierbarkeit (I).

Diese sieben Grundsätze wurden in einer früheren Belegarbeit des Autors bereits untersucht und dort ausführlich beschrieben. Es sei darum an dieser Stelle auf diese Quelle verwiesen. Im Weiteren erfolgt zunächst eine kurze Vorstellung der jeweiligen Empfehlungen sowie eine Untersuchung mit der Zielsetzung, die funktionale Umsetzung dieser Grundsätze bei der Entwicklungsumgebung ECLIPSE zu ermitteln.

Für den Grundsatz der Individualisierbarkeit führte der Autor diese Untersuchungen in o. g. Belegarbeit ebenfalls bereits durch. Die entsprechenden Ergebnisse werden hier deshalb im Weiteren nur kurz dargestellt. Analog zur damaligen Untersuchung erfolgt auch die Prüfung der jeweiligen Empfehlungen. Die Bewertung erfolgt nach einer Skala in tabellarischer Weise. Die Bewertungsskala umfasst hierbei die Werte GUT, BEFRIEDIGEND, UNGENÜGEND sowie KEINE UNTERSTÜTZUNG. Es sei darauf verwiesen, dass diese Arbeit, ähnlich wie die bereits vorgelegte Belegarbeit, im **Kontext des Schuleinsatzes potenzieller Entwicklungsumgebungen** steht. Aus Sicht des Autors müssen auch deshalb insbesondere die Empfehlungen im Fokus der Untersuchung stehen, die er als besonders wichtig für Schüler bzw. den Schuleinsatz einstuft. Diese Einordnung erfolgt unter Berücksichtigung der „Arbeitsaufgabe Programmierung“. Diese Betrachtungsweise kann bei den Empfehlungen hinsichtlich der Funktionalitäten natürlich zu Überschneidungen führen.

Aufgabenangemessenheit

> „Ein interaktives System ist aufgabenangemessen, wenn es den Benutzer unterstützt, seine Arbeitsaufgabe zu erledigen, d.h., wenn Funktionalität und Dialog auf den charakteristischen Eigenschaften der Arbeitsaufgabe basieren, anstatt auf der zur Aufgabenerledigung eingesetzten Technologie.“

Zur Aufgabenangemessenheit gibt die DIN EN ISO 9241 Teil 110 **sieben** Empfehlungen an:

- **A1 (Informationen zur Aufgabenerledigung):** „*Der Dialog sollte dem Benutzer solche Informationen anzeigen, die im Zusammenhang mit der erfolgreichen Erledigung der Arbeitsaufgabe stehen.*“

  Die geforderte Qualität, Quantität sowie Art der angezeigten Informationen wird dabei durch die Aufgabenerfordernisse bestimmt. In einem Beispiel hierzu führt die Norm das Dialogsystem eines Online-Shops an. Unter Berücksichtigung der o. g. Empfehlung sollte dieses dem Nutzer demgemäß eine kontextsensitive Hilfe anbieten, die alle erforderlichen Schritte zur Vervollständigung einer Bestellung beschreibt.

- **A2 (Relevante Informationen):** „*Der Dialog sollte dem Benutzer keine Informationen anzeigen, die nicht für die erfolgreiche Erledigung relevanter Arbeitsaufgaben benötigt werden.*“

  Empfehlung A2 verfolgt das Ziel, eine Verminderung der Arbeitsleistung durch unnötige mentale Belastung (z.B. Ausgabe überflüssiger Informationen) zu verhindern. Der Empfehlung folgend, sollten z.B. in einem Online-Ticket-System eines Stadions nur noch wirklich frei verfügbare Plätze dargestellt und zum Verkauf angeboten werden. Demzufolge werden alle weiteren, nicht zur eigentlichen Aufgabe - dem Kauf eines Tickets - gehörenden Informationen nicht oder erst auf Anfrage bereitgestellt.

- **A3 (Angepasste Ein-/Ausgabe):** „*Die Form der Eingabe und Ausgabe sollte der Arbeitsaufgabe angepasst sein.*“

  In der Norm wird an dieser Stelle auf eine Anwendung zur Umrechnung von Währungen verwiesen. Diese sollte die Genauigkeit automatisch der Zielwährung anpassen. Die entsprechende Untersuchung erfolgt bereits mehrfach bei anderen Empfehlungen.

- **A4 (Eingabewerte):** „*Wenn für eine Arbeitsaufgabe ganz bestimmte Eingabewerte typisch sind, sollten diese Werte dem Benutzer automatisch als voreingestellte Werte verfügbar sein.*“

  Hier ist zum Beispiel ein Verkehrsinformationssystem denkbar, über das Kunden Informationen über mögliche Verkehrsverbindungen, An- und Abfahrzeiten von Bussen und

Bahnen erhalten. Als Startpunkt fungiert der jeweilige aktuelle Standort/Haltestelle. Optional kann jedoch auch eine beliebige andere Haltestelle gewählt werden.

- **A5 (Dialogschritte):** „*Die vom interaktiven System verlangten Dialogschritte sollten zum Arbeitsablauf passen, d.h., notwendige Dialogschritte sollten enthalten sein und unnötige Dialogschritte sollten vermieden werden.*“
  Ein intelligentes Verkehrsinformationssystem kann zum Beispiel nach einer klar getroffenen Vorauswahl (wie Datum oder Verkehrsmittel) bereits bei der Eingabe von Start- und Zielbahnhof ohne weitere Zwischendialoge mögliche Verkehrsverbindungen vorschlagen.

- **A6 (Quelldokumente):** „*Wenn bei einer Arbeitsaufgabe Quelldokumente verwendet werden, sollte die Benutzungsschnittstelle kompatibel zu den charakteristischen Eigenschaften der Quelldokumente sein.*“
  Bei einer Entwicklungsumgebung ist die Arbeit mit Quelldokumenten der verwendeten Sprache essentiell und somit nicht nur für den Einsatz in Schulen von besonderer Bedeutung. Nach Auffassung des Autors muss von einer Entwicklungsumgebung grundsätzlich erwartet werden, dass die für die Programmierung charakteristischen Quelldokumente (Quelltexte, Bibliotheken, ...) und ihre Eigenschaften verarbeitet werden können. Eine gesonderte Betrachtung der funktionalen Umsetzung dieser Empfehlung erfolgt deshalb nicht.

- **A7 (Ein-/Ausgabemedien):** „*Die Eingabe- und Ausgabemedien des interaktiven Systems sollten aufgabenangemessen sein.*“
  Bei einfachen Anwendungen im Bereich der Grafikbearbeitung sind Eingaben über einfache Eingabegeräte wie Tastatur sowie Maus grundlegend. Die Unterstützung solcher grundlegenden Eingabegeräte wird auch für eine (schulische) Entwicklungsumgebung als zwingend notwendig angesehen und von Eclipse gewährleistet. Eine weitergehende Untersuchung wird deshalb als nicht erforderlich angesehen. Bei komplexeren Grafikanwendungen hingegen ist der Einsatz eines Tablet sinnvoll und sollte von der Anwendung bestmöglich unterstützt werden. Für einfache Programmierungsarbeiten sind Maus und Tastatur ausreichend.

Selbstbeschreibungsfähigkeit

> „Ein Dialog ist in dem Maße selbstbeschreibungsfähig, in dem für den Benutzer zu jeder Zeit offensichtlich ist, in welchem Dialog, an welcher Stelle im Dialog er sich befindet, welche Handlungen unternommen werden können und wie diese ausgeführt werden können."

Zur Selbstbeschreibungsfähigkeit gibt die DIN **sechs** Empfehlungen an:

- **SB1 (Nutzer leiten):** „*Die bei jedem Dialog angezeigten Informationen sollten den Benutzer leiten, den Dialog erfolgreich abzuschließen.*"
  Wird in einem Onlineshop der Einkauf von Artikeln durch einen aus mehreren Schritten bestehenden Dialog unterstützt, dann sollten dem Kunden auch Buttons mit den Funktionen *Zurück* und *Weiter* zur Verfügung stehen. Der Abschluss der Interaktion wird dem Nutzer durch einen Button *Kauf abschließen* angezeigt.

- **SB2 (Notwendigkeit externer Hilfe):** „*Während der Interaktion mit dem System sollte die Notwendigkeit, Benutzerhandbücher und andere externe Informationen heranzuziehen, minimiert sein.*"
  Die Umsetzung dieser Empfehlung innerhalb einer Software-Anwendung bedeutet, dass diese zum Beispiel Menüpunkte anbieten muss, deren Bezeichnung von den üblicherweise auszuführenden Benutzeraufgaben abzuleiten ist. Eine Software zur Veranlagung von Beiträgen sollte folglich Menüs und Funktionen wie *Zuordnung von Bemessungsgrundlagen*, *Veranlagung*, *Mahnung* oder *Abrechnung* enthalten.

- **SB3 (Zustandsänderungen):** „*Der Benutzer sollte über Änderungen des Zustandes des interaktiven Systems informiert werden, z.B. wann Eingaben erwartet werden oder durch Bereitstellung eines Überblickes über die nächsten Dialogschritte.*"
  Über die Druckverwaltung eines Betriebssystems sollte der Nutzer zum Beispiel jederzeit Informationen über den aktuellen Status eines Druckauftrages erhalten können. Somit wird für ihn ersichtlich, ob ein Drucker verfügbar ist, ein Druckauftrag versendet wurde, ob das Dokument überhaupt gedruckt werden kann und wie ggf. der Druckfortschritt ist.

- **SB4 (Erwartete Eingaben):** „*Wenn eine Eingabe verlangt wird, sollte das interaktive System dem Benutzer Informationen über die erwartete Eingabe bereitstellen.*“

  Das bereits als Beispiel benannte Verkehrinformationssystem erwartet für eine Verbindungsangabe u. a. die Eingabe des gewünschten Reisedatums. Damit die Eingabe durch den Nutzer korrekt erfolgt, gibt das System das gewünschte Format (TT|MM|JJ) als Hilfestellung vor. Alternativ kann das (bereits richtig formatierte) Datum auch aus einem vom System angebotenen Kalender ausgewählt werden. Eine Untersuchung der Umsetzung dieser Empfehlung innerhalb der Entwicklungsumgebung Eclipse wird ebenfalls als nicht erforderlich angesehen. Bei der „Arbeitsaufgabe Programmierung“ wird vordergründig auf die Entwicklung von Quelltext (durch Benutzereingabe) abgestellt. Die Überprüfung der korrekten Dateneingabe obliegt dabei dem Compiler und nicht der zu untersuchenden Benutzerschnittstelle. Darüber hinaus leistet die Syntaxhervorhebung eine Hilfestellung zur korrekten Eingabe. Diese ist bereits Gegenstand der Untersuchung zur Empfehlung ***F1***.

- **SB5 (Dialoginteraktion):** „*Dialoge sollten so gestaltet sein, dass die Interaktion für den Benutzer offensichtlich ist.*“

  Als praktikables Beispiel führt die Norm hierzu ein Softwarepaket zum Abspielen von DVDs an. Es stellt dem Anwender Schaltflächen mit Symbolen der „üblichen“ Tasten für *Abspielen*, *Stopp*, *Pause*, *schneller Vorlauf* usw. zur Verfügung.

- **SB6 (Formatinformationen):** „*Das interaktive System sollte dem Benutzer Informationen über die erforderliche Formate und Einheiten bereitstellen.*“

  Denkbar ist ein System zur Bestellung von Obst und Gemüse. Das System zeigt dem Nutzer an, in welchen Mengen und Einheiten Obst bestellt werden kann. Ihm wird zum Beispiel angezeigt, dass Äpfel mindestens kiloweise, Melonen hingegen stückweise zu ordern sind.

Erwartungskonformität

> „Ein Dialog ist erwartungskonform, wenn er den aus dem Nutzungskontext heraus vorhersehbaren Benutzerbelangen sowie allgemein anerkannten Konventionen entspricht.“

Zur Erwartungskonformität werden **neun** Empfehlungen angegeben. Hierbei wird angemerkt, dass Konsistenz grundsätzlich die Vorhersehbarkeit eines Dialoges erhöht.

- **E1 (Vokabular):** „*Das interaktive System sollte das Vokabular verwenden, das dem Benutzer bei der Ausführung der Arbeitsaufgabe vertraut ist oder von ihm auf Grund seiner Kenntnisse und Erfahrungen verwendet wird.*“

  Für die Programmierung existiert bereits eine gewohnte und vertraute Terminologie. So ist die Verwendung von Begriffen wie Compiler oder Debugger in entsprechenden Anwendungen die Regel.

- **E2 (Rückmeldung):** „*Auf Handlungen des Benutzers sollte eine unmittelbare und passende Rückmeldung folgen, soweit dies den Erwartungen des Benutzers entspricht.*“

  Wurde eine Anwendung installiert oder deinstalliert, erhält der Nutzer nach erfolgreicher Ausführung des Vorganges eine Rückmeldung (z.B. *„Installationsvorgang erfolgreich beendet“*).

- **E3 (Erwartete Antwortzeit):** „*Kann vorhergesehen werden, dass erhebliche Abweichungen von der vom Benutzer erwarteten Antwortzeit entstehen, sollte der Benutzer hiervon unterrichtet werden.*“

  Ein wider Erwarten länger andauernder Installationsprozess sollte dem Nutzer z.B. mit einer Ausgabe *„Dieser Vorgang kann einige Minuten dauern“* angezeigt werden. Im „klassischen Fall“ wird er dabei durch das System auch in geeigneter Weise über den Prozessfortschritt informiert. Beim Kopieren von großen Daten kalkuliert er so bereits eine längere Dauer ein. Eine Anzeige der voraussichtlichen Prozessdauer (geschätzte, sich aktualisierende Zeit) oder ein Fortschrittsbalken sind hierbei hilfreiche Informationen. Durch eine minimale Animation sollte dem Nutzer mindestens angezeigt werden, dass das System weiterhin arbeitet.

- **E4 (Informationsstrukturierung):** „*Informationen sollten so strukturiert und organisiert sein, wie es vom Benutzer als natürlich empfunden wird.*“

  Am Beispiel von Online-Shops ist deutlich erkennbar, wie wichtiges es ist, Informationen so zu strukturieren und organisieren, dass der Benutzer den Umgang mit diesen Informa-

tionen als natürlich empfindet. Ist im Warenangebot keine klare Struktur erkennbar, fällt dem Nutzer die Auswahl der Waren unnötig schwer. Angebote und Waren sollten folglich ähnlich gruppiert werden, wie es die Kunden aus den Abteilungen eines klassischen Kaufhauses gewohnt sind. Solche Gruppierungen können in Analogie zu „realen“ Warenhäusern beispielsweise Kategorien wie *Bücher*, *Musik*, *Filme*, *Textilien* oder *Möbel* sein.

- **E5 (Formatkonventionen):** „*Formate sollten geeigneten kulturellen und sprachlichen Konventionen entsprechen.*“

  Zur Verdeutlichung greift die Norm hier die Bündigkeit des Textes in unterschiedlichen Kultursprachen auf. So sollte eine Textverarbeitung in arabischer Sprache auch die Rechtsbündigkeit der Sprache umsetzen. Bei der Arbeit mit Währungen oder Mengen sollten die lokalen Verhältnisse unterstützt werden. In Europa also zum Beispiel der Euro. Da die Programmierung üblicherweise in englischer Sprache erfolgt, muss der Einfluss spezieller kultureller Eigenarten nicht gesondert betrachtet werden. Darüber hinaus folgt auch die Programmierung den Konventionen der jeweiligen Sprache. Eine Prüfung auf Korrektheit erfolgt auch hier durch den Compiler. Analog zu Empfehlung ***SB4*** wird Empfehlung ***E5*** deshalb im Weiteren ebenfalls nicht mehr untersucht.

- **E6 (Art von Rückmeldungen):** „*Art und Länge von Rückmeldungen oder Erläuterungen sollten den Benutzerbelangen entsprechen.*“

  Rückmeldungen oder Erläuterungen sollten in Abhängigkeit von der jeweiligen Arbeitsaufgabe unterschiedlich umfangreich dargestellt werden. Bei einfachen Veränderungen der Textformatierung reicht ein entsprechendes Symbol in der Textverarbeitung aus. Der markierte Text wird nach Betätigung des Symbolbuttons in der Folge entsprechend neu formatiert. Bei der Erstellung eines komplexen Serienbriefes hingegen, muss durch das System wesentlich mehr Hilfe (z.B. Handlungsabfolge) bereitgestellt werden.

- **E7 (Konsistentes Dialogverhalten):** „*Dialogverhalten und Informationsdarstellung eines interaktiven Systems sollten innerhalb von Arbeitsaufgaben und über ähnliche Arbeitsaufgaben hinweg konsistent sein.*“

  Diese Empfehlung geht davon aus, dass vergleichbare oder ähnliche Arbeitsaufgaben auch gleich ausgeführt werden sollten. Für einen Nutzer ähnelt der Vorgang des Speicherns

einer Datei dem des Öffnens. Ferner sollten häufig verwendete Buttons immer gleich positioniert werden und analoges Verhalten aufweisen. So sind die Buttons in Dialogen mit *„Ja“/“Nein“*-Rückfragen der Konvention des Betriebssystems entsprechend, stets gleich anzuordnen.

- **E8 (Eingabeposition):** „*Wenn eine bestimmte Eingabeposition auf der Grundlage von Benutzererwartungen vorhersehbar ist, dann sollte diese Position für die Eingaben voreingestellt sein.*“

  Im westlichen Sprachbereich positioniert eine Textverarbeitung den Cursor stets links oben. Die Bearbeitung des Dokuments wird traditionell dort begonnen. Erwartet ein Dialogsystem Eingaben, können diese in der Regel immer mit der Eingabetaste bestätigt werden. Zwischen einzelnen Eingabefeldern kann mit der Tabulator-Taste gewechselt werden.

- **E9 (Art der Informationen):** „*Mitteilungen, die dem Benutzer angezeigt werden, sollten in einer objektiven und konstruktiven Art formuliert sein.*“

  Mitteilungen an den Nutzer sollten stets unterstützend und hilfreich sein. Diese Empfehlung ist insbesondere bei Eingabefehlern durch den Nutzer bedeutsam. So sollte die Fehlermeldung bei Eingaben mit falschem Format präzise Hinweise auf das korrekte Format geben. Müssen für das Ausführen von Aufgaben zunächst Vorbedingungen erfüllt sein, so sollten diese auch genannt werden.

Lernförderlichkeit

> „Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen der Nutzung des interaktiven Systems unterstützt und anleitet.“

Zur Lernförderlichkeit benennt die DIN **sieben** Empfehlungen:

- **L1 (Regeln & Konzepte):** „*Regeln und zugrunde liegende Konzepte, die für das Erlernen nützlich sind, sollten dem Benutzer zugänglich gemacht werden. Dies erlaubt es dem Benutzer, sich eigene Ordnungsschemata und Merkregeln aufzubauen.*“

  Soll ein Anwender beispielsweise mit Hilfe eines Autorenwerkzeuges interaktive Medien

erstellen, benötigt er - sofern keine wesentlichen Vorkenntnisse vorhanden sind - eine entsprechende Hilfestellung. Es ist also sinnvoll, ihm entsprechende Dokumentationen und Informationen über das zu Grunde liegende Konzept des Autorenwerkzeuges zur Verfügung zu stellen.

- **L2 (Dialoghilfe):** „*Wenn ein Dialog selten gebraucht wird oder charakteristische Eigenschaften des Benutzers es erfordern, den Dialog erneut zu erlernen, dann sollte geeignete Unterstützung dafür bereitgestellt werden.*“
  Eine Anwendung zur Zertifizierung von Daten sollte ein Hilfe-System bereitstellen, das alle erforderlichen Schritten aufzeigt und prägnant beschreibt. Ist ein Nutzer nicht mit Makros einer Tabellenkalkulation vertraut oder arbeitet nur sehr selten mit ihnen, dann sollte das System eine geeignete Schritt-für-Schritt-Anleitung zur Verfügung stellen.

- **L3 (Dialogunterstützung):** „*Geeignete Unterstützung sollte bereitgestellt werden, damit der Benutzer mit dem Dialog vertraut wird.*“
  Dialoge mit wichtigen Einstellungen sind somit stets mit einem Hilfe-Button zu versehen, der über die möglichen Optionen unterrichtet. Zum jeweiligen Button oder der möglichen Einstellung können zusätzliche Informationen auch über Tooltipps vom System angeboten werden.

- **L4 (Umfang von Rückmeldungen):** „*Rückmeldung und Erläuterungen sollten den Benutzer unterstützen, ein konzeptionelles Verständnis vom interaktiven System zu bilden.*“
  Als Beispiel können wiederum verschiedene Arbeitsaufgaben bei einer Bild- oder Grafikbearbeitung gelten. Im einfachsten Falle kann ein (unerfahrener) Nutzer eine Tonwertkorrekturen an Bildern durch das System automatisch ausführen lassen. Die Anwendung hält für ihn zusätzlich eine Vorschau-Funktion bereit, aus der er die zu erwartenden Änderungen und das voraussichtliche Ergebnis entnehmen kann. Neben der Möglichkeit das „fertige Produkt“ zu sehen, wird dem Anfänger ein anschaulicher Eindruck über Sinn und Zweck der Option vermittelt. Dem Experten stehen darüber hinaus selbstverständlich zahlreiche weitere Optionen zur Verfügung um das aus seiner Sicht perfekte Ergebnis zu erzielen.

- **L5 (Rückmeldungen über Handlungsergebnisse):** „*Der Dialog sollte ausreichende Rückmeldung über Zwischen- und Endergebnisse von Handlungen bereitstellen, damit die Benutzer von erfolgreich ausgeführten Handlungen lernen.*“

  Als Beispiel sei erneut auf das erwähnte Online-Ticket-System verwiesen. Es ist unbedingt erforderlich, dem Nutzer eine Rückmeldung über seine ausgeführten Schritte und die entsprechenden Zwischenergebnisse zu geben (Plätze ausgewählt, Plätze reserviert, Zahlungsmethode, Bestellbestätigung erfolgt, ...). Als Zwischenergebnisse im Sinne der Programmierung können lauffähige Programmversionen angesehen werden. Das erfolgreiche Ausführen der Anwendung oder entsprechende Informationen des Compilers können dem zu Folge auch als Rückmeldung eines Handlungsergebnisses betrachtet werden. Diese Aufgaben werden somit nicht von der IDE-Oberfläche sondern vom Compiler erzeugt. Eine weitere Untersuchung erfolgt nicht.

- **L6 (Ausprobieren von Dialogschritten):** „*Falls es zu den Arbeitsaufgaben und den Lernzielen passt, sollte das interaktive System dem Benutzer erlauben, Dialogschritte ohne nachteilige Auswirkungen neu auszuprobieren. In sicherheitskritischen Systemen kann dies jedoch unpassend sein.*“

  Ein klassisches Beispiel hierfür sind die aus der Fotobearbeitung oder auch der Textverarbeitung bekannten *„Undo“*- sowie *„Redo“*-Funktionen. Mehrere ausgeführte Aktionen können über diese Funktionen rückgängig gemacht oder gegebenenfalls erneut ausgeführt werden.

- **L7 (Lernaufwand):** „*Das interaktive System sollte es dem Benutzer ermöglichen, die Arbeitsaufgabe mit minimalem Lernaufwand auszuführen, indem es den Dialog mit minimaler Eingabe von Informationen ermöglicht, jedoch zusätzliche Information auf Anforderung zur Verfügung stellt.*“

  Eine Anwendung zur Konvertierung von Musikdateien in unterschiedliche Formate sollte es dem Nutzer ermöglichen, diese Konvertierung auch ohne umständliche Konfiguration auf Basis von Standardwerten vorzunehmen. Auf Anforderung werden jedoch zusätzliche Optionen angeboten, um bestimmte Werte oder Dateieigenschaften ändern zu können.

Steuerbarkeit

> „Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist."

Zur Steuerbarkeit von Dialogen gibt die DIN **acht** Empfehlungen an:

- **ST1 (Interaktionsgeschwindigkeit):** „*Die Geschwindigkeit der Interaktion sollte nicht durch das interaktive System vorgegeben werden. Sie sollte vom Benutzer steuerbar sein, und zwar unter Berücksichtigung der Benutzerbelange und der charakteristischen Eigenschaften des Benutzers.*" *(Hierzu wird jedoch angemerkt, dass die Nutzung bestimmter interaktiver Systeme mit Auflagen verbunden sein kann, die eine Steuerung durch den Benutzer unter bestimmten Umständen nicht zulassen, z. B. Test-Situationen.)*
  Eine E-Mail-Anwendung bietet die Möglichkeit verschiedene Entwürfe von Nachrichten anzufertigen. Es ist nicht erforderlich, die Nachricht sofort zu versenden. Diesen Zeitpunkt kann der Nutzer nach eigenem Ermessen bestimmen. Die Nachricht bleibt zunächst als Entwurf gespeichert. Somit besteht jederzeit die Möglichkeit, Änderungen vorzunehmen.

- **ST2 (Fortsetzen des Dialoges):** „*Der Benutzer sollte die Steuerung darüber haben, wie der Dialog fortgesetzt wird.*"
  Beim Empfang einer neuen E-Mail-Nachricht muss der Nutzer über das weitere Vorgehen entscheiden können: Nachricht öffnen, speichern, löschen oder zunächst keine Aktion ausführen. Einige Textverarbeitungen öffnen automatisch den letzten bearbeiteten Vorgang. Als Alternative muss der Nutzer jedoch die Möglichkeit haben, nach eigenem Ermessen mit anderen Dokumenten beginnen zu können.

- **ST3 (Wiederaufnahme des Dialoges):** „*Ist der Dialog unterbrochen worden, sollte der Benutzer die Möglichkeit haben, den Wiederaufnahmepunkt der Fortsetzung des Dialoges zu bestimmen, falls es die Arbeitsaufgabe erlaubt.*"
  Die Arbeit an einem Dokument muss zu Gunsten der Bearbeitung eines anderen Dokumentes jederzeit unterbrochen werden können (z.B. aus Gründen der Dringlichkeit). Der

Nutzer kann den Zeitpunkt der Wiederaufnahme des Dialoges sowie die Fortsetzung der Arbeit am ursprünglichen Dokument selbstständig (durch Laden) festlegen.

- **ST4 (Reversibilität von Handlungen):** „*Wenigstens der letzte Dialogschritt sollte zurückgenommen werden können, soweit Handlungsschritte reversibel sind und falls es der Nutzungskontext erfordert.*“

  Als Beispiel für diese Empfehlung sei nochmals auf die bereits genannte *„Undo“*-Funktion verwiesen.

- **ST5 (Darstellung von Daten):** „*Wenn die Datenmenge, die für eine Arbeitsaufgabe von Bedeutung ist, groß ist, dann sollte der Benutzer die Möglichkeit haben, die Anzeige der dargestellten Datenmenge zu steuern.*“

  Eine Kalenderanwendung gestattet es dem Benutzer prinzipiell, nach eigenem Ermessen eine Auswahl zwischen den verschiedenen Kalenderoptionen zu treffen. Es kann sich dabei je nach Anforderung um Tagesansichten, Wochenansichten, Monatsansichten oder auch Jahresansichten handeln. Einzelne Ereignisse kann der Nutzer zu Gruppierungen zusammenfassen. Durch Ein- und Ausblenden der jeweiligen Gruppierung ist die Menge der anzuzeigenden Daten variabel. Angezeigte Datenmengen im Sinne der Programmierung können einerseits Quelltexte, andererseits jedoch auch die Menge an Compiler- oder Debuggerinformationen sein. Aus didaktischen Gründen sollte die Anzeige des Quelltextes nach Auffassung des Autors nicht reduziert werden. Die Menge der dargestellten Informationen wurde bereits im Rahmen der Individualisierbarkeit untersucht. Dem entsprechend erfolgt im Rahmen dieser Empfehlung keine gesonderte Untersuchung.

- **ST6 (Ein-/Ausgabemittel):** „*Der Benutzer sollte dort, wo es geeignet ist, die Möglichkeit haben jedes verfügbare Eingabe-/Ausgabemittel benutzen zu können.*“

  Die Ausführungen zu Empfehlung ***A7*** gelten hier analog.

- **ST7 (Voreinstellungen):** „*Wenn es für die Arbeitsaufgabe zweckmäßig ist, sollte der Benutzer voreingestellte Werte ändern können.*“

  Der Autor sieht die Möglichkeit, dass insbesondere Schüler mit wenig Erfahrung in Bezug auf Konfigurationen/Konfigurationseinstellungen Voreinstellungen ändern können, eher als

einen Nachteil an. Das Ändern von Werten sollte in diesem Falle den Lehrern vorbehalten sein. Ähnlich wie in Empfehlung ***ST5*** erfolgten hierzu bereits umfangreiche Untersuchungen im Rahmen der Individualisierbarkeit.

- **ST8 (Ändern von Daten):** „*Wenn Daten verändert wurden, sollten die Originaldaten für den Benutzer verfügbar bleiben, wenn dies für die Arbeitsaufgabe erforderlich ist.*“
  Mit einer Tabellenkalkulation können anhand konkreter Produktdaten des Unternehmens nach Einlesen der Originaldaten unterschiedlichste marktstrategische Simulationen gefahren werden. Im Ergebnis der Simulation entstehen neue Produktdaten. Die Originaldaten werden zuvor in einem gesonderten Tabellenblatt abgelegt und bleiben auf Dauer unverändert.

Fehlertoleranz

> „Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.“ Fehlertoleranz wird mit den Mitteln Fehlererkennung und -vermeidung (Schadensbegrenzung), Fehlerkorrektur oder Fehlermanagement (um mit Fehlern umzugehen, die sich ereignen) erreicht.

Zur Fehlertoleranz werden **zehn** Empfehlungen angegeben:

- **F1 (Eingabefehler):** „*Das interaktive System sollte den Benutzer dabei unterstützen, Eingabefehler zu entdecken und zu vermeiden.*“
  Wird in Kalkulationsprogrammen mit speziellen Werten gearbeitet, sollte das System verhindern, dass falsch strukturierte oder ungültige Daten eingegeben werden können. So kann z.B. bei der Eingabe des Datums der mögliche Wertebereich bereits entsprechend angepasst werden.

- **F2 (Undefinierte Systemzustände):** „*Das interaktive System sollte verhindern, dass irgendeine Benutzer-Handlung zu undefinierten Systemzuständen oder zu Systemabbrüchen führen kann.*“
  Ein einfaches Anwendungsbeispiel ist der Aufruf von nicht mehr existierenden URLs im

Internet. Der verwendete Browser sollte mit ungültigen Links angemessen umgehen können und eine entsprechende Fehlermeldung anzeigen. Die Suche nach dem Hostrechner darf dabei nicht in einer Endlosschleife enden.

- **F3 (Fehlerhilfe):** „*Wenn sich ein Fehler ereignet, sollte dem Benutzer eine Erläuterung zur Verfügung gestellt werden, um die Beseitigung des Fehlers zu erleichtern.*“
  So sollte eine Dateiverwaltung beim Versuch, auf einen vollen Datenträger zu schreiben, nicht nur auf einen ungültigen Schreibzugriff oder Schreibfehler verweisen. Vielmehr sollte auch einen Hinweis gegeben werden, dass die Speicherkapazität des Datenträgers ausgeschöpft ist und wie weiter verfahren werden kann.

- **F4 (Fehlerbeseitigung):** „*Aktive Unterstützung zur Fehlerbeseitigung sollte dort, wo typischerweise Fehler auftreten, zur Verfügung stehen.*“
  Fehler können in allen Phasen eines Handlungsprozesses auftreten. Aktive Fehlervermeidung sollte also bereits an der Quelle einsetzen; in einer Textverarbeitung zum Beispiel an der Stelle des Fehlers. Bei einem fehlerhaft geschriebenen Wort können über ein Kontextmenü z.B. mögliche Korrekturen direkt ausgewählt werden.

- **F5 (Automatische Fehlerkorrektur):** „*Wenn das interaktive System Fehler automatisch korrigieren kann, sollte es den Benutzer über die Ausführung der Korrektur informieren und ihm Gelegenheit geben, zu korrigieren.*“
  Dies trifft auch auf eine automatische Fehlerkorrektur in einem Kalkulationsprogramm zu. Wurde ein Wert automatisch korrigiert, wird dieser farblich hervorgehoben. Der Nutzer kann entscheiden, ob er diesen Korrekturvorschlag annimmt oder abweist.

- **F6 (Zurücksetzen der Fehlerkorrektur):** „*Der Benutzer sollte die Möglichkeit haben, die Fehlerkorrektur zurückzustellen oder den Fehler unkorrigiert zu lassen, es sei denn, eine Korrektur ist erforderlich, um den Dialog fortsetzen zu können.*“
  Beispielhaft sei nochmals eine Datenbank zur Mitgliederverwaltung genannt. Sind Daten partiell nicht bekannt, bleiben einzelne Felder zunächst leer. Die Anwendung sollte nicht auf der Korrektur bzw. dem Ausfüllen einzelner Felder bestehen.

- **F7 (Zusätzliche Fehlerinformationen):** „*Wenn möglich, sollten dem Benutzer auf An-*

frage zusätzliche Informationen zum Fehler und dessen Beseitigung zur Verfügung gestellt werden.“

An dieser Stelle sei auf die zusätzlichen Beispiele in Empfehlung ***F3*** verwiesen.

- **F8 (Datenprüfung):** „*Die Prüfung auf Gültigkeit und Korrektheit von Daten sollte stattfinden, bevor das interaktive System die Eingabe verarbeitet.*“

  Ein Online-Ticket-System wird in der Regel die benötigten Daten vor dem Senden auf Plausibilität (E-Mail-Adresse, korrekte Datenformate und ausgefüllte Felder, ...) überprüfen. Ebenso kann ein Internetbrowser vor Absenden einer Abfrage prüfen, ob z.B. das Format einer eingegebenen Adresse prinzipiell korrekt und gültig sein kann.

- **F9 (Fehlerbehebung):** „*Die zur Fehlerbehebung erforderlichen Schritte sollten minimiert sein.*“

  Die Korrektur eines falsch geschriebenen Wortes sollte in einer Textverarbeitung möglichst einfach sein. Beispielsweise könnte das System die korrekte Schreibweise bereits direkt nach der Eingabe vorschlagen.

- **F10 (Auswirkungen von Benutzerhandlungen):** „*Falls sich aus einer Benutzerhandlung schwerwiegende Auswirkungen ergeben können, sollte das interaktive System Erläuterungen bereitstellen und Bestätigung anfordern, bevor die Handlung ausgeführt wird.*“

  Ist beispielsweise das Löschen von Systemdaten nicht prinzipiell unterbunden, dann sollte bei einem Versuch eine Warnung ausgegeben werden. Diese informiert über möglichen Folgen des Vorganges oder weist darauf hin, dass es sich um eine Systemdatei handelt, die nicht gelöscht werden darf.

Individualisierbarkeit

> „Ein Dialog ist individualisierbar, wenn Benutzer die Mensch-System-Interaktion und die Darstellung von Informationen ändern können, um diese an ihre individuellen Fähigkeiten und Bedürfnisse anzupassen.“

Die Norm präsentiert für die Individualisierbarkeit **zehn** verschiedene Empfehlungen.

- **I1 (Benutzercharakteristik):** „*Das interaktive System sollte dem Benutzer dort, wo typischerweise unterschiedliche Benutzerbelange vorkommen, Techniken zur Anpassung an die charakteristischen Eigenschaften der einzelnen Benutzern bereitstellen.*“
  Dies können kulturelle Eigenschaften, individuelle Kenntnisse oder ebenso unterschiedliche Möglichkeiten zur Ausführung von Wahrnehmungsaufgaben, sensomotorischen und kognitiven Aktivitäten sein. In erster Konsequenz sollten für Nutzer mit Sehschwäche Möglichkeiten bestehen, größere Schriften zu verwenden oder eine kontrastreichere Farbgestaltung einzustellen. Auch sollten Optionen angeboten werden, das Erscheinungs- und Eingabeprofil an die kulturellen Gegebenheiten des Nutzers anzupassen. So sollte das interaktive System entsprechende Lokalisierungen unterstützen.

- **I2 (Darstellungsformen):** „*Das interaktive System sollte es dem Benutzer erlauben, zwischen verschiedenen Formen der Darstellung zu wählen, wenn es für die individuellen Bedürfnisse unterschiedlicher Benutzer zweckmäßig ist.*“
  Im Idealfall kann jeder Nutzer für sich ein entsprechendes Profil anlegen. Anwendungen zur Textverarbeitung arbeiten häufig mit Symbolleisten. Hierbei können sich nicht benötige Symbolleisten ausblenden bzw. wieder einblenden lassen. Weiterhin besteht die Möglichkeit, einzelne Symbole von der Symbolleiste zu entfernen oder aus einer Menge von Befehlen benötigte auszuwählen und zu beliebigen Symbolleisten hinzuzufügen.

- **I3 (Erläuterungen):** „*Der Umfang von Erläuterungen (z.B. Details in Fehlermeldungen, Hilfeinformationen) sollte entsprechend dem individuellen Kenntnisstand des Benutzers veränderbar sein.*“
  In einer möglichen Anwendung könnte der Nutzer den Detaillierungsgrad von Ausgaben an seine Situation anpassen. Befindet er sich noch beim Erlernen einer Anwendung, lässt er sich möglichst viele Details anzeigen. Fehlermeldungen könnten wesentlich weniger Informationen enthalten und Statusinformationen werden auf ein Minimum reduziert. Im Idealfall kann der Nutzer diese Einstellungen an seinen Erfahrungsstand anpassen und entsprechend verändern.

- **I4 (Eigenes Vokabular):** „*Benutzer sollten, soweit zweckmäßig, die Möglichkeit haben, eigenes Vokabular einzubinden, um Objekte und Funktionen („Werkzeuge“) individuell zu*

benennen.“

Als Beispiel können Makros in Tabellenkalkulationen angeführt werden. Sie geben dem Nutzer die Möglichkeit, komplexere und wiederkehrende Funktionen zusammenzufassen und über einen Befehl abzurufen bzw. auszuführen.

- **I5 (Ein-/Ausgabegeschwindigkeit):** „*Der Benutzer sollte, soweit zweckmäßig, die Möglichkeit haben, die Geschwindigkeit von dynamischen Eingaben und Ausgaben einstellen können, um sie an seine individuellen Belange anzupassen.*“

  Eine Möglichkeit wäre das Anpassen der Scrollgeschwindigkeit bei Browsern um so festzulegen, wie weit die Seite bei einem Klick gescrollt wird. Eine andere zeitrelevante Funktion bieten viele Suchmaschinenanbieter. Hier kann optional bestimmt werden, wie viel Suchergebnisse zu einem Zeitpunkt (auf einer Seite) angezeigt werden sollen.

- **I6 (Dialogtechniken):** „*Die Benutzer sollten, soweit zweckmäßig, die Möglichkeit haben, zwischen unterschiedlichen Dialogtechniken zu wählen.*“

  Bei der Verwaltung von Musik stehen in diversen Anwendungen unterschiedlichste Dialogtechniken zur Verfügung um bestimmte Titel zu suchen und auszuwählen. In einer ersten Variante kann dieser über verschiedene Filter (Genre, Künstler, Album) eingegrenzt und aus der nachfolgend erstellten folgenden Liste ausgewählt werden. In einer anderen Variante kann der gewünschte Titel, der Künstler oder ggf. der Titel eines Albums manuell in ein Suchfeld eingeben werden.

- **I7 (Interaktion):** „*Der Benutzer sollte die Möglichkeit haben, das Niveau und die Methoden der Mensch-System-Interaktion so auszuwählen, dass sie am besten seinen Bedürfnissen entsprechen.*“

  In Umsetzung dieser Empfehlung bieten viele Programme Anwendern die Möglichkeit, Funktionen aus einem Menü auszuwählen oder über Kurzwahltasten aufzurufen.

- **I8 (Ein-/Ausgabedarstellung):** „*Der Benutzer sollte die Möglichkeit haben, die Art zu wählen, in der Eingabe-/Ausgabedaten dargestellt werden (Format und Typ).*“

  Für die Umsetzung sei auf das bereits beschriebene Beispiel einer Mindestgröße der Schriftarten zur Darstellung von Texten verwiesen. Ebenso erlauben Anwendungen zur Musik-

verwaltung Alben als reine Liste darzustellen oder, angelegt an eine klassische Plattensammlung, durch ihr CD-Cover dargestellt zu navigieren.

- **I9 (Dialogelemente):** „*Soweit zweckmäßig, sollte es den Benutzern möglich sein, Dialogelemente oder Funktionen hinzuzufügen oder neu zu ordnen, insbesondere um individuelle Bedürfnisse bei der Ausführung von Arbeitsaufgaben zu unterstützen.*“
  Hier sei wiederum auf das Beispiel diverser Anwendungen zur Textverarbeitung verwiesen. Nutzer können sich hier Symbolleisten und die enthaltenen Befehle individuell anpassen.

- **I10 (Dialogeinstellungen):** „*Individuelle Einstellungen eines Dialoges sollten rückgängig gemacht werden können und es dem Benutzer erlauben, zu den ursprünglichen Einstellungen zurückzugehen.*“
  So können in Textverarbeitungsanwendungen die veränderten Symbolleisten oftmals im selben Dialog auf den Standard-Zustand zurückgesetzt werden.

2.4.2 Funktionale Umsetzung der Gebrauchstauglichkeit in Eclipse

Die Untersuchungen zur Gebrauchstauglichkeit erfolgen auf Basis der ECLIPSE-Version 3.5.0 unter Windows XP. Eine Ausnahme bildet die Untersuchung zur Individualisierbarkeit. Diese wurde bereits zu einem früheren Zeitpunkt für die Version 3.4.1 durchgeführt. Im Mittelpunkt der weiteren Untersuchungen steht vordergründig die Arbeitsaufgabe „Programmieren“ und die in diesem Zusammenhang von ECLIPSE angebotene funktionale Umsetzung der Empfehlungen. Nicht untersuchte Empfehlungen werden nicht erwähnt. Hierfür sei auf die bereits vorgestellten Begründungen verwiesen. Eine Gesamtübersicht der Ergebnisse (inklusive der Individualisierbarkeit) kann Tabelle 12 entnommen werden.

Aufgabenangemessenheit

Begründung A1 (Informationen zur Aufgabenerledigung)

- Bei der Entwicklung von *Java*-Applikationen mit ECLIPSE wird die *Java*-Perspektive standardmäßig zur Verfügung gestellt und angewendet. Sie stellt dem Nutzer nur die für die Aufgabe benötigten Funktionen und Informationen zur Verfügung. Ihm steht zur Projekt-

Empfehlung	Bewertung
A1 (Inf. zur Aufgabenerledigung)	gut
A2 (Relevante Informationen)	gut
A4 (Eingabewerte)	gut
A5 (Dialogschritte)	gut

Tabelle 6: Bewertung der Aufgabenangemessenheit

übersicht neben dem Hierarchie-Browser ein Package-Explorer zur Verfügung. Unter dem zentralen *Java*-Editor befinden sich Fenster, in dem alle nötigen Informationen, z.B. zu Fehlern bei der Kompilierung, dargestellt werden.

- Der Editor zeigt nur den reinen Quellcode - erweitert um die Visualisierung von Fehlern oder zum Beispiel von Haltepunkten - an.
- Soll ein neues *Java*-Projekt erzeugt werden, stellt ECLIPSE hierfür einen Wizard zur Verfügung. Dieser verlangt nur die für die Erstellung nötigen Informationen, wie Name, *Java-Runtime-Environment* oder die Angabe, welche zusätzlichen Projekte oder Bibliotheken eingebunden werden sollen.

Begründung A2 (Relevante Informationen)

- Die *Java*-Perspektive stellt die für die Programmierung relevante Werkzeuge und Informationen bereit. Informationen, die für die reine Programmierung nicht erforderlich sind, werden entweder ausgeblendet oder in andere Perspektiven verlagert. So zum Beispiel die Team-Synchronisation. Während der Programmierung ist keine ständige Information über ein entferntes Repository nötig.
- Darüber hinaus hat der Nutzer die Möglichkeit, die IDE-Oberfläche an seine spezifischen Bedürfnisse anzupassen. Nicht benötigte Informationen oder Werkzeuge können über diesen Weg somit auch ausgeblendet werden.

Begründung A4 (Eingabewerte)

- Beim Anlegen einer Klasse schlägt ECLIPSE bereits typische Werte vor. So wird zunächst der Name auf korrekte Syntax überprüft. Des Weiteren ist zum Beispiel eine Auswahl

möglich ob es sich um eine öffentliche (*public*) oder abstrakte Klasse handelt. Für unterschiedliche Typen (Klasse, Interface, Package, ...) werden dann verschiedene Menüs mit unterschiedlichen charakteristischen Möglichkeiten angeboten.

- Beim Anlegen einer neuen Klasse wird bereits das Grundgerüst (z. B. Konstruktor) erstellt.

Begründung A5 (Dialogschritte)

- Das Anlegen eines *Java*-Projektes über den Wizard beschränkt sich im einfachsten Fall auf das Eintragen eines Namens sowie dessen Bestätigung. Ähnliches gilt auch für Typen wie Klassen oder auch Interfaces. Unnötige Dialogschritte wurden somit vermieden. Zusätzliche Schritte sind jedoch stets verfügbar.

- Beim Debuggen ist schrittweise das Schalten von einem Haltepunkt zum nächsten möglich. Zusätzlich werden beim Debuggen Informationen wie aktuelle Variablenbelegungen angezeigt.

- Häufig benötigte Dialoge können über Symbolbuttons schnell ausgeführt werden.

Selbstbeschreibungsfähigkeit

Empfehlung	**Bewertung**
SB1 (Nutzer leiten)	gut
SB2 (Notwendigkeit externer Hilfe)	gut
SB3 (Zustandsänderungen)	befriedigend
SB5 (Dialoginteraktion)	gut
SB6 (Formatinformationen)	befriedigend

Tabelle 7: Bewertung der Selbstbeschreibungsfähigkeit

Begründung SB1 (Nutzer leiten)

- Für das erfolgreiche Anlegen eines neuen Projekts ist im Wizard im einfachen Fall ein einzelner Schritt nötig. Um den Dialog zu beenden ist lediglich ein Projektname anzugeben. Im Wizard wird darauf direkt (unterhalb der Überschrift) hingewiesen. Ist der Name ungültig, erhält der Nutzer eine zusätzlich Information.

- Für das Anlegen von Klassen gelten o. g. Ausführungen analog.

- In ECLIPSE besteht jederzeit die Möglichkeit, den aktuellen Workspace zu wechseln. Hierbei kann aus den bisher verwendeten Workspaces gewählt werden. Alternativ kann über einen zusätzlichen Dialog ein neuer Workspace erstellt werden. Dieser Dialog beschreibt in kurzer Form, worum es sich beim Workspace handelt und was als Eingabe erwartet wird.

Begründung SB2 (Notwendigkeit externer Hilfe)

- Verschiedene Funktionen lassen sich in ECLIPSE über das Menü, über Kontextmenüs oder auch Symbole aufrufen. Das Menü ist dabei logisch gegliedert. Funktionen mit Relevanz für ein Projekt befinden sich im *Project*-Menü. Funktionen zum Starten bzw. Testen der Anwendung sind im Menüpunkt *Run* eingeordnet.
- Für die Symbole wurden weitgehend einfache, selbstbeschreibende Darstellungen gefunden. Das Starten einer Anwendung erfolgt mit Klick auf eine *Play*-Taste - ähnlich den Medienplayern. Für das Anlegen neuer Packages oder Klassen nutzt ECLIPSE die häufig verwendeten Symbole mit einem leuchtenden Plus.
- Über Kontextmenüs können für markierte Dateien, markierte Ausdrücke oder Zeilen im Quelltext sinnvolle Funktionen ausgewählt werden.
- In jedem Wizard ist unter der Überschrift eine kurze Information enthalten, wozu dieser Wizard dient. Es erfolgt ferner ein Verweis auf die Hilfe.

Begründung SB3 (Zustandsänderungen)

- Die Bereitstellung nachfolgender Dialogschritte erfolgt in Wizards wie bereits in mehreren Beispielen beschrieben. Über die Symbolleiste ist eine Auswahl der potenziellen und in der Programmierung häufig verwendeten Dialogschritte möglich.
- Es gibt dabei jedoch keine Angabe über die Anzahl folgender Dialogschritte. Nur der *Next*-Button zeigt weitere mögliche Dialogschritte an.
- Wird eine Eingabe erwartetet, wird das über einen blinkenden Cursor signalisiert. Der Cursor erscheint auch am Ende eines bereits markierten Wertes.

Begründung SB5 (Dialoginteraktion)

- Die ECLIPSE-Oberfläche ist in mehrere Teile gegliedert. Speziell auf die *Java*-Perspektive bezogen gilt: Zur Anwendungsentwicklung wird dem Nutzer ein zentral positionierter Editor angeboten. Funktionen (Compiler, Debugger, ...) können ebenfalls über gut positionierte und klar erkennbare Symbole oder aus Menüs aufgerufen werden. Das Projekt ist über Wizards um weitere Klassen erweiterbar. Die Interaktion zur Programmentwicklung ist somit einfach und übersichtlich gestaltet.

Begründung SB6 (Formatinformationen)

- Der allgemeine Dialog zum Öffnen von Dokumenten suggeriert dem Nutzer zunächst, jeden möglichen Dateityp öffnen zu können. Wird z.B. versucht, eine *MP3*-Datei zu öffnen, startet in der Folge die dafür im System registrierte Anwendung.
- Zu einem Projekt können zusätzliche *JARs* hinzugefügt werden. Der mögliche Dateityp wird dabei als einzige Information zusätzlich ausgegeben.
- Eingaben in Dialogen werden durch Zusatzinformationen, wie Tooltipps, unterstützt.

Erwartungskonformität

Empfehlung	**Bewertung**
E1 (Vokabular)	gut
E2 (Rückmeldung)	gut
E3 (Erwartete Antwortzeit)	befriedigend
E4 (Informationsstrukturierung)	gut
E6 (Art von Rückmeldungen)	befriedigend
E7 (Konsistentes Dialogverhalten)	gut
E8 (Eingabeposition)	gut
E9 (Art der Informationen)	befriedigend

Tabelle 8: Bewertung der Erwartungskonformität

Begründung E1 (Vokabular)

- ECLIPSE arbeitet komplett mit einer Terminologie, wie sie auch für die Programmierung erwartet wird. Insbesondere für *Java* können neue Klassen, Packages, Interfaces oder auch Annotationen angelegt werden.

- Zur Bearbeitung der in den Projekten organisierten Dateien, stehen Editoren zur Verfügung. Ferner kommen ein Compiler sowie ein Debugger zum Einsatz.

Begründung E2 (Rückmeldung)

- ECLIPSE reagiert mit geeigneten Rückmeldungen unmittelbar auf die verschiedenen Handlungen des Nutzers. Im klassischen Fall ruft er einen Dialog auf, der vom System unmittelbar dargestellt wird. So wird z.B. eine Texteingabe durch das System visualisiert und grafisch angezeigt (Cursor).

- Ausgewählte Prozesse werden durch eine Fortschrittsanzeige im unteren rechten Bereich der Anwendung dargestellt. Das erfolgt beispielsweise beim Kompilieren oder bei der Erstellung eines WorkingSet.

- Erfolgt ein Wechsel der Perspektive, fragt das System zunächst nach. Nur wenn der Nutzer diesen Dialog zuvor abstellt, wird seine Entscheidung auch in der Folge automatisch durchgeführt.

- Treten Fehler auf, werden diese über eine Dialogbox oder die Konsole ausgegeben. Diese entsprechen der Handlung des Nutzers. Fehler mit dem Dialogsystem kommen in der Dialogbox, Programmierfehler über die Konsole zur Anzeige/Ausgabe.

Begründung E3 (Erwartete Antwortzeit)

- Es wurde keine spezielle Meldung gefunden. Jedoch verwendet ECLIPSE für lang andauernde Prozesse einen Fortschrittsbalken.

Begründung E4 (Informationsstrukturierung)

- Innerhalb einer IDE wird mit verschiedenen Informationen gearbeitet. Aus Sicht der Schüler sind für sie hierbei besonders die Daten interessant, mit denen sie arbeiten (z.B. der Quelltext oder eventuelle Fehlerausgaben). Der Quelltext im Editor wird entsprechend der Syntax der Sprache organisiert und hervorgehoben. Fehlermeldungen werden in der Konsole dargestellt.

- Beim Debuggen der Anwendung werden die Variablenbelegungen übersichtlich dargestellt, Haltepunkte entsprechend visualisiert.
- Die Dateien werden über Projekte organisiert und durch den PackageExplorer übersichtlich hierarchisch dargestellt. Auch die Anzeige einer Typen-Hierarchie ist möglich. Allgemein ist die Oberfläche der Entwicklungsumgebung entsprechend der Arbeitsaufgabe gegliedert (Quelltexteditor, Konsole, PackageExplorer, ...).

Begründung E6 (Art von Rückmeldungen)

- In ECLIPSE stehen mehrere Erläuterungen zur Verfügung. Symbole enthalten einen Tooltip. Dieser informiert jedoch nur, was hier möglich ist. Er enthält keine Information über die Verwendung.
- Wizards enthalten unterhalb ihrer Kopfzeile jeweils eine kurze Erläuterung.
- Ausführliche Erläuterungen stehen jeweils erst in der Hilfe zur Verfügung.
- In Anbetracht des Schuleinsatzes haben Systemrückmeldungen in Form von Fehlermeldungen hier eine besondere Bedeutung. Fehlerausgaben des Compilers sowie des Editors stehen dabei im Vordergrund. Im Editor werden entsprechende Fehler gekennzeichnet und mit einer kurzen Information versehen ausgegeben. Eine falsche Typenkonvertierung wird zum Beispiel direkt an der jeweiligen Zeile des Editors angezeigt und gibt Aufschluss über die Art des Fehlers.
- Kompilierungsfehler werden in der Konsole ausführlich dargestellt. Diese Ausgaben geben Aufschluss über die gesamte Fehlerkette, so über die Art und die Stelle des Fehlers. Sie sind aber für Schüler ohne oder mit nur geringer Kenntnis teilweise zu lang und zu umständlich.

Begründung E7 (Konsistentes Dialogverhalten)

- Das Dialogverhalten sowie die Informationsdarstellung gestalten sich über das gesamte System hinweg konsistent. Alle Wizards sind gleich aufgebaut. Die Navigation und Bedienung erfolgt jeweils in identischer Weise.

- Es stehen insgesamt drei unterschiedliche Konsolen zur Verfügung (Process, Stacktrace, CVS), deren Darstellung von Informationen ebenfalls konsistent ist.

Begründung E8 (Eingabeposition)

- Gemäß der Windows-Konvention für Benutzerschnittstellen stellt auch ECLIPSE die gewohnten Eingabemöglichkeiten zur Verfügung. So erfolgt z.B. auch hier die Bestätigung eines Dialoges mit der Enter-Taste oder der Wechsel zwischen relevanten Elementen mittels Tabulatortaste.
- Ist die genaue Angabe von Speicherorten oder von Verzeichnisse erforderlich, müssen diese durch den Nutzer nicht eingeben werden. Er kann sie stets über den Windows-Explorer auswählen.
- Erwartet die IDE Benutzereingaben, wird der Corsor automatisch in das betreffende Feld gesetzt.

Begründung E9 (Art der Informationen)

- Systemmitteilungen, die für den Benutzer hilfreich und unterstützend sind, stehen wie in Empfehlung ***E6*** aufgeführt, in größerem Umfang zur Verfügung.
- Für den schulischen Einsatz wurde in den Fehlerausgaben des Compilers jedoch ein Problem erkannt. Die Compilerausgaben geben zwar Aufschluss über die Fehlerkette, sind aber nur für den fortgeschrittenen Anwender eine wirkliche Hilfe.

Lernförderlichkeit

Empfehlung	**Bewertung**
L1 (Regeln & Konzepte)	gut
L2 (Dialoghilfe)	gut
L3 (Dialogunterstützung)	gut
L4 (Umfang von Rückmeldungen)	befriedigend
L6 (Ausprobieren von Dialogschritten)	gut
L7 (Lernaufwand)	gut

Tabelle 9: Bewertung der Lernförderlichkeit

Begründung L1 (Regeln & Konzepte)

- Um dem Nutzer den Einstieg zur erleichtern, stellt ECLIPSE ein umfangreiches Hilfepaket bereit. So ist in ECLIPSE u. a. ein Guide speziell zum Workbench, zur *Java*-Entwicklung oder auch zur Plug-In-Entwicklung enthalten. Im Standardpaket steht diese Hilfe allerdings nur in englischer Sprache zur Verfügung.

Begründung L2 (Dialoghilfe)

- Hier steht eine Unterstützung dergestalt zur Verfügung, dass zum Beispiel in Wizards eine kurze Erläuterung gegeben wird. Darüber hinaus steht überall ein zusätzlicher Button zur Verfügung, über den eine kontextsensitive Hilfe aufgerufen und einblendet werden kann.

Begründung L3 (Dialogunterstützung)

- Hierfür steht der bereits in Empfehlung ***L2*** benannte Button zur Verfügung. Die Symbole der Benutzerschnittstelle sind mit jeweils einen Tooltip und einer kurzen Beschreibung verbunden.

Begründung L4 (Umfang von Rückmeldungen)

- Umfangreiche Erläuterungen, die ein konzeptionelles Verständnis des Systems fördern, sind überwiegend nur über das Hilfe-Menü zu erreichen.
- Rückmeldungen sind in großem Unfang vorhanden (siehe Empfehlung ***E6***).

Begründung L6 (Ausprobieren von Dialogschritten)

- Für das Testen und die Arbeit mit diversen Dialogschritten bietet ECLIPSE eine umfangreiche *„Redo“-/“Undo“*-Funktion an. Hiermit ist es möglich, Änderungen im Quelltext ebenso wie ein Refactoring zurückzunehmen.
- Des Weiteren können durch den Benutzer verschiedene Einstellungen „gefahrlos“ ausprobiert werden. ECLIPSE bietet mit o. g. Funktionen die Möglichkeit, den Ausgangszustand jederzeit wieder herzustellen.

Begründung L7 (Lernaufwand)

- Der Lernaufwand für das Erstellen einer einfachen kleinen *Java*-Anwendung mittels ECLIPSE ist tatsächlich minimal. Für das Erstellen eines entsprechenden Projektes kann in einem ersten Schritt auf einen einfach zu bedienenden Wizard zurückgegriffen werden.
- Nach dem Verfassen des Quelltextes kann die Anwendung über die *„Run“*-Schaltfläche gestartet werden. Für eine entsprechende Laufkonfiguration stehen bereits voreingestellte Werte zur Verfügung. Diese können bei Bedarf ebenfalls auch verändert und angepasst werden. Bei einfachen Anwendungen ist das jedoch nicht nötig.

Steuerbarkeit

Empfehlung	**Bewertung**
ST1 (Interaktionsgeschwindigkeit)	gut
ST2 (Fortsetzen des Dialoges)	gut
ST3 (Wiederaufnahme des Dialoges)	gut
ST4 (Reversibilität von Handlungen)	gut
ST6 (Ein-/Ausgabemittel)	gut
ST7 (Voreinstellungen)	gut
ST8 (Ändern von Daten)	gut

Tabelle 10: Bewertung der Steuerbarkeit

Begründung ST1 (Interaktionsgeschwindigkeit)

- Über die Einstellungen kann der Nutzer festlegen, welche Plug-Ins bereits bei Anwendungsstart geladen werden und in der Folge dann sofort zur Verfügung stehen.
- Nachfragende Dialoge können abgestellt und gegebenenfalls über die Einstellungen erneut aktiviert werden.
- Beim Debuggen ist nach eigenem Ermessen ein Springen von Haltepunkt zu Haltepunkt möglich.
- Das Speichern von Bearbeitungsständen kann jederzeit manuell erfolgen.
- Diese Möglichkeiten sind nach Ansicht des Autors für schulische Belange ausreichend.

Begründung ST2 (Fortsetzen des Dialoges)

- Dialoge können prinzipiell bestätigt oder abgebrochen werden. Letztere Option besteht ebenfalls für laufende Prozesse.
- In den verschiedenen Wizards kann ein Anwender, sofern er hierfür Bedarf sieht, optional weitere Schritte ausführen.

Begründung ST3 (Wiederaufnahme des Dialoges)

- Diese Empfehlung wird durch das ECLIPSE-Konzept des Workspace wirksam umgesetzt. Es wurde im Verlauf dieser Arbeit bereits vorgestellt.
- Ein Workspace kann mit all seinen Einstellungen und Projekten gespeichert und erneut geladen werden. Ebenso können einzelne Projekte exportiert werden. Die Unterbrechung und spätere Wiederaufnahme der Arbeitsaufgabe ist mit ECLIPSE also jederzeit möglich.

Begründung ST4 (Reversibilität von Handlungen)

- Siehe Empfehlung ***L6***

Begründung ST6 (Ein-/Ausgabemittel)

- Quelltext wird bei ECLIPSE grundsätzlich über die Tastatur eingegeben. Zusätzlich kann eine Screen-Reader-Software sowie Spracherkennung zum Einsatz kommen.

Begründung ST7 (Voreinstellungen)

- Siehe Empfehlung ***I10***

Begründung ST8 (Ändern von Daten)

- Wie bereits erläutert, bietet ECLIPSE die Option, veränderte Einstellungen auf Standardwerte zurückzusetzen. Wird ein Projekt in einen Workspace importiert, besteht ferner die Möglichkeit, die eingegebenen Daten mit zu kopieren. Die Originaldaten bleiben in diesem Fall erhalten.

Empfehlung	Bewertung
F1 (Eingabefehler)	gut
F2 (Undefinierte Systemzustände)	gut
F3 (Fehlerhilfe)	gut
F4 (Fehlerbeseitigung)	gut
F5 (Automatische Fehlerkorrektur)	keine Unterstützung
F6 (Zurücksetzen der Fehlerkorrektur)	befriedigend
F7 (Zusätzliche Fehlerinformationen)	gut
F8 (Datenprüfung)	gut
F9 (Fehlerbehebung)	befriedigend
F10 (Auswirkungen von Benutzerhandlungen)	gut

Tabelle 11: Bewertung der Fehlertoleranz

Fehlertoleranz

Begründung F1 (Eingabefehler)

- Insbesondere unter dem Aspekt der Programmierung kommt bei ECLIPSE entsprechend der Sprache *Java* eine Syntaxhervorhebung zum Einsatz. Die Fehlerwahrscheinlichkeit wird hierdurch erheblich reduziert.
- Weiterhin ist eine Funktion „Vervollständigung“ verfügbar. Funktionen oder Variablen können durch den Nutzer so korrekt eingesetzt und verwendet werden.

Begründung F2 (Undefinierte Systemzustände)

- Eine mögliche Folge der fehlerhaften Programmierung einer Anwendung kann ein Deadlock sein. Der ECLIPSE-Debugger ermöglicht es, diese Anwendungen versuchsweise auszuführen und bei kritischen Systemzuständen sofort zu unterbrechen.
- Wurden falsche Dateitypen geöffnet, werden die dafür im System registrierten Anwendungen gestartet.

Begründung F3 (Fehlerhilfe)

- Beim Auftreten von Syntaxfehler steht dem Nutzer eine Hilfe zur korrekten Verwendung der jeweiligen Funktionen zur Verfügung. Auch in Wizards wird auf eventuelle Fehler hingewiesen.

- Weiterhin werden Compilerausgaben als Rückmeldung ausgegeben.

Begründung F4 (Fehlerbeseitigung)

- Eclipse stellt Unterstützung zur Fehlerbeseitigung auch dort bereit, wo die Fehler typischerweise gemacht werden. Diese werden direkt an Ort und Stelle visualisiert. Je nach Einstellung erfolgt das durch einen roten Strich im Editor.
- Des Weiteren werden Verweise am Rand des Editors angezeigt. Treten Fehler bei der Ausführung (beispielsweise eine *NullPointerException*) auf, erfolgt deren Anzeige zunächst in der Konsole. Darüber hinaus werden die verantwortlichen/beteiligten Klassen als Link aufgezeigt. Somit ist ein direkter Sprung an die betroffene Stelle möglich.

Begründung F5 (Automatische Fehlerkorrektur)

- Eine vollständige Auto-Korrektur von Fehlern konnte nicht festgestellt werden. Eine Korrektur findet erst nach Nutzerinteraktion (Klicken auf Verweise um Vorschläge auszuwählen) statt.

Begründung F6 (Zurücksetzen der Fehlerkorrektur)

- Eine Auto-Korrektur wurde nicht festgestellt. Dem zur Folge ist eine Rücknahme nicht möglich.
- Fehler im Quelltext können unkorrigiert bleiben, führen in der Folge jedoch möglicherweise zu einer fehlerhaften Anwendung.
- Ein Wizard kann hingegen erst bei Verwendung eines Namens gemäß der *Java*-Namenskonvention fortgesetzt werden.

Begründung F7 (Zusätzliche Fehlerinformationen)

- Eclipse stellt spezielle themenabhängige Guides und entsprechende *API*-Referenzen zur Verfügung. Zusätzliche Hilfe wird über die Website bereitgestellt.

Begründung F8 (Datenprüfung)

- Neben der erwähnten Syntaxhervorhebung findet parallel eine Prüfung des Quelltextes statt. Potenzielle Fehler werden (soweit nicht anders eingestellt) rot markiert und am linken Editor-Fenster angezeigt. Wird die Anwendung trotz vorhandener Fehler gestartet, erfolgt durch das System eine erneute Nachfrage.

Begründung F9 (Fehlerbehebung)

- Wird z.B. ein Funktionsaufruf falsch geschrieben, wird dieser markiert. ECLIPSE schlägt nach *Links*-Klick auf den Fehler bereits mögliche Korrekturen vor und stellt diese zur Auswahl. Die Korrektur findet dann entsprechend der getroffenen Wahl automatisch statt.

Begründung F10 (Auswirkungen von Benutzerhandlungen)

- Da sich durch das Löschen von Dateien aus dem Projekt gegebenenfalls schwerwiegende Auswirkungen ergeben können, erfolgt durch das System zunächst eine Sicherheitsabfrage. Die Anfrage ist durch den Nutzer zu quittieren und wird erst dann ausgeführt.
- Ebenso erfolgt eine Abfrage, falls fehlerhafter Code ausgeführt werden soll.

Individualisierbarkeit

Auf die Individualisierbarkeit wird, wie erwähnt, nicht gesondert eingegangen. Die ausführliche Untersuchung kann in der Belegarbeit des Autors nachgelesen werden.

2.4.3 Zusammenfassung

Die Untersuchung zur funktionalen Umsetzung der Grundsätze ergab ein **überwiegend positives Ergebnis**. Insbesondere die Empfehlungen zur **Steuerbarkeit** sowie zur **Lernförderlichkeit** finden eine **gute funktionelle Unterstützung**. Ebenso konnten die zur **Aufgabenangemessenheit** untersuchten Empfehlungen mit GUT bewertet werden. Lediglich bei **Individualisierbarkeit und Fehlertoleranz wurden in wenigen Fällen schlechtere Wertungen vorgenommen.** Insgesamt sieht der Autor das Ergebnis der Analyse als positiv an. Die Untersuchung zeigt somit, dass der Spagat zwischen einer umfangreichen, funktionellen IDE und den Grundsätzen der Gebrauchstauglichkeit möglich und sinnvoll realisiert werden kann.

Tabelle 12: Bewertung der Gebrauchstauglichkeit von ECLIPSE

Grundsatz	Empfehlung		Bewertung
Aufgabenangemessenheit	A1	(Inf. zur Aufgabenerledigung)	gut
	A2	(relevante Informationen)	gut
	A4	(Eingabewerte)	gut
	A5	(Dialogschritte)	gut
Selbstbeschreibungsfähigkeit	SB1	(Nutzer leiten)	gut
	SB2	(Notwendigkeit ext. Hilfe)	gut
	SB3	(Zustandsänderungen)	befriedigend
	SB5	(Dialoginteraktion)	gut
	SB6	(Formatinformationen)	befriedigend
Erwartungskonformität	E1	(Vokabular)	gut
	E2	(Rückmeldung)	gut
	E3	(Erwartete Antwortzeit)	befriedigend
	E4	(Informationsstrukturierung)	gut
	E6	(Art von Rückmeldungen)	befriedigend
	E7	(Konsistentes Dialogverhalten)	gut
	E8	(Eingabepositionen)	gut
	E9	(Art der Informationen)	befriedigend
Lernförderlichkeit	L1	(Regeln & Konzepte)	gut
	L2	(Dialoghilfe)	gut
	L3	(Dialogunterstützung)	gut
	L4	(Umfang von Rückmeldungen)	befriedigend
	L6	(Ausprobieren von Dialogschritten)	gut
	L7	(Eingabepositionen)	gut
Steuerbarkeit	ST1	(Interaktionsgeschwindigkeit)	gut
	ST2	(Fortsetzen des Dialoges)	gut
	ST3	(Wiederaufnahme des Dialoges)	gut
	ST4	(Reversibilität von Handlungen)	gut
	ST6	(Ein-/Ausgabemittel)	gut
	ST7	(Voreinstellungen)	gut
	ST8	(Ändern von Daten)	gut
Fehlertoleranz	F1	(Eingabefehler)	gut
	F2	(Undefinierte Systemzustände)	gut
	F3	(Fehlerhilfe)	gut
	F4	(Fehlerbeseitigung)	gut
	F5	(Automatische Fehlerkorrektur)	keine Unterstützung
	F6	(Zurücksetzen der Fehlerkorrektur)	befriedigend
	F7	(Zusätzliche Fehlerinformationen)	gut
	F8	(Datenprüfung)	gut
	F9	(Fehlerbehebung)	befriedigend
	F10	(Auswirkung von Benutzerhandlungen)	gut
Individualisierbarkeit	I1	(Benutzercharakteristik)	befriedigend
	I2	(Darstellungsform)	gut
	I3	(Erläuterungen)	gut
	I4	(eigenes Vokabular)	keine Unterstützung
	I5	(Ein-/Ausgabegeschwindigkeit)	ungenügend
	I6	(Dialogtechniken)	gut
	I7	(Interaktion)	gut
	I8	(Eingabe-/Ausgabedarstellung)	gut
	I9	(Dialogelemente)	gut
	I10	(Dialogeinstellungen)	gut

3 Didaktische Anforderungen

3.1 Grundlagen

Welche Anforderungen sind insbesondere aus didaktischer Sicht an eine Entwicklungsumgebung für Schulen zu stellen? In Beantwortung dieser Frage ist vorerst zu klären, was den Einsatz von IDEs an Schulen von anderen Anwendungsgebieten besonders abhebt. Darüber hinaus ist zu analysieren, ob oder warum für Schulen überhaupt eine spezielle IDE-Version erforderlich ist. Hierzu wird im Weiteren zunächst betrachtet, welche Lehrinhalte an den Schulen im Informatikunterricht vermittelt werden sollten. Fleischer (1992) sieht die Vermittlung der folgenden vier Schlüsselqualifikationen als einen Schwerpunkt innerhalb des Informatikunterrichts an:

- Abstraktionsvermögen,
- Logisches Denkvermögen,
- Beherrschen von Arbeitstechniken sowie das
- Beherrschen von Problemlösungsmethoden.

Diese Fähigkeiten sollen die Schüler nach Fleischer anhand einer konkreten Sprache in unterschiedlichen Lernphasen erlangen. Er benennt dabei folgende Phasen:

- Einstiegsphase,
- Erarbeitungsphase,
- Vertiefungs-/Sicherungsphase,
- Transferphase.

Diese Lernphasen verdeutlicht Fleischer am Beispiel der Programmiersprache *Pascal*. So sollen in der Einstiegsphase zunächst Zahlendarstellungen verstanden werden. In der Erarbeitungsphase stehen verschiedenen Arten von Datentypen im Mittelpunkt. Der Dozent vermittelt hierzu schrittweise Inhalte, Beispiele sowie Musterwege. **In der Vertiefungs-/Sicherungsphase**

treten eigene Aktivitäten der Schüler mehr in den Vordergrund indem sie eigene Übungen absolvieren. Sie ist somit durch Selbststudium sowie Gruppenarbeit geprägt. Diese Phase bietet folglich gute Ansatzpunkte für den ersten **Einsatz einer möglichen Entwicklungsumgebung**. Mit deren Hilfe könnten Schüler Ihre Aufgaben erfüllen und zugleich die in den Phasen 1 und 2 gewonnen Kenntnisse, Fähigkeiten und Fertigkeiten vertiefen. In der Transferphase werden die erbrachten Leistungen vom Lehrer zunächst bewertet und in der Folge werden weitere Anwendungen oder Probleme in anderem Kontext aufgezeigt. Zum Beispiel erwähnt Fleischer hier die Möglichkeit, die benutzereigene Definition von Datentypen zu vermitteln. Wie bereits dargestellt, bezog Fleischer seine Ausführungen zu den Schlüsselqualifikationen bzw. Lernphasen ausschließlich auf die Programmiersprache *Pascal*. Aus der Literatur sind weitere Beispiele bekannt, in der auf die Sprache BASIC und deren Dialekte (vgl. Klingen & Otto (1986), Baumann (1990)) Bezug genommen wurde. Folglich sind die **an eine Entwicklungsumgebung zu stellenden Anforderungen auch unter dem Aspekt deren Unterstützung von Programmiersprachen zu betrachten**. Bei der Frage, welche Programmiersprachen durch Entwicklungsumgebungen unterstützt bzw. bevorzugt werden sollten, ergeben sich unterschiedliche Ansatzpunkte.

Nach Klingen & Otto (1986) sind Laufzeiteffizienz sowie die Möglichkeit, mit Hilfe einer Sprache professionelle Anwendungen erstellen zu können, für die Schule dabei eher von untergeordneter Bedeutung. Viel wichtiger hingegen ist in letzter Konsequenz die **Eignung einer Programmiersprache zur interaktiven Lehre**. Wichtig hierbei ist jedoch auch, dass die Sprache nicht nur prinzipiell geeignet sein sollte, sondern auch **in das entsprechend gewählte Unterrichtskonzept passt und dieses ergänzt**. In ihren Untersuchungen beschreiben Forneck & Goorhuis (1990) verschiedene Entwicklungstendenzen derartiger Konzepte und weisen auf damit verbundene Probleme in der Didaktik der Informatik hin. Sie benennen dabei genauer einen:

- algorithmenorientierten,
- anwendungsorientierten,
- benutzerorientierten und einen

- rechnerorientierten Ansatz.

Auf Letzteren wird jedoch nicht näher eingegangen. Die Ansätze untergliedern Forneck & Goorhuis (1990) in zwei verschiedene Konzeptionen. Den algorithmisch- sowie rechnerorientierten Ansatz weisen sie dabei der Gruppe der wissenschaftspropädeutischen[3] Konzeption zu. Der lebenspraktisch-orientierten Konzeption ordnen sie hingegen den anwendungs- sowie benutzerorientierten Ansatz zu. Den Studien zu Folge, wird letzterer auf Gymnasien jedoch nicht verwendet. Zum Zeitpunkt der genannten Analysen dominierte dort der rein algorithmische Ansatz. Dieser wird um einen Teil mit Beschreibungen gesellschaftswissenschaftlicher Auswirkungen ergänzt. **Als besonders problematisch an diesem Vorgehen sehen sie vor allem das Verstärken der Probleme der jeweiligen Ansätze.** Nach ihrer Auffassung verfestige dies weiterhin die konzeptionelle Spaltung zwischen beiden Konzepten und tangiere die vertikale Durchlässigkeit des Bildungssystems. Schlussfolgernd schlagen sie vor, **beide Konzepte in den Unterricht einfließen zu lassen**. Angepasst an das jeweilige Schulniveau, sollte ein wissenschaftspropädeutisches Konzept einer lebenspraktischen und infolgedessen ganzheitlich-integrativen Konzeption vorgeschaltet werden. Damit unterstützen sie indirekt die Thesen von Koerber und Peters. Auch wenn diese sich explizit nur auf die allgemeinbildenden Schulen beziehen, so kritisieren sie jedoch ebenso, dass spezifische Einzelheiten einer Sprache nicht der eigentliche Gegenstand des Informatikunterrichts sein solle (vgl. Koerber und Peters zitiert durch Baumann (1990)). Die Praxis gestaltet sich Baumann (1990) zu Folge jedoch so, dass der Unterricht als Programmiersprachenkurs realisiert wird:

> „An kleinen Problemen, die jeweils ein bestimmtes sprachliches Ausrucksmittel thematisieren, wird - informell - in Syntax und Semantik dieser Sprache eingeführt."

Gleichzeitig wird jedoch auch ein rein anwendungsorientierter Ansatz kritisiert. **Ähnlich wie Keuffer schlägt Baumann einen kombinierten Ansatz vor, bei dem die Programmiersprachen noch immer im Zentrum stehen.** Jedoch sollten in den einzelnen Sekundarstufen unterschiedliche Programmiersprachen behandelt werden. So schlägt er für die **Sekundarstu-**

[3] Keuffer beschreibt die Wissenschaftspropädeutik basierend auf Fischer als ein spezifisches didaktisches Prinzip der gymnasialen Oberstufe, dass über eine reine Wissenschaftsorientierung hinaus als Hinführung zur modernen Wissenschaftlichkeit in ihrer Maßgeblichkeit, Zuständigkeit, aber auch Bedenklichkeit verstanden wird (vgl. Keuffer & Kublitz-Kramer (2008)).

fe I eine IMPERATIVE Programmiersprache wie *Pascal* vor. Für die **Sekundarstufe II** sieht er DEKLARATIVE oder FUNKTIONALE Sprachen als besser geeignet an. Diese Sprachen dienen der Beschreibung von Problemlösungen, der Darstellung allgemeiner Modelle der Informationsverarbeitung und schließlich der Formulierung der theoretischen Grundlagen der Informatik. Beispielhaft wird hierfür die Sprache Prolog vorgeschlagen. Auf Grund ihrer einfachen Syntax kann sich der Nutzer auf grundlegende Probleme konzentrieren.

Welche Schlussfolgerungen lassen sich hieraus für den Einsatz von Entwicklungsumgebungen an Schulen (speziell an sächsischen Gymnasien) ziehen? Der sächsische Lehrplan des naturwissenschaftlichen Profils für Gymnasien schreibt explizit keine Programmiersprache vor. Den vorgenannten Ausführungen folgend, sollte eine Entwicklungsumgebung aus didaktischer Sicht möglichst den folgenden beiden angeführten Anforderungen genügen:

- Unterstützung möglichst mehrerer Programmiersprachen.
  Idealerweise sind dies Imperative, Deklarative sowie Funktionale. Dies ermöglicht den übergreifenden Einsatz ein und derselben Entwicklungsumgebung über mehrere Klassenstufen hinweg.

- Eignung zur Veranschaulichung unterschiedlicher Programmierkonzepte.
  Mit der Verwendung einer IDE ist meist ein Lernprozess verbunden. Selbst in einer minimalen Version. Werden Schüler schon früh an diese herangeführt, so entfällt ein ständiges Umlernen auf Grund der IDE. In der Folge können in unterschiedlichen Klassenstufen unterschiedliche Konzepte der Programmierung an Hand ein und derselben Entwicklungsumgebung veranschaulicht werden. Ebenso wird aufgezeigt, dass ein solches Programmierkonzept nicht von der gewählten Entwicklungsumgebung abhängig ist.

Probleme treten häufig jedoch besonders dann auf, wenn bereits in frühen Stufen eine in der Regel sehr komplexe Entwicklungsumgebung zum Einsatz kommt. REICHERT ET AL. (2005) weisen zu Recht darauf hin, dass immer umfangreichere Entwicklungsumgebungen erhältlich sind. Diese sind gerade für Einsteiger jedoch keineswegs einfach zu handhaben. **Der Einstieg in die Programmierung wird so unnötig erschwert.** Als mögliche Auswege

schlagen sie eine **Reduzierung des Sprachumfang** sowie eine **Reduzierung der Entwicklungsumgebung** vor. So genannte Mini-Umgebungen reduzieren zunächst die Komplexität der Entwicklungsumgebung und gestatten somit einen einfachen Einstieg. Über einen Aktoren werden bestimmte Aufgaben und Probleme gelöst. Als Beispiel verweisen Reichert et al. (2005) auf die Kara-Umgebung, die versucht einen spielerischen Einstieg in die *Java*-Programmierung zu vermitteln. In höheren Stufen ist dann der Umstieg auf die „richtige“ Entwicklungsumgebung JavaKara möglich. Der zweite Lösungsansatz basiert auf der Reduzierung des Sprachumfanges der Entwicklungsumgebung. So stellt z.B. JavaKara eine spezielle Schnittstelle zur Verfügung, die den Schülern nur die für sie relevanten Ausschnitte bereitstellt. Ein ähnlicher Lösungsweg wurde mit der Mini-Umgebung Karel the Robot beschritten. Dennoch weist dieser Ansatz auch Nachteile auf. So kritisieren Schubert & Schwill (2004), dass so die Tiefe, bis zu der Informatikkenntnisse erworben werden können, beschränkt wird. Weiterhin führen sie an, dass bei dieser Verfahrensweise gegebenenfalls **nicht alle Konzepte der Programmiersprache vorhanden sind**. Folglich wäre später der Umstieg auf eine vollwertige Programmiersprache zwangsweise nötig. Auch das Problem unterschiedlicher Vorkenntnisse im Klassenverbund wird hiermit seiner Meinung nach nur teilweise gelöst.
Außer den von Reichert bereits genannten Beispielen ist die Erwähnung eines zusätzlich eingebauten Strukturedi tors im Karel-System besonders bemerkenswert. Dieser Editor ermöglicht es, Programme über kontextsensitive Menüs zu erstellen. Für die Cursorposition werden jeweils nur die syntaktisch korrekten Strukturelemente angeboten. Handeingaben und folglich Syntaxfehler können somit reduziert werden. **Unerfahrenen Schülern verschafft diese Lösung nach Ansicht des Autors daher einen leichteren Einstieg in die Programmierung.** Zusammenfassend formulieren Reichert et al. (2005) folgende Anforderungen an Entwicklungsumgebungen:

- Komplexität reduzieren bzw. verstecken,
- Grafische Darstellung der Aktionen und des Zustandes der Maschine,
- Konzeptionell einfachere Maschine als einen Computer für Anfänger,
- Verkleinerter Sprachumfang sowie eine

- Einfache Programmierumgebung.

Einen weiteren Lösungsansatz stellt Huberwieser (2004) vor, in dem er auf den **Einsatz von Simulatoren sowie Code-Generatoren** verweist. Aus formalen Beschreibungen auf sehr abstrakter Ebene kann so ein lauffähiger Programmcode erstellt werden. Den Schülern wird die Bedeutung ihrer Modellierungsergebnisse dementsprechend direkt „vor Augen geführt". Simulationen bezieht Huberwieser hierbei auf die visuelle Simulation des zeitlichen Ablaufs der entwickelten Modelle. Die Schüler erhalten hierdurch die Möglichkeit, die Ergebnisse mit der Anforderungsbeschreibung unmittelbar nach der Modellierungsphase zu vergleichen. **Insbesondere die Code-Generatoren betrachtet der Verfasser der vorliegenden Arbeit als einen geeigneten Ansatz um speziell den Schülern niedrigerer Klassenstufen ohne Interesse oder Bezug zur Informatik den Einstieg in die Programmierung zu erleichtern.** In höheren Gymnasialstufen könnte diese Funktion dann zunächst auf grundlegende Bausteine sowie auf eine Syntaxhervorhebung reduziert werden.
Weitere Anforderungen für Entwicklungsumgebungen lassen sich aus den Darlegungen von Dick (2000) abgeleiten. Dick näherte sich über das Gebiet der **Gebrauchstauglichkeit** speziell dem Feld der Lernprogramme und deren didaktischer Gestaltung. Die von ihm formulierten Anforderungen gelten aus Sicht des Autors jedoch auch für die Entwicklungsumgebungen in der Schule. Diese werden im Stile eines Lernprogramms für Programmiersprachen verwendet. Für den Geltungsbereich der Didaktik stehen für Dick (2000) besonders die **Lernförderlichkeit** sowie die **Selbstbeschreibungsfähigkeit** im Mittelpunkt. So führte er aus, dass die Programmbedienung insbesondere bei Lernprogrammen so zu gestalten ist, dass **kein zusätzlicher Lernprozess erforderlich ist**. Eine selbsterklärende sowie intuitive Benutzerführung ist dabei Grundelement. Folgende Mittel erscheinen ihm speziell für Lernprogramme wichtig:

- Verzicht auf apokryphe Symbole, deren Bedeutung erst gelernt werden muss. Im einfachsten Fall sind nur Beschriftungen oder Symbole mit Beschriftungen zu verwenden.
- Bessere Orientierung durch festen Platz auf dem Bildschirm. Auch sollten Symbole in unterschiedlichen Anwendungen nicht für unterschiedliche Funktionalitäten verwendet werden, da dieses entgegen den Erwartungen des Nutzers steht. Die Oberfläche der IDE sollte also ein konsistentes Verhalten bieten.

- Steuerelemente auf die jeweils Nötigen reduzieren.
- Fehler schon bei Eingabe abfangen.

Nach Auffassung des Verfassers dieser Arbeit kann diese Liste darüber hinaus um die weiteren Vorgaben der Gebrauchstauglichkeit ergänzt werden, welche bereits bei der Untersuchung der Entwicklungsumgebung ECLIPSE vorgestellt wurden. Die hier angeführten Beispiele verdeutlichen besonders den Faktor des Einsatzes von Anwendungen speziell in der Lehre. Eine lange Lernphase nur für die Entwicklungsumgebung behindert nur die Arbeit an den eigentlichen Lernzielen. Ebenso führt der Einsatz nicht relevanter Steuerelemente zur Überforderung der Nutzer oder gar zur Verführung zu nicht gewünschter Aktivitäten (vgl. DICK (2000)). **Zusammenfassend stellen sich also zunächst die folgenden Anforderungen dar:**

- Unterstützung möglichst mehrerer Programmiersprachen
- Eignung zur Veranschaulichung unterschiedlicher Programmierkonzepte
- Reduzierung des Sprachumfang
- Reduzierung (der Komplexität) der Entwicklungsumgebung
- Grafische Darstellung der Aktionen und des Zustandes der Maschine
- Konzeptionell einfachere Maschine als ein Computer für Anfänger
- Einfachere Programmierumgebung
- Selbsterklärende und verständliche Symbole oder Beschriftungen
- Konsistente Benutzeroberfläche
- Fehlertoleranz

3.2 Evaluation mit Lehrern

Wie bereits dargelegt, besteht das Ziel der vorliegenden Arbeit in der Erstellung einer, für den Einsatz in der Lehre an sächsischen Gymnasien angepasste, skalierbare LAZARUS-Version.

Zur Vertiefung und Präzisierung der praktischen Anforderungen an eine solche Version wurden Lehrer sächsischer Gymnasien über den Weg einer schriftlichen Befragung in diesen Prozess einbezogen. Hierbei war zu beachten, welche unter Abschnitt 3.1 genannten Punkte im praktischen Unterricht Berücksichtigung finden müssen, welche dem Lehrkörper dabei besonders wichtig erscheinen und welche hingegen aus dessen Sicht für die Lehre eher von untergeordneter Bedeutung sind.

Insgesamt nahmen an der Befragung 15 Lehrer teil. Von Ihnen unterrichten 10 in der Sekundarstufe I und/oder 14 in der Sekundarstufe II . Zum überwiegenden Teil (11) handelte es sich dabei um Lehrer, die unter anderem im naturwissenschaftlichen Profil unterrichten. Weiterhin unterrichten 2 Lehrer im künstlerischen Profil, je einer im Sportprofil und im gesellschaftswissenschaftlichen Profil. Hinzu kommen 3 Nennungen von Informatik oder Informatiksystemen. Diese stellen jedoch kein eigenes Profil dar. Interessant ist auch die Aussage eines Lehrers aus dem Bundesland Brandenburg. Ihm zu Folge ist im dortigen Schulsystem kein Profilunterricht vorgesehen.

Im Weiteren Verlauf dieses Kapitels werden nun die den Lehrern zur Problematik „Didaktische Anforderungen an eine Entwicklungsumgebung für Schulen“ vorgelegten Fragen erläutert. Mit Hilfe dieser Ergebnisse sollen, unter Einbezug der Ergebnisse der Literaturrecherche, **allgemeine didaktische Anforderungen an eine schulische Entwicklungsumgebung formuliert werden**. Im Verlauf der Evaluation wurden darüber hinaus jedoch auch Fragen zu konkreten Änderungen an der Lazarus-IDE gestellt. Diese haben jedoch keinen allgemeinen Charakter und gelten speziell für Lazarus. Um diese inhaltlich von den hier zu formulierenden Anforderungen zu trennen, werden diese Fragen in Kapitel 4.2 gesondert betrachtet.

Welche Programmiersprachen werden an Ihrer Schule gegenwärtig vermittelt?

Im Ergebnis vorhergehender Untersuchungen wurde bereits dargelegt, dass im Informatikunterricht an sächsischen Gymnasien unterschiedliche Programmiersprachen vermittelt werden. Die Ursache hierfür ist vor allem darin zu suchen, dass die Lehrpläne hinsichtlich zu verwendender Programmiersprachen keine Vorgaben enthalten. So zeigte sich im Ergebnis der Befragung dann

auch, **dass das Spektrum der im Informatikunterricht verwendeten Programmiersprachen sehr breit gefächert ist**. Immerhin kommen 12 unterschiedliche Programmiersprachen zum Einsatz.

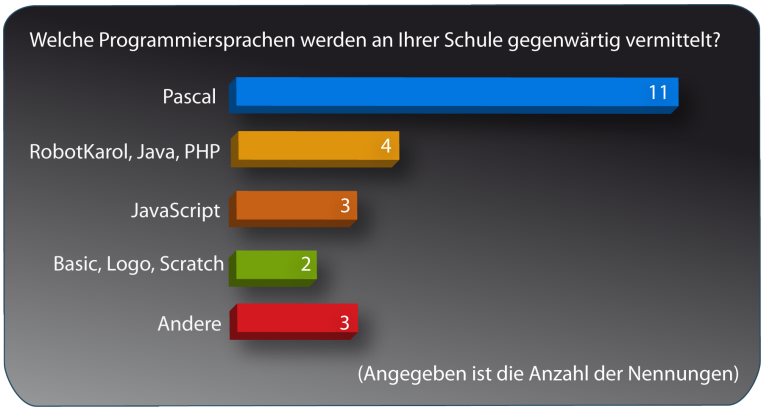

Abbildung 5: „Welche Programmiersprachen werden an Ihrer Schule gegenwärtig vermittelt?“

Abbildung 5 zeigt, dass dabei die *Pascal*-Dialekte mit 11 Nennungen die größte Verbreitung finden (Delphi-/Lazarus-Nutzer gemeinsam gezählt). Die speziell für den Unterricht konzipierte Sprache *Basic* ist hingegen mit 2 Nennungen kaum anzutreffen. Da in die Befragung nicht nur Lehrer der Sekundarstufe II sondern auch der Sekundarstufe I involviert waren, erfolgten ferner auch Nennungen für die Umgebungen Karel the Robot (4 Nennungen) sowie Logo (2 Nennungen). Jeweils viermal wurde ferner die Skriptsprache *PHP* sowie *Java* benannt. Über die Gründe für den Einsatz dieser Sprachen geben die Antworten auf eine weitere Frage an die Lehrer Aufschluss: Der Einsatz einer Sprache ist eng mit der Verwendung einer Entwicklungsumgebung verbunden. Also wurde hinterfragt, was aus didaktischer Sicht für den Einsatz der gewählten Entwicklungsumgebung sprach.

Welche Aspekte - insbesondere aus didaktischer Sicht - führten zur Entscheidung für den Einsatz der derzeit verwendeten IDE?

Auf Grund ihrer engen Verknüpfung gelten die gegebenen Antworten aus Sicht des Autors gleichermaßen für Programmiersprache und Entwicklungsumgebung. So unterstrich ein Lehrer, er

verwendet in seinem Unterricht *Java*, *PHP* sowie *SQL* mit Hilfe des Editors PHASE 5, die Bedeutung für den praktischen Einsatz im späteren Leben wie folgt:

> „... denn nicht jeder Schüler wird Informatik studieren. Aber jeder Schüler wird später mit Webtechniken umgehen müssen."

Die Bedeutung - besonders von *Java* - für die Gesellschaft wurde in der Folge auch von weiteren Lehrern unterstrichen. Ein Lehrer bezog dieses in seiner Antwort ausdrücklich auf den späteren Beruf oder ein mögliches Studium der Schüler. In seinem Unterricht setze er jedoch die Entwicklungsumgebung ECLIPSE ein. Aus seiner Sicht sprechen hierfür besonders folgende positive Aspekte:

- Plattformunabhängigkeit
- Erweiterbarkeit durch Plug-Ins
- Freie Verfügbarkeit

Ein weiter interessanter Gesichtspunkt wurde durch einen Lehrer eingebracht, der u. a. DELPHI verwendet. Er bewertete die Möglichkeit, in seinem Unterricht auf **unterschiedliche Konzepte eingehen zu können als besonders positiv**:

> „Jetzt kann man im Profilunterricht die imperative Programmiermethode nutzen, um dann mit steigenden Fertigkeiten auf das neue Konzept der objektorientierten Programmierung einzugehen. DELPHI ermöglicht also 2 Zugänge: Zur imperativen und der objektorientierten Programmiermethode."

Eine ähnliche Wertung gibt auch ein weiterer Lehrer ab. Für ihn ist die Möglichkeit, in der Sekundarstufe II mit Hilfe von DELPHI auf die objektorientierte Programmierung übergehen zu können, besonders wichtig. **Beide Lehrer unterstreichen somit direkt die von Baumann oder auch Keuffer vorgeschlagenen Ideen der unterschiedlichen Programmierkonzepte und -sprachen in unterschiedlichen Stufen.** Zusätzlich benannten diverse Lehrer für den Unterricht folgende Merkmale von DELPHI als positiv:

- Leichter Einstieg in die Programmierung da alle notwendigen Objekte vorgegeben sind (*Pascal*-Grundlagen),
- Deutschsprachige Versionen und Hilfe,
- Syntax entspricht genau den gewünschten Schreibweisen lt. Lehrbuch,
- Gute Fehlererkennung und Erklärung,
- Grafische Oberfläche,
- Gute Debuggerfunktion sowie
- Überwachung von Ausdrücken und Funktionsaufrufen.

Entscheidend für den Einsatz einer Entwicklungsumgebung im Unterricht sind jedoch nicht nur rein funktionelle Aspekte sondern auch deren möglichst freie Verfügbarkeit. **Ein weiterer Proband wies in diesem Zusammenhang auf die zahlreichen positiven Aspekte der freien Verfügbarkeit von Lazarus hin.** In seinen Ausführungen legte er ferner dar, dass die dortige Auswahl für eine Entwicklungsumgebung in drei Stufen erfolgte:

> „Die Entscheidung für Delphi bzw. Lazarus wurde vor allem durch die Kostenfrage bestimmt und wir wollten den Schülern die Möglichkeit geben, zu Hause zu Programmieren. (Deshalb „nur“ Delphi 7). Weiterhin führte der Motivationseffekt einer grafischen Oberfläche zur Entscheidung. An dritter Stelle stand die Tatsache, dass die Kollegen die Delphi-Syntax von Turbo Pascal (der häufigsten Lehrsprache beim Studium) her kannten.“

Die Kostenfrage sowie die Möglichkeit, dass Schüler zu Hause am eigenen PC schulische Aufgaben üben und vertiefen können darf bei den Betrachtungen zu einer schulischen Entwicklungsumgebung also neben allen theoretischen Überlegungen nicht außer Acht gelassen werden. Diese wesentliche Aussage wird im Weiteren durch die Ausführungen eines anderen Lehrers zur Delphi-IDE bekräftigt:

> „Solange es für die Schüler möglich war eine freie Version zu beziehen, war eine Festigung durch Hausarbeiten und Projekte beim Umgang mit dieser IDE möglich.

> Diese Möglichkeit besteht leider nicht mehr, so dass die Arbeit mit der IDE auf die Unterrichtszeiten beschränkt ist.“

Neben Lizenzproblemen bewerteten die Lehrer hinsichtlich der Delphi-IDE folgende Punkte negativ:

- Zu viel „Overhead“,
- Nicht ohne weiteres für die Schüler frei verfügbar,
- Umständliche Ein-/Ausgaben durch die Typumwandlungen wie „StrToInt()“ usw. machen es Anfängern schwer,
- Verwendung des Objektmodell,
- Weiterentwicklungen sind zu komplex.

Speziell aus den Äußerungen zu *Java*/Eclipse sowie *Pascal*/Delphi lassen sich also aus Sicht der Praxis bereits wertvolle Erkenntnisse für eine Entwicklungsumgebung ableiten. Dabei ist jedoch zu berücksichtigen, dass speziell in einer Antwort die Vertrautheit des Fachlehrers mit Delphi als einziger Grund für den Einsatz von Delphi benannt wurde. Auch zu anderen Sprachen/Entwicklungsumgebungen wurden Kommentare und Anmerkungen abgegeben. Zunächst erfolgte dabei eine Aufzählung deren positiver Aspekte:

- Bedeutung der grafischen Oberfläche
- Korrektheit
- Zuverlässigkeit
- Syntaxhervorhebung
- Integration von UML
- Unterstützung mehrerer Programmiersprachen
- Mobiler Einsatz ist ohne zusätzlichen Installationsaufwand möglich. Damit ist die in der Schule verwendete Umgebung identisch mit der in der häuslichen Umgebung benutzten.

Folgende Aspekte wurden als negativ bewertet:

- Ein Editor ohne Syntaxhervorhebung erweist sich für die Schüler in der Regel als schwierig.
- Komplexität
- Hohe Beanspruchung der Hardware und damit verbundene lange Wartezeiten.
- Starke Quelltextorientierung und damit hoher Aufwand zum Erwerb der Syntax.

Um das Bild im Weiteren abzurunden und didaktische Anforderungen präziser ableiten zu können, wurde ferner die Auffassung der Lehrer zu den aus der Fachliteratur bekannten Anforderungen abgefragt:

Welche Anforderungen stellen Sie an eine, den speziellen Belangen einer Schule angepasste, Entwicklungsumgebung?

Die Auffassung der befragten Lehrer wird durch Abbildung 6 zusammengefasst. **Hierbei ist besonders auffällig, dass die Lehrer mit der Syntaxhervorhebung ein Merkmal am häufigsten nannten, welches bei den theoretischen Untersuchungen zunächst eher beiläufig erwähnt wurde.** Besonders aus dem Blickwinkel vor allem „uninteressierter Schüler“ (im Sinne von nicht an Programmierung interessiert) wird die Häufigkeit der Nennung jedoch verständlich. So bietet die Syntaxhervorhebung entsprechend der Sprache eine wertvolle Hilfe bei der Programmierung. Dem Versuch, fehlerhaften Quellcode übersetzen zu wollen, kann somit frühzeitig entgegengewirkt werden.
Als weiterhin wichtig erachten jeweils 73% der Lehrer die Einfachheit und Robustheit einer Entwicklungsumgebung. Zusätzlich sollte sie selbsterklärend sein. Das verringert die Einarbeitungszeit für die Schüler und unterstützt somit auch die Lernförderlichkeit. Immerhin deutlich mehr als die Hälfte der befragten der Lehrer erachtete eine **geringere Komplexität** darüber hinaus als wichtig. Hingegen wurden die **Individualisierbarkeit** der IDE oder ein möglicher **eingeschränkter Sprachumfang kaum genannt** (20%). Dieses Ergebnis weist aus Sicht des Autors zumindest hinsichtlich des eingeschränkten Sprachumfangs auf

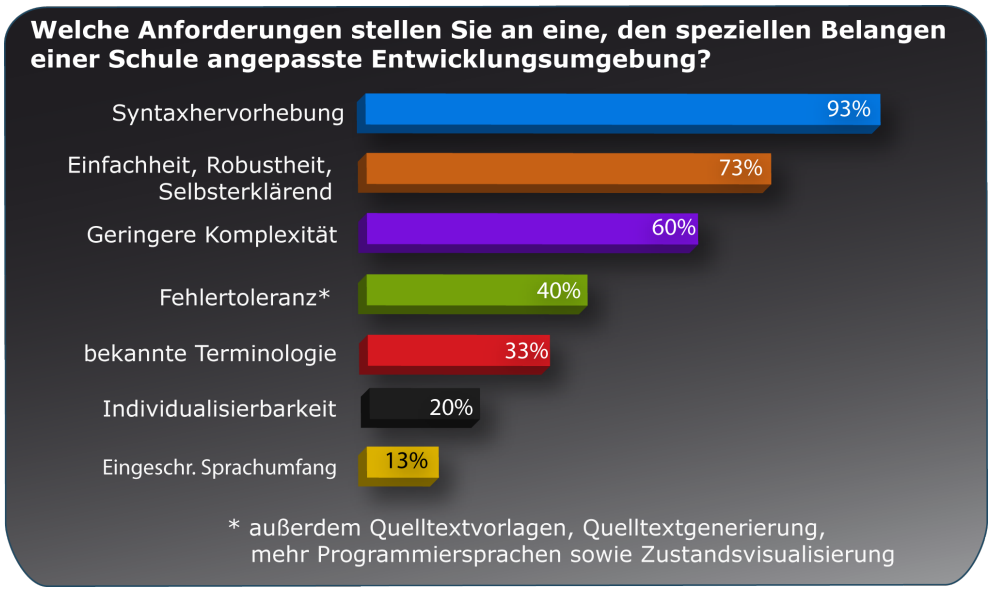

Abbildung 6: „Welche Anforderungen stellen Sie an eine, den speziellen Belangen einer Schule angepasste, Entwicklungsumgebung?“

einen Widerspruch zwischen Theorie und Praxis hin (siehe hierzu auch Abschnitt 3.4). Darüber hinaus wurde den Lehrern die Möglichkeit gegeben, weitere (eigene) Anforderungen zu formulieren. Hierbei wurden jedoch teilweise schlicht aus den negativen Merkmalen der momentan verwendeten Entwicklungsumgebungen Anforderungen formuliert.

- Geringe Hardwareanforderungen
- Ausführliche deutsche Hilfe
- Portabilität
- Erzeugung von Strukturdiagrammen (ähnlich Karol the Robot)
- Plattformunabhängigkeit
- Freie Verfügbarkeit
- Einfache Installation

Im folgenden Abschnitt wird nun versucht, Erkenntnisse aus Theorie und Praxis zusammenzuführen und Anforderungen an eine schulische Entwicklungsumgebung aus didaktischer Sicht aufzustellen.

3.3 Folgerungen für eine schulische Entwicklungsumgebung

Die Evaluation mit Unterstützung der Lehrer zeigte, dass sich nicht jede theoretische Erkenntnis 1:1 in die Praxis umsetzen lässt. Aus den theoretischen Untersuchungen konnten Anforderungen abgeleitet werden, die einen ersten interessanten Ansatzpunkt für künftige Entwicklungsumgebungen lieferten. Unter Einbeziehung der in der praktischen Evaluation gewonnenen Erkenntnisse werden in der Folge die nunmehr präzisierten Anforderungen kurz dargestellt. Hierbei erfolgt eine Unterteilung in „Soll"- und „Kann"-Anforderungen. Der Grund für die Unterteilung liegt in der unterschiedlichen Bedeutung der Anforderungen. Erstere sieht der Autor als elementar an. Sie müssen von einer schulischen Entwicklungsumgebung unterstützt bzw. erfüllt werden. Die „Kann"-Anforderungen haben hingegen aus Sicht des Autors empfehlenden Charakter.

3.3.1 „Soll"-Anforderungen

- **Unterstützung mehrerer Programmiersprachen**

Bereits während der Literaturrecherche rückte nach Auffassung des Verfassers besonders die Unterstützung mehrerer Programmiersprachen als eine grundlegende Anforderung in den Vordergrund. Ferner gaben auch in der Evaluation zahlreiche Lehrer an, mehrere Programmiersprachen bereits im Unterricht zu vermitteln. Würden nun in der Folge mit Hilfe dieser Sprachen unterschiedliche Sprachkonzepte (wie von diversen Autoren gefordert) unterstützt, eröffnet das die Möglichkeit des Einsatzes einer über die Jahre „schulbegleitenden" Entwicklungsumgebung. Ein ständiger Wechsel von Entwicklungsumgebungen und ein damit verbundenes Umlernen würden somit hinfällig. Dies wird auch durch die Evaluation nochmals bekräftigt. Besonders die zahlreichen Kommentare zur Delphi-Umgebung bestätigen diesen Umstand. Hier wurde insbesondere hervorgehoben, dass Dank Delphi der Umstieg von der imperativen auf die objektorientierte Programmierung gut möglich ist.

- **Syntaxhervorhebung**

Aus der Fachliteratur konnten kaum Hinweise auf eine Syntaxhervorhebung als maßgebliche Anforderung ermittelt werden. Huberwieser erwähnt zunächst Generatoren zur Quelltextgene-

rierung. Weiterhin führt Reichert den Strukturediteor des KAREL-Systems an, der nur syntaktisch korrekte Strukturelemente anbietet. Die Evaluation zeigte mit 93% der Stimmen jedoch, dass gerade diese Funktion von den Lehrern als außerordentlich wichtig angesehen wird. In den Kommentaren der Lehrer wird diese Aussage eindrucksvoll bekräftigt. An dieser Stelle sei abschließend nochmals auf den bereits erwähnten Lehrerkommentar verwiesen, in dem er auf die Probleme der Schüler bei nicht-vorhandener Syntaxhervorhebung hinwies.

- **Einfachheit / geringe Komplexität**

Nahezu ein Dreiviertel der befragten Lehrer gab in der Evaluation die Einfachheit als eine der wichtigsten Forderungen an eine Entwicklungsumgebung an. Für 60% ist eine geringere Komplexität einer IDE sehr wichtig. Auf Grund des engen, wechselseitigen Zusammenhanges beider Forderungen werden diese in der Folge nicht mehr differenziert betrachtet. Viel mehr sieht der Autor in der geringeren Komplexität einen Weg hin zu einer einfacheren Entwicklungsumgebung. Einfache Entwicklungsumgebungen wurden in Theorie (Literatur) sowie Praxis (Evaluation) mehrfach nachdrücklich gefordert. Reichert sprach allgemein von einer zu großen Komplexität bei Entwicklungsumgebungen. Ein Lehrer umschrieb diese Problematik mit „Zu viel *Overhead*“. Dick verwies darauf, dass Steuerelemente nur auf das Nötigste reduziert werden sollten. Zusammengefasst sollte die Entwicklungsumgebung also auf einen angemessenen, einfachen Rahmen reduziert werden. Dies gilt speziell für die Sekundarstufe I. Die Nennungen und die damit verbundene Verwendung der Umgebungen LOGO sowie KAREL THE ROBOT unterstreichen diese Aussage im Besonderen. Der in der Literatur geforderte bzw. vorgeschlagene eingeschränkte Sprachumfang wurde von den Fachlehrern jedoch nicht befürwortet. Gerade einmal 13% empfanden diese Anforderung als wichtig. Diese Forderung wurde in der Folge deshalb auch vom Autor verworfen.

- **Fehlertoleranz / Selbstbeschreibungsfähigkeit**

Auf Fehlertoleranz sowie die Selbstbeschreibungsfähigkeit als solches wurde in der Fachliteratur kaum hingewiesen. Als Grundsätze der Dialoggestaltung sind sie dem Gebiet der Gebrauchstauglichkeit zuzuordnen und sollten nach Meinung des Autors grundsätzlich maßgebend für eine Entwicklungsumgebung sein - nicht nur für den Schuleinsatz. Sie werden jedoch explizit

als Anforderung aufgeführt, weil der schulische Einsatz diese aus Sicht des Autors besonders fordert: Schüler benötigen eine Entwicklungsumgebung, die potenzielle Fehler abfangen kann. Nicht jeder falsche Schritt führt somit automatisch zum Absturz. Ein robuster Umgang mit der IDE ist folglich möglich. Dies ist besonders vor dem Hintergrund des „Ausprobierens“ notwendig. Schülern mit mangelnden Kenntnissen kann so die Angst genommen werden, etwas falsch zu machen. Somit ist auch zu erklären, dass in der Evaluation 73% der Lehrer eine robuste IDE forderten. Zusätzlich zur Robustheit forderte 40% der Lehrer Fehlertoleranz als zentrales Merkmal einer potenziellen IDE.

Des Weiteren sollte die Entwicklungsumgebung natürlich selbst frei von Fehlern sein. Diese Aussage wird im Ergebnis einer weiteren Frage aus dem Evaluationsbogen (speziell zu Lazarus) durch die Lehrer deutlich bekräftigt. Über 60% der Lehrer ziehen die Arbeit mit einer stabilen und fehlerfreien Version einer ständig aktualisierten Variante vor.

Zur Selbstbeschreibungsfähigkeit liefert Dick bereits erste Ansätze. Für ihn ist eine selbsterklärende sowie intuitive Benutzerführung ein Grundelement. So fordert er den Verzicht apokrypher Symbole. Einfache Symbole oder gar nur Beschriftungen helfen hingegen viel mehr. Die Evaluation zeigte, dass wiederum 73% der Lehrer diese Forderung als wichtig ansehen. Dieser Meinung schließt sich der Autor der vorliegenden Arbeit an. Es kann für die Motivation nicht förderlich sein, wenn die Schüler einen großen Teil der Unterrichtszeit damit verbringen, Symbole oder Funktionen zu erraten. Das trifft insbesondere dann zu, wenn die Schüler der Programmierung nicht zugetan sind oder ihnen das Feld der Programmierung ohnehin fremd ist.

- **Freie Verfügbarkeit**

Freie Verfügbarkeit von Software scheint aus didaktischer Sicht zunächst kaum ein interessanter Aspekt zu sein. Dementsprechend konnten in der Literatur auch keine Verweise auf diese oder ähnliche Forderungen gefunden werden. Hingegen legen jedoch besonders die Kommentare der Lehrer offen, wie zentral und bedeutend diese Forderung für sie ist: Sollen sich die Schüler auch zu Hause mit der Programmierung befassen, kommt der freien Verfügbarkeit von Software auch aus didaktischer eine noch größere Bedeutung zu. Es geht hierbei nicht (nur) darum, interessierte und motivierte Schüler zu fördern. Vielmehr ist es ein wichtiges Anliegen, allen Schülern das

Üben und somit Festigen in Heimarbeit zu ermöglichen. In letzter Konsequenz bedeutet dies natürlich auch, Hausaufgaben mit Hilfe einer Entwicklungsumgebung anzufertigen. In Zeiten mangelnder Gelder kann somit eine restriktive Lizenzpolitik keine förderliche Basis sein.

Für die Schüler muss es nach Ansicht des Autors die Möglichkeit geben, sich die geforderte Software kostenlos und legal herunterzuladen um im erforderlichen Umfang damit arbeiten zu können. Der Weg kostenlose, eingeschränkte Schülerversionen anzubieten, von Schulen hingegen Lizenzgebühren einzufordern, mag ein Zwischenweg sein. Aus Sicht des Autors ist hier der Weg über vollständig freie und kostenlos verfügbare Software jedoch der Bessere. Auf Grund der dargelegten Fakten, insbesondere der Arbeit zu Hause, ist diese Forderung für den Autor eine der zentralen Anforderungen an eine moderne schulische Entwicklungsumgebung.

3.3.2 „Kann“-Anforderungen

- **Grafischer Editor für Benutzerschnittstellen**

Neben den bereits genannten Aspekten stellen die Lehrer in ihren Kommentaren wiederholt die Bedeutung eines visuellen Editors zur Erstellung einer grafischen Benutzerschnittstelle in den Vordergrund. Besonders Lehrer, die Delphi oder auch Lazarus verwenden, wussten diese Funktionalität zu schätzen. Beide Entwicklungsumgebungen verfügen über eine solche Möglichkeit, selbstständig grafische Benutzerschnittstellen durch Auswahl von Komponenten „zusammenzuklicken“. Weiterhin sprach ein Lehrer von einem erhöhten Motivationseffekt. Natürlich stellt auch eine Entwicklungsumgebung wie Eclipse dem Nutzer eine grafische Benutzerschnittstelle zur Verfügung. Jedoch wird per Standard kein solcher visueller Editor bereitgestellt. Von Lehrern, die mit dieser Entwicklungsumgebung arbeiten, wurde kein entsprechender Hinweis angeführt. Sie erachten folglich einen solchen Editor nicht als Pflicht. Vielmehr stellt es sich aus ihrer Sicht so dar, dass ein erfolgreicher Unterricht auch ohne einen solchen visuellen Editor möglich ist. Aus diesem Grund wird diese Forderung als „Kann“-Anforderung eingeordnet.

- **Quelltextvorlagen**

Wenigstens 40% der Lehrer votierten für eine Funktion zur Quelltextgenerierung. Es sei auch nochmals erwähnt, dass bereits Huberwieser auf Simulatoren sowie Code-Generatoren verwies.

Diese sollten verwendet werden, um aus formalen Beschreibungen auf sehr abstrakter Ebene lauffähigen Programmcode zu erstellen. Er beschreibt dabei als einen positiven Aspekt, dass den Schülern die Bedeutung ihrer Modellierungsergebnisse direkt „vor Augen geführt“ wird. Unter Beachtung der Evaluationsergebnisse sowie der Kommentare der Lehrer wurde diese Anforderung durch den Autor als „Kann“-Anforderung klassifiziert. Gerade für Schüler der Sekundarstufe I wäre ein Generator sicherlich ein guter Einstieg in die Programmierung. Einsteiger müssen zunächst erstmal ein grundlegendes Gefühl und Verständnis für die Programmierung entwickeln. Nur ein Lehrer gab in der Evaluation den Einsatz eines ähnlichen Werkzeuges innerhalb seiner Entwicklungsumgebung an. Er erwähnte dabei die Integration eines *UML*-Editors. Ob hieraus jedoch Quelltext generiert werden kann, ist nicht näher erläutert.

Vor diesem Hintergrund scheint eine solche Funktion also eher als hilfreiches, jedoch nicht notwendiges Feature angesehen zu werden.

- **Quelltextgenerierung**

Bei der Quelltextgenerierung zeichnet sich ein ähnliches Bild wie bei den Quelltextvorlagen ab. In der Fachliteratur wurde die Möglichkeit erwähnt, Nutzern bzw. im vorliegenden Fall den Schülern, ein Codegerüst zur Verfügung zu stellen. Hier müsste dann zum Beispiel nur noch die Anwendungslogik implementiert werden. Der prinzipielle Aufbau eines Programms in der jeweiligen Sprache sollte den Schülern bereits nach wenigen Unterrichtsstunden bekannt sein. Bei größeren Projekten oder Aufgaben ist eine solche Funktion jedoch sicherlich sinnvoll. Für typische Programme des Unterrichts (z.B. Umsetzung eines einfachen Algorithmus) ist eine solche Möglichkeit nach Auffassung des Autors wohl eher überdimensioniert. Auch den Lehrern erschien die Syntaxhervorhebung für die Programmierung wichtiger als die automatisierte Generierung von Quelltexten. Folglich wurde die Quelltextgenerierung als „Kann“-Anforderung charakterisiert.

- **Deutsche Sprachversion**

Einige Lehrer gaben in ihren Kommentaren an, dass sie das Vorhandensein einer deutschen Version - speziell einer deutschsprachigen Hilfe - als sehr hilfreich empfanden. Angesichts einer zunehmend globalisierten Welt, in der die Schüler ohnehin schon sehr frühzeitig an die englische

Sprache herangeführt werden, ist den Schülern der eingeschränkte Umgang mit der englischen Sprache sicherlich zuzumuten. Dies gilt insbesondere für das englischsprachig geprägte Feld der Informatik. Mangelnde Kenntnisse der Sprache oder Terminologie der Anwendung können jedoch zu Verständnisproblemen führen. Dies erschwert die Arbeit mit der Entwicklungsumgebung unnötig. Aus den oben genannten Gründen wird eine deutsche Version trotzdem „nur“ als „Kann“-Anforderung angesehen. Hierzu wird angemerkt, dass sich das „Kann“ ausdrücklich nur auf eine deutsche Sprachversion der Hilfe bezieht. Nicht jedoch auf das prinzipielle Vorhandensein einer Hilfe.

3.3.3 Nicht berücksichtigte Anforderungen

In den vorstehenden Betrachtungen wurden didaktische Anforderungen an eine IDE aufgestellt. Dabei wurden im Ergebnis der Evaluation nicht alle potenziell interessanten Anforderung in die Anforderungsliste übernommen. Diese Selektion erfolgte vorrangig dann, wenn sich für diese Anforderungen/Aspekte nur eine geringere Anzahl von Lehrern entschied oder von einer Mehrheit abgelehnt wurde. Dies traf in besonderem Maße auf den eingeschränkten Sprachumfang zu. Er fand bei den Probanden kaum Unterstützung. Analog gilt das auch für die Individualisierbarkeit der Benutzerschnittstelle. Das vollständige Anpassen der Benutzerschnittstelle scheint aus Sicht der Lehrer weder sinnvoll noch notwendig. Dies bedeutet jedoch keineswegs, dass die an der Benutzerschnittstelle von Lazarus vorgenommenen Änderungen (siehe Kapitel 4.3) aus Sicht der Lehrer hinfällig sind. Vielmehr benannten viele Lehrer Probleme mit Komponenten und Eigenschaften in Delphi/Lazarus. Diese Ergebnisse der Evaluation werden in Kapitel 4.2 gesondert betrachtet.

Bereits in seinen Untersuchungen zur Gebrauchstauglichkeit erwähnte der Autor, dass eine angepasste Terminologie für Entwicklungsumgebungen vorausgesetzt wird. Dem schienen sich die Lehrer anzuschließen, in dem 40% von ihnen Wert auf eine angepasste Terminologie legen. Da es sich hierbei nicht um die Mehrheit der befragten Lehrer handelt, lässt sich hieraus schlussfolgern, dass sie mit der ohnehin vorhandenen Verwendung der Begriffe der Programmierung in modernen Entwicklungsumgebungen einverstanden sind. Weiterhin entfiel die Visualisierung von Zuständen. Auch hier unterstützten weniger als die Hälfte der Lehrer diese Anforderung. Auch

in den theoretischen Untersuchungen konnte nur ein Verweis auf eine solche Funktion gefunden werden.

Ein weiterer interessanter Ansatzpunkt für mögliche Anforderungen ist die von einigen Lehrern geforderte Portabilität von Entwicklungsumgebungen. Eine solche Version kann ohne Installation dann zum Beispiel von einem *USB*-Stick aus verwendet werden. Die Lehrer begründeten ihren Wunsch auf Portabilität mit der Eignung zur Heimarbeit. Aus Sicht des Autors stellt das zwar einen guten Ansatz dar, ist jedoch nicht zwingend notwendig. Viel wichtiger hingegen erscheint die freie Verfügbarkeit. Selbst wenn am heimischen Rechner keine Internetverbindung für den Download einer IDE bereit steht, so kann in diesem Fall die Installationsdatei in der Schule heruntergeladen und dann per *USB*-Stick zu Hause installiert werden. Die mobile Verwendung scheint für Schüler hier nicht vordergründig wichtig.

Ähnliches gilt für die Plattformunabhängigkeit. Diese wurde ausschließlich von Lehrern erwähnt, die Eclipse einsetzen bzw. *Java* unterrichten. Der Ansatz ist sicherlich wünschenswert und zu unterstützen. In Zeiten, in denen Microsoft mit seinen Betriebssystemen *Windows XP* sowie *Vista* noch immer einen Marktanteil von ca. 90% hat (vgl. NetApplications.com) sollte dies keine zwingende Anforderung an eine schulische IDE sein.

3.3.4 Bedeutung für die Änderungen an Lazarus

Die formulierten Anforderungen liefern auch einen ersten Ansatzpunkt für die in Lazarus vorzunehmenden Änderungen. Die von der Mehrzahl der Lehrer geforderte Syntaxhervorhebung ist dabei ebenso bereits vorhanden, wie eine deutsche Sprachversion oder die freie Verfügbarkeit der IDE. **Hauptansatzpunkt für Änderungen**, und damit der eigentliche Fokus der folgenden Arbeiten, ist folglich die Anforderung der **geringeren Komplexität**. Diese ist zu realisieren, um Lehrern und Schülern eine einfachere Entwicklungsumgebung anbieten zu können. Nähere Ausführungen hierzu folgen ab Abschnitt 4.2.

3.4 Zusammenfassung

Auf Basis theoretischer Untersuchungen sowie unter Einbezug der praktischen Erfahrungen von Informatiklehrern wurden mehrere didaktische Anforderungen an eine IDE beschrieben. Durch den Autor wurden diese bewertet, selektiert und anschließend in „Soll“- und „Kann“-Anforderungen eingeteilt.
Hierbei zeigte sich, dass einige Funktionen in der Theorie durchaus sinnvoll erscheinen, in der Praxis jedoch ein zweckmäßiger Einsatz nicht angebracht erscheint oder teilweise die Unterstützung bzw. Bereitschaft zur Verwendung fehlt. Ferner kristallisierte sich heraus, dass sich Funktionen in der Praxis bewähren, die in der Literatur vorher keine (besondere) Erwähnung fanden. Hierfür steht besonders die **große Akzeptanz der Syntaxhervorhebung**. Gleichermaßen gilt diese Aussage auch für die **freie Verfügbarkeit** einer Entwicklungsumgebung.
Bei der Bewertung der Evaluationsergebnisse ist zu beachten, dass die **Probanden zumeist in Sekundarstufe I und II oder nur Sekundarstufe II unterrichten**. Entsprechend ist der Charakter der Sekundarstufe II in dieser Untersuchung stärker vertreten. Aus Sicht des Autors erklärt das auch die geringe Unterstützung (13 Prozent der Lehrer) des eingeschränkten Sprachumfanges. Jedoch gaben immerhin vier Lehrer die Verwendung der Umgebung Karel the Robot an. Diese zeichnet sich unter anderem durch den Einsatz eines reduzierten Sprachumfanges aus. Bei genauerer Betrachtung der Ergebnisse scheint der verringerte Sprachumfang nicht vornehmlicher Grund für den Einsatz von Karel the Robot zu sein. Vielmehr sprechend die Ergebnisse eher für die eine verringerte Komplexität und die damit verbundene Einfachheit der Umgebung.

Aus Sicht des Autors zeigten die Untersuchungsergebnisse auch, dass eine an schulische Bedürfnisse angepasste Entwicklungsumgebung nicht der einzige Schlüssel für einen erfolgreichen Informatikunterricht sein kann. Neben dieser sind unter anderem eine alltagsorientierte Aufgabenstellung sowie der Einsatz praxisnaher Sprach- sowie Programmierkonzepte nötig. In dieser Auffassung sieht sich der Autor durch Theorie und Praxis bestätigt. Derartige Konzepte sollten gleich wohl von einer eingesetzten Entwicklungsumgebung unterstützt werden.

4 Lazarus

4.1 Bisherige Arbeiten

Kapitel 4.1 greift die Arbeit aus der bereits vom Autor vorgelegten Belegarbeit auf und führt diese fort. Im Mittelpunkt der o. g. Arbeit stand die Entwicklung eines minimalen LAZARUS-Prototyps, der an die Anforderungen der Schulen angepasst ist. Für den Prototyp kam die frei verfügbare Entwicklungsumgebung LAZARUS zum Einsatz. Es handelt sich hierbei um einen *Open-Source*-Klon der DELPHI-IDE. Der Fokus des Projektes liegt dabei - neben dem Rapid Application Development - auf der Konvertierung von DELPHI-Anwendungen in LAZARUS-Anwendungen. LAZARUS steht unter dem Motto „Write once, compile everywhere!“ für verschiedene Plattformen wie *Windows*, *MacOS* sowie die diversen *Unix-Derivate* zur Verfügung.

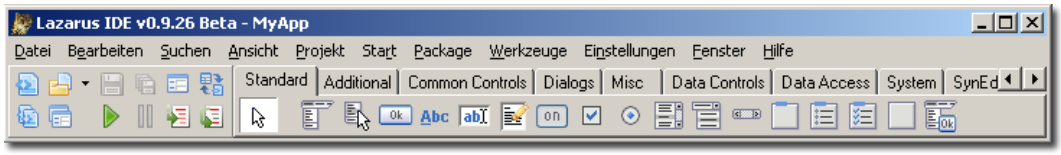

Abbildung 7: Menü sowie Komponentenpalette der Original-Version

In der Belegarbeit wurde besonders die Benutzerschnittstelle beschränkt. Ähnlich wie DELPHI bietet LAZARUS die Möglichkeit, grafische Benutzerschnittstellen mit Hilfe einer Komponentenpalette (vgl. Abbildung 7) sowie eines Objektinspektors „zusammenzuklicken“. Die größten Änderungen zielten auf diese beiden Teile ab. So wurden beide auf ein Minimum beschränkt. Nach einer Evaluation dieses minimalen Prototyps mit Schülern, konnte in der Folge ein überwiegend positives Fazit gezogen werden. Die Evaluation ergab jedoch auch, dass der Rahmen insgesamt zu eingeschränkt ist. Gleichlautend standen nach Lehrer- und Schülermeinung zu wenig Komponenten oder Eigenschaften zur Verfügung. **Ziel der folgenden Untersuchungen ist die Entwicklung einer möglichst frei skalierbare Lazarus-Version.** Dazu wird untersucht, welche Komponenten, Eigenschaften und Ereignisse in einer minimalen Version und welche in einer angepassten verfügbar sein sollten. Hierzu führte der Autor die im Kapitel 3.2 vorgestellte Evaluation durch. Die dort bereits vorgestellten und behandelten Fragen werden im

Weiteren nicht nochmals betrachtet.

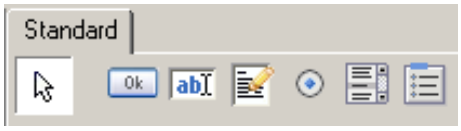

Abbildung 8: Die angepasste Komponentenpalette

Bei der ersten minimalen Version handelt es sich um einen „wirklichen“ Prototyp. Demgemäß gestaltet sich hierbei die Installation auch noch etwas umständlich. Es ist theoretisch zwar möglich, LAZARUS schlicht in das vorgesehene Verzeichnis zu installieren, aber für Abweichungen ist dann noch eine Vielzahl an „manueller Nacharbeit“ nötig. Dies betrifft vor allem das Bearbeiten von Pfaden für den mitgelieferten *FreePascal*-Compiler. Da dieser beträchtliche Aufwand Lehrern (und auch potenziellen Schüler) nicht zumutbar ist, wird zunächst eine verbesserte Installationsroutine „nachgeliefert“. Diese wurde mit Hilfe des *Nullsoft Scriptable Install System* (NSIS) der Firma NULLSOFT realisiert. Weitere Erläuterungen hierzu können in Anhang C nachgesehen werden.

4.2 Fortführung der Lehrer-Evaluation

Nachfolgend werden nun die in der Evaluation speziell zu DELPHI/LAZARUS gestellten Fragen näher betrachtet. Zunächst wurden prinzipielle Fragen über den potenziellen Einsatz gestellt. Ferner wurde gefragt, welche Probleme die Probanden im Informatikunterricht bei den Schülern am häufigsten beobachten. Anschließend wurde ermittelt, welche Komponenten, Eigenschaften und Ereignisse für welche Version als sinnvoll angesehen werden. Leider beschränkten sich die Antworten der Teilnehmer zum überwiegenden Teil auf Antworten zu den Komponenten. Für Eigenschaften und Ereignisse steht somit nur eine eingeschränkte Anzahl von Antworten zur Verfügung. Bei den ersten fünf Fragen kam eine *Likert*-Skala zum Einsatz. Die Antworten konnten im Bereich von 1 („Auf keinen Fall“) bis 5 („Auf jeden Fall“) gegeben werden.

Zunächst wurde folgende Frage gestellt:

„Würden Sie eine angepasste* Lazarus-*Version Ihrer derzeit verwendeten IDE vorziehen?“

Wie aus Abbildung 9 ersichtlich, ist hier durchaus eine **große Skepsis der Teilnehmer** zu erkennen. Immerhin votierten ca. 43% hier neutral oder wenigstens tendenziell positiv. Zwei Lehrer schlossen die Verwendung aus.

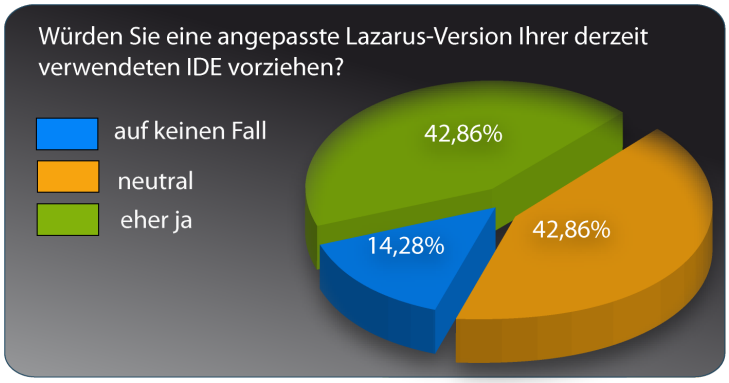

Abbildung 9: „Würden Sie eine angepasste Lazarus-*Version Ihrer derzeit verwendeten IDE vorziehen?“*

Ein Problem an der Delphi-IDE ist deren fehlende Möglichkeit, Anpassungen vornehmen zu können (vgl. Belegarbeit des Autors). Das trifft beispielsweise auf die Komponentenpalette zu. Lazarus hingegen kann über das Bearbeiten des Quellcodes und anschließendes erneutes Übersetzen der Entwicklungsumgebung angepasst werden. Darauf zielten auch die nächsten Fragen ab:

„Würden Sie Änderungen (im Quellcode/via Einstellungen) vornehmen, um die IDE besser an den Unterricht anzupassen?“

Erwartungsgemäß stellten sich die Ergebnisse unterschiedlich dar. Wie aus Abbildung 10 erkennbar ist, besteht bei den Probanden eine große Bereitschaft, entsprechende Änderungen vorzunehmen um die IDE somit besser an den Unterricht anpassen zu können. Diese Bereitschaft ist jedoch in der Regel auf das Verändern über Einstellungen beschränkt. Diese Aussage trifft immerhin auf über 70% der Probanden zu. Bei der Bereitschaft zur Bearbeitung des Quellcodes kehrt sich diese Aussage nahezu in das Gegenteil um. **Fazit: Die Bereitschaft zur eigen-**

ständigen Anpassungen einer IDE ist vorhanden. Arbeit am Quellcode wird jedoch nicht gewünscht.

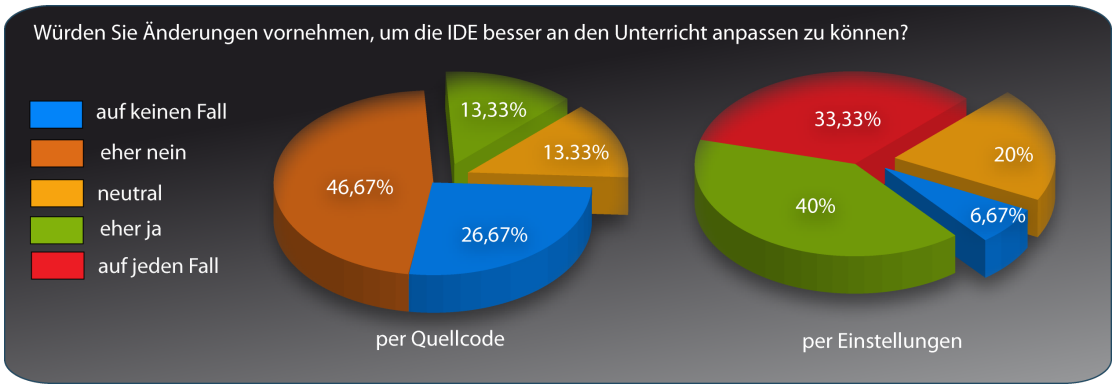

Abbildung 10: „Würden Sie Änderungen vornehmen, um Ihre IDE an den Unterricht anpassen zu können?“

Für den Autor war es außerdem wichtig zu erfahren, was für eine Version die Teilnehmer der Evaluation prinzipiell bevorzugen. Zur Auswahl stand hier eine frei skalierbare oder vorkonfigurierte Version (Minimal, Erweitert, Maximal). Aus den gegebenen Antworten kann jedoch **keine definitive Aussage** getroffen werden. Tendenziell entschieden sich ca. 60% der Befragten für eine vorkonfigurierte Version.

Im Laufe dieser Arbeit wurden bereits Anforderungen wie ***Fehlertoleranz*** und ***Robustheit*** diskutiert. Es ist zu beachten, dass es sich bei LAZARUS noch immer nicht um eine finale Anwendung im *„1.0-Status“* handelt. Neben einer stabilen Basisversion stehen auch häufige Aktualisierungen zur Verfügung. Die ständige Erweiterung um neue Funktionen führt oftmals auch zu neuen Fehlern in der Anwendung. Hier kommen besonders die geschilderten Anforderungen zum Tragen. Deshalb erschien es bedeutsam zu erfragen, was den Lehrern aus praktischer Sicht wichtiger erscheint:

„Ziehen Sie die langfristige Arbeit mit einer stabilen Version einer Entwicklungsumgebung, der einer Variante mit regelmäßigen Aktualisierungen vor?“

Wie die Abbildung 11 belegt, bevorzugt mit **70%** eine deutliche Mehrheit die **langfristige Arbeit mit einer stabilen Version**. Bei Abzug der neutralen Meinungen (ca. 20%), entschieden sich nur knapp 13% für einen ständigen Versionswechsel.

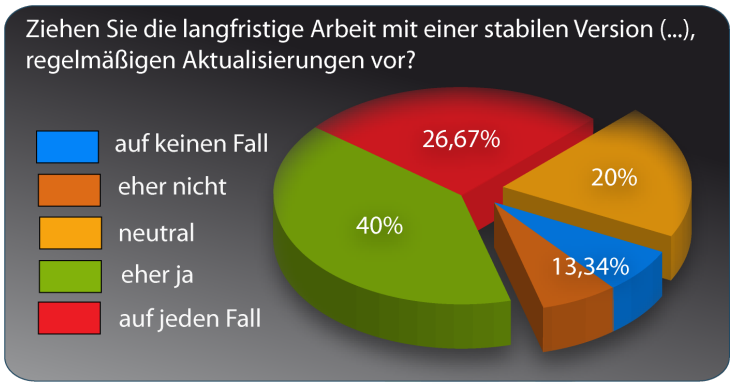

Abbildung 11: „Ziehen Sie die langfristige Arbeit mit einer stabilen Version einer Entwicklungsumgebung, der einer Variante mit regelmäßigen Aktualisierungen vor?“

Zusammenfassend kann aus diesen fünf einführenden Fragen geschlussfolgert werden, dass prinzipiell eine **große Bereitschaft zum Lazarus-Einsatz** besteht. Dabei wird jedoch bevorzugt, langfristig mit einer stabilen Version zu arbeiten und Änderungen bequem über ein Menü vornehmen zu können.

Bereits in der zuvor genannten Belegarbeit des Autors wurde gefragt, welche Probleme aus Sicht der Befragten im Unterricht im Umgang mit DELPHI beobachtet werden. Da nun eine größere Anzahl an Lehrern zur Verfügung stand, stellte der Autor diese Frage erneut. Neben der Möglichkeit weitere Angaben zu machen, standen dabei folgenden Varianten zur Auswahl:

- Verwirrung über zu viele Funktionen
- Suche nach den benötigten Komponenten
- Suche nach den benötigten Eigenschaften oder Ereignissen
- Ausprobieren aller möglichen Komponenten oder Eigenschaften
- Mangelnde Fokussierung auf die Aufgabe
- Prinzipielle Probleme im Umgang mit der Entwicklungsumgebung
- Fehlendes Verständnis für die Programmierung

Positiv kann konstatiert werden, dass kaum ein Lehrer bei den Schülern prinzipielle Probleme beim Umgang mit der Entwicklungsumgebung vermerkte. Hingegen gab die Mehrzahl der Teil-

nehmer (**60%**) ein **fehlendes Verständnis der Schüler für die Programmierung** an. Ein Schwerpunkt der o. g. Belegarbeit bestand auch darin, die **Suche nach den benötigten Komponenten** zu vereinfachen. Befragte Lehrer beschrieben diesen Umstand zuvor als ein großes Problem. Diese Aussage wurde auch in dieser Evaluation **durch die Probanden bestätigt**. Immerhin beobachten mehr als die Hälfte der Befragten dieses Problem. Weiterhin stellten fast **47% eine mangelnde Fokussierung** der Schüler auf die Aufgabe fest. Deren Konzentration wird u. a. durch Suchen in der DELPHI-IDE oder der regulären LAZARUS-Version eingeschränkt. Die zahlreichen Antworten auf die weiteren Auswahlmöglichkeiten (Ausprobieren, Verwirrung) bekräftigen diese Aussagen.

Einzelne Lehrer benannten darüber hinaus ***unverständliche Fehlermeldungen*** sowie eine ***komplizierte Projektverwaltung als Probleme.***

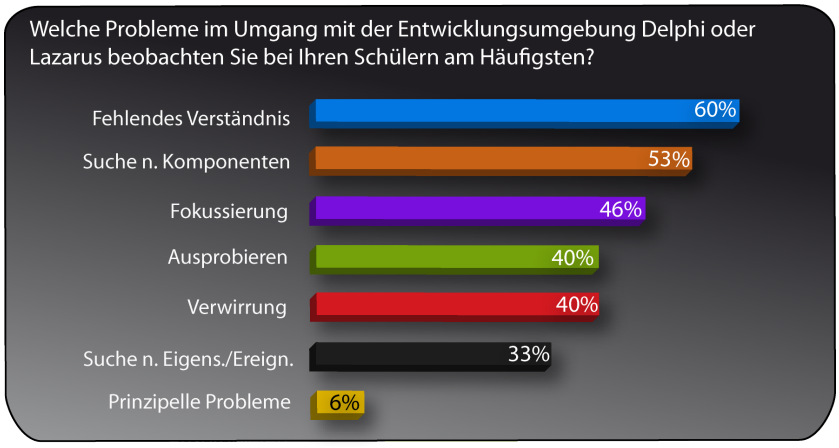

Abbildung 12: „Welche Probleme im Umgang mit der Entwicklungsumgebung DELPHI *oder* LAZARUS *beobachten Sie bei Ihren Schülern am Häufigsten?“*

Im Rahmen der Belegarbeit des Autors wurde, neben weiteren Änderungen, auch die Benutzerschnittstelle von LAZARUS bereits minimiert. Verwiesen sei hier etwa auf den minimierten Objektinspektor. Für die weitere Arbeit war es nun wichtig zu erfahren, welche Komponenten, Eigenschaften sowie Ereignisse die Befragten im Unterricht tatsächlich benötigen. Welche erwarten sie in einer minimalen Version? Was betrachten sie dabei als Zusatz oder Zugabe?

Den Ausgangspunkt für diese Untersuchung bildeten die Komponenten. Dabei galt es zunächst zu ermitteln, welche Komponenten nach Ansicht der Befragten Bestandteil einer Minimalversion

Komponente	Anzahl Nennungen	Komponente	Anzahl Nennungen
TEdit	12	TMainMenu	8
TButton	12	TMemo	8
TCheckBox	11	TGroupBox	7
TLabel	11	TScrollBar	3
TListBox	11	TPopupMenu	2
TComboBox	10	TCheckGroup	2
TRadioGroup	9	TActionList	1
TRadioButton	9	TToggleBox	0
TPanel	8		

Tabelle 13: „Welche Standard-Komponenten benötigen Sie mindestens für den Unterricht?“

sein müssen. Hierbei dienten die für den Prototyp gewählten Komponenten als erste Vorlage. Diese wurde um einige, in der damaligen Evaluation vermissten Komponenten, erweitert. Für die Probanden standen somit zunächst 17 Komponenten zur Auswahl. Bei insgesamt 15 empfangenen Bögen beantworteten 12 Lehrer diese speziellen Fragen. Aus Gründen der Übersichtlichkeit werden die Antworten auf diese und die folgenden Fragen in tabellarischer Form präsentiert. Tabelle 13 verdeutlicht dabei die Ergebnisse für die eben genannte Frage.

Zusätzlich wurde erfragt, welche Komponenten darüber hinaus als sinnvoll erachten werden. Neben den vier vorgegebenen Komponenten (*TBitBtn*, *TSpeedButton*, *TStaticText* sowie *TImage*) hatten die Teilnehmer dann die Möglichkeit, eigene Komponenten zu benennen. Tabelle 14 stellt das Ergebnis dieser Frage dar.

Analog zu den Komponenten wurde gefragt, welche Eigenschaften genau benötigt werden.

Komponente	Anzahl Nennungen
TImage	11
TBitBtn	5
TStringGrid	2
Komp. zur Datenbankanbindung	2
TSpeedButton	1
TTimer / TIdleTimer	1
TStaticText	1
TDBGrid	1
TPageControl	1
Dialoge	1

Tabelle 14: „Welche Standard-Komponenten erachten Sie darüber hinaus als sinnvoll?“

Auf die Angabe des exakten Namens konnte hierbei verzichtet werden. Dieser Umstand ist der

Vielzahl an Eigenschaften in DELPHI sowie LAZARUS geschuldet. Dabei nutzen unterschiedliche Komponenten teilweise ähnliche Eigenschaften. Der LAZARUS-Prototyp ist bereits auf die Eigenschaften *Name*, *Caption*, *Visible*, *Text*, *Checked* sowie *Items* beschränkt. Leider formulierten nur sieben Lehrer die in Tabelle 15 aufgelisteten Eigenschaften.

Komponente	**Anzahl Nennungen**	**Komponente**	**Anzahl Nennungen**
Visible	5	Top	1
Caption	4	Left	1
Font	4	Items	1
Color	3	Hint	1
Name	3	ShowHint	1
Enabled	2	ParentFont	1
Height	2	ParentShowHint	1
Width	2	TabOrder	1
Text	2	WordWrap	1
Align	1		

Tabelle 15: „Welche Eigenschaften von Komponenten benötigen Sie mindestens für den Unterricht (exakter Name nicht notwendig)?“

Auch hier konnten weitere, als sinnvoll erachtete, Eigenschaften angegeben werden. Diese Möglichkeit nutzten lediglich drei Lehrer. Ein Lehrer verwies hierbei noch einmal auf Formatierungsmöglichkeiten. In einer weiteren Nennung wurde die Eigenschaft *AutoSize* empfohlen. Ein dritter Lehrer gab folgende Anregung (Komponente, zugehörige Eigenschaft):

- *TLabel* (*WordWrap*)
- *StringGrid* (*FixedCols, FixedRows, DefaultColWidth, DefaultRowHeight, ColCount, RowCount, Borderstyle*)
- *TImage* (*Picture*)

In Analogie zur bisherigen Verfahrensweise wurden im Weiteren auch die benötigten Ereignisse abgefragt. Wiederum bestand die Möglichkeit frei und ohne exakten Namen zu antworten. Die Ergebnisse werden in Tabelle 16 dargestellt.

Bereits die vom Autor verfasste Belegarbeit beschränkte LAZARUS ausschließlich auf das nun noch einmal von den Lehrern gewünschte *OnClick*-Ereignis. Auf die Frage, welche Ereignisse darüber hinaus als sinnvoll für eine minimale Version erachten würden, gab es lediglich zwei

Komponente	Anzahl Nennungen
OnClick	7
OnChange	3
OnMouseMove	2
OnDblClick	1
OnCreate	1
OnKeyPress	1
FormCreate	1

Tabelle 16: „Welche Ereignisse benötigen Sie mindestens für den Unterricht?“

Rückmeldungen: Erneut wurden dabei je einmal das *OnChange*-Ereignis sowie das *OnMouse-Move*-Ereignis benannt.

4.3 Anpassungen

4.3.1 Auswahl der Konfigurationen

Wie bereits dargelegt, besteht die Zielstellung dieser Arbeit insbesondere darin, neben der freien Skalierbarkeit auch unterschiedliche Konfigurationen für eine IDE zu definieren. Idealerweise sind dieses eine ***minimale***, eine ***erweiterte*** sowie eine ***vollständige*** **Lazarus-*Konfiguration***. Hierbei wird auf den bereits vorhandenen LAZARUS-Prototypen aufgebaut. Änderungen wie die Reduktion der Seiten des Objektinspektors oder zusätzliche Einstellungen bleiben hierbei erhalten. Für die Schulen bleibt somit ein eingeschränkter Anwendungsrahmen bestehen. Alternativ besteht für Lehrer weiterhin die Möglichkeit, eine unveränderte LAZARUS-Version zu verwenden.

Komponenten

Die Definition einer minimalen Konfiguration beginnt mit der Komponentenpalette und ihren Komponenten. Die prototypische Version enthält derzeit bereits die Komponenten *TButton*, *TEdit*, *TMemo*, *TRadioButton*, *TListBox* sowie *TRadioGroup*. Alle sechs vorhandenen Komponenten wurden auch in der aktuellen Evaluation von den Lehrern vielfach benannt. Sie bilden deshalb die Basis für eine minimale Konfiguration. Aus Tabelle 13 ist eine Zweiteilung der benötigten Komponenten abzulesen (weniger sowie häufiger benötigte Komponenten). Die Grenze wird dabei zwischen den beiden Komponenten *TGroupBox* sowie *TScrollbar* gezogen. Während

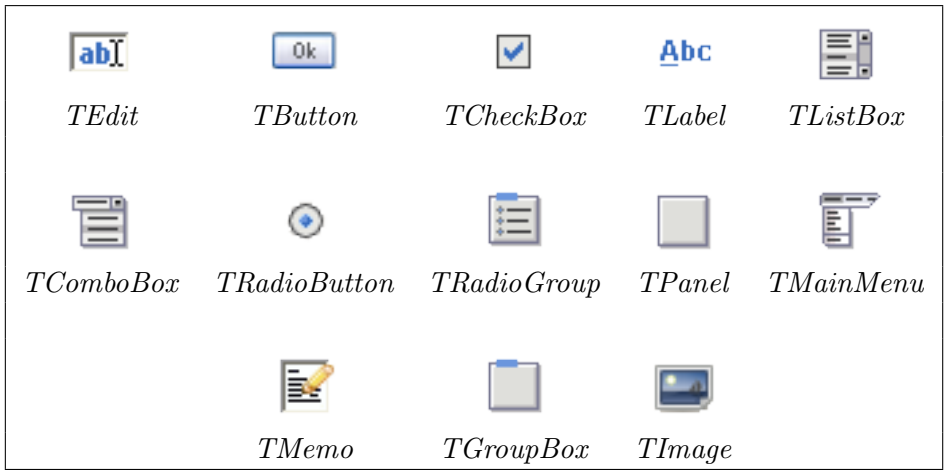

Tabelle 17: Komponenten der minimalen Konfiguration

Erstere noch von mehr als der Hälfte der Probanden (7 Nennungen) benötigt wird, kam die zuletzt genannte auf nur noch 3 Nennungen. **Die ersten zwölf, der in Tabelle 13 aufgeführten Komponenten, werden für eine minimale Konfiguration ausgewählt.** Von denen, durch die Befragten zusätzlich gewünschten Komponenten, wurde nur die Komponenten *TImage* mit 11 Nennungen sowie *TBitBtn* mit 5 Nennungen in signifikanter Anzahl erwähnt. Letztere wird auf Grund der geringen Anzahl an Nennungen nicht für eine minimale Konfiguration berücksichtigt. Demnach wird nur die Komponente *TImage* zu dieser Konfiguration hinzugefügt. Somit umfasst sie insgesamt die in Tabelle 17 dargestellten **13 Komponenten**.
Weitere Komponenten, mit wenigstens einer Erwähnung, werden in einer erweiterten Konfiguration zusammengefasst. Für Datenbanken relevante Komponenten werden unter diesem Oberbegriff weiter geführt, da es sich hierbei um eine Vielzahl an relevanten Komponenten handelt. Gleiches gilt für jene Komponenten, welche Dialoge zum Beispiel zum Öffnen und Speichern von Dateien bereitstellen. Insgesamt handelt es sich bei der ***erweiterten Konfiguration*** somit um die in Tabelle 18 dargestellten zusätzlichen Komponenten.

Eigenschaften

Das Verfahren zur Definition der Eigenschaftskonfigurationen gestaltet sich in analoger Weise wie bei den Komponenten. Es sei hervorgehoben, dass **sämtliche Eigenschaften im Lazarus-Sprachumfang weiterhin prinzipiell zur Verfügung stehen**. Im Weiteren geht es also

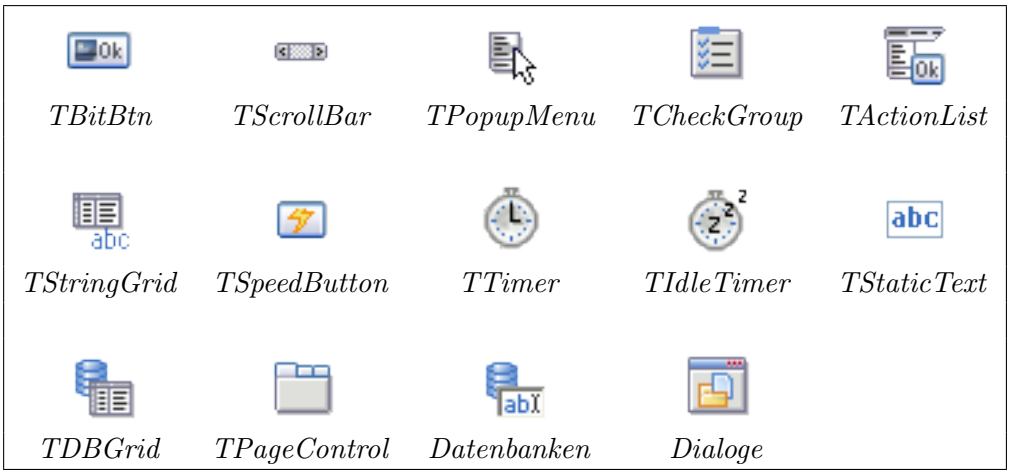

Tabelle 18: Komponenten der erweiterten Konfiguration

darum, die Anzeige der möglichen Eigenschaften im Objektinspektor zu reduzieren. Das soll einerseits insbesondere die erwähnte Verwirrung der Schüler reduzieren. Andererseits soll somit die Möglichkeit des Ausprobierens reduziert und in der Folge die Fokussierung auf das Wesentliche verbessert werden. In der prototypischen Version waren bereits die Eigenschaften *Name*, *Caption*, *Visible*, *Text*, *Checked* sowie *Items* verfügbar. Auf Grund der Anzahl ihrer Nennungen (jeweils mehr als 2) in dieser Evaluation werden diese erneut in eine minimale Konfiguration übernommen. Eine Ausnahme bilden die Eigenschaften *Checked* und *Items*. In der minimalen Konfiguration der Komponenten steht zum Beispiel die Komponente *TCheckBox* zur Verfügung. Um diese sinnvoll verwenden zu können, ist die Überprüfung des Wertes der Eigenschaft Checked notwendig. Analog gilt dies für die Eigenschaft *Items* der Komponente *TListBox*. Auch hier basiert die Entscheidung für die Einordnung in die jeweilige Konfiguration auf der Anzahl an Nennungen. Zusätzlich werden für Abhängigkeiten von Komponenten weitere Eigenschaften in die Konfiguration übernommen. Dies gilt zum Beispiel für *MaxLength* (für *TMemo*) oder

Konfiguration	**Eigenschaften**
Minimal	Name, Caption, Visible, Text, Checked, Items, Font, Color, Enabled, Height, Width, MaxLength, Picture, Columns
Erweitert (zusätzlich)	Align, Left, Top, Hint, ShowHint, ParentFont, TabOrder, ParentShowHint, WordWrap, FixedCols, FixedRows, DefaultColWidth, DefaultRowHeight, ColCount, RowCount, Borderstyle, Glyph, State, Interval, DataSource, DataField sowie Eigenschaften zum Datenbankzugriff

Tabelle 19: Konfiguration der Eigenschaften

Picture (für *TImage*) in der minimalen Konfiguration. In der erweiterten Konfiguration sind zum Beispiel Eigenschaften für den Datenbankzugriff zusätzlich nötig. Eine Übersicht über die Konfiguration der Eigenschaften bietet Tabelle 19.

Ereignisse

Einen wesentlich geringeren Umfang haben die benötigten Ereignisse. Im Prototyp erfolgte die Reduzierung auf ein Ereignis - *OnClick*. Die damalige Wahl wurde von den Lehrern nahezu bestätigt. Immerhin erreichte es erneut 7 Nennungen. Kein anderes Ereignis erreichte ähnliche Zahlen. Mehrfachnennungen gab es ferner nur für die Ereignisse *OnChange* sowie *OnMouseMove*. Da die Fragen bezüglich der Ereignisse nur durch eine geringere Anzahl von Teilnehmern beantwortet wurden, fiel eine entsprechende Einordnung schwerer. Drei Ereignisse wurden in eine ***minimale Konfiguration*** übernommen:

- *OnClick*
- *OnChange*
- *OnMouseMove*

Da die Ereignisse *OnDblClick*, *OnCreate*, *OnKeyPress* sowie *FormCreate* wenigstens eine Nennung erreichen konnten, wurden diese in die ***erweiterte Konfiguration*** übernommen.

4.3.2 Realisierung

In der ersten konzeptionellen Phase an der vorliegenden Arbeit trat das LAZARUS-Entwicklerteam an den Autor heran. Das Team plant, das LAZARUS-Projekt perspektivisch um ein Package für den Bildungsbereich zu erweitern. Sie erhielten über die Internetpräsentation der betreuenden Professur Kenntnis von der bisherigen Arbeit und zeigten starkes Interesse an den Ergebnissen und Erkenntnissen des Autors. In der Folge konnte mit Zustimmung des betreuenden Hochschullehrers, Herrn Prof. Friedrich, sowie des Betreuers der Arbeit, Herrn Dr. Rohland, eine konstruktive Zusammenarbeit vereinbart werden. Sie ermöglichte es, **wesentliche Vorstellungen des Autors** bereits jetzt zu realisieren und **in die zukünftige offizielle Lazarus-Erweiterung einfließen zu lassen**.

Im Rahmen dieser Zusammenarbeit entsteht ein eigenständiges LAZARUS-Package **„Lazarus for Education“**. Dieses Package kann zusätzlich zur vorhandenen Entwicklungsumgebung installiert werden. Dadurch wird es künftig möglich sein, Anpassungen der LAZARUS-Oberfläche vorzunehmen. Bisher waren hierfür Änderungen am LAZARUS-Quellcode selbst erforderlich. Diese Erweiterung findet hierbei für den Anwender auf Basis der Einstellungen bzw. Optionen statt. Ihm werden dann zusätzliche Optionen („Bildungseinstellungen“) angeboten.

Die Entwicklung der o. g. Lazarus-Erweiterung ist mit Vorlage dieser Arbeit noch nicht abgeschlossen. Der Grund hierfür ist einerseits in der Art der Änderungen zu suchen. Um diese realisieren zu können, musste der Autor erneut Eingriffe am LAZARUS-Quellcode vornehmen. Um diese Änderungen vollständig in eine eigenständige LAZARUS-Erweiterung integrieren zu können, sind zuvor grundsätzliche Änderungen an der offiziellen LAZARUS-Version sowie der verwendeten Klassenbibliothek (*LCL*) durch die Entwickler erforderlich. Anderseits will das Entwickler-Team noch weitere, bisher nicht näher bezeichnete Optionen hinzufügen. Mit der vorliegenden Arbeit reicht der Autor deshalb als Zwischenschritt eine **eigenständige Lazarus-Version** ein. In diese wurden die gewünschten Änderungen (soweit möglich) bereits in Form des zukünftigen Pakets integriert. War dies nicht möglich, wurden die Änderungen direkt am Quellcode der IDE durchgeführt. Die durch den Autor vorgenommenen Änderungen werden nun in der Folge erläutert. Zunächst folgt ein kurzer Überblick über die Struktur des integrierten Packages.

Paketstruktur

Grundlage einer jeden Erweiterung bildet die zugehörige Package-Datei. Bei der „Education“-Erweiterung ist dies die Datei *educationlaz.lpk*. Sie beschreibt das Paket mittels *XML*. Sie gibt ferner an, welche Klassen im Paket enthalten sind, welche benötigt werden und stellt zusätzliche Informationen wie Versionsnummer oder Beschreibung bereit. Mit Hilfe dieser Datei erfolgt außerdem die spätere Installation durch den Anwender. Die zur ausgelieferten Version zu Grunde liegende *educationlaz.lpk* ist im Anhang B aufgeführt.

Die im Paket enthalten Klassen fügen der Entwicklungsumgebung neue Funktionalitäten hinzu. Die folgende Tabelle gibt einen Überblick über die beteiligten Dateien/Units sowie ihre Funktion.

Datei / Unit	Funktion
Eduenvoptsframe.pas	Definiert den Dialog zur De-/Aktivierung der Bildungseinstellungen.
Eduoptions.pas	Definiert einen eigenen LAZARUS-Optionentyp. Dieser dient speziell für die Optionen der Bildungserweiterung.
EduPkgSystem.pas	Definiert Optionen für das LAZARUS-Paketsystem.
Educomppalette.pas	Dialog zur Konfiguration der Komponentenpalette.
Edumenu.pas	Dialog um Menüs ein- oder ausblenden zu können.
Edunewprogram.pas	Erstellt per Button automatisch eine Vorlage für ein *Pascal*-Programm.
Edupropsevents.pas	Dialog zum Ändern der Anzeige von Eigenschaften und Ereignissen im Objektinspektor.
Eduoipages.pas	Dialog zur Auswahl der zusätzlichen ObjektInspektor-Registerkarten.
Eduspeedbuttons.pas	Dialog zur Auswahl der benötigten Speedbuttons.

Tabelle 20: Die Units der Erweiterung

Neben diesen Units hat auch die *XML*-Datei *Education.xml* eine zentrale Bedeutung. Sämtliche vom Nutzer gewählte Bildungseinstellungen werden innerhalb dieser Konfigurationsdatei gespeichert. Existiert diese noch nicht, wird sie bei der ersten durchgeführten Änderung im LAZARUS-Konfigurationsverzeichnis angelegt. Während des Installationsprozesses der LAZARUS-Version des Autors kann der Anwender sich für eine der drei vorgestellten Konfigurationen bereits entscheiden. Diese Auswahl basiert auf drei unterschiedlichen Versionen dieser *Education.xml*.

Vorarbeit

Das Grundgerüst des Paketes wurde vom Lazarusentwickler auf Basis der aktuellen LAZARUS-Entwicklerversion erstellt. Auf Grund einer Vielzahl von Änderungen an LAZARUS ist das Paket zunächst nur mit dieser aktuellen Version lauffähig. Mit Abschluss seiner Diplomarbeit wollte der Autor den Lehrern und Schülern jedoch den Einsatz dieser Erweiterung innerhalb einer stabilen LAZARUS-Version ermöglichen. Er entschied sich deshalb, **das Paket in die aktuelle stabile Lazarus-Version zu integrieren (0.9.28.2)**. Hierfür mussten jedoch einige Units um Funktionen aus der Entwicklerversion erweitert werden. Dem Interface *MenuIntf* musste die Funktion *TIDEMenuItem.HasAsParent(Item: TIDEMenuItem): boolean;* hinzugefügt werden. Das Interface *LazIDEIntf* wurde um das Flag *nfAskForFilename* erweitert. Im Interface *ProjectIntf*

wurden die Flags *FBuildFileIfActive*, *FRunFileIfActive* sowie die für das erste Flag verwendete Eigenschaft *property BuildFileIfActive: boolean read FBuildFileIfActive write FBuildFileIfActive;* hinzugefügt. Außerdem wurde eine aktualisierte *Lazarus Component Library* verwendet. Mit diesen Maßnahmen ist es nun möglich die Erweiterung mit der aktuell als stabil veröffentlichten Lazarus-Version zu verwenden.

Komponentenpalette

Die Unit zur Änderung der Komponentenpalette entstand auf Vorschlag des Autors in Zusammenarbeit mit dem Lazarus-Entwickler. Hierbei wurde im Rahmen der regulären Umgebungseinstellungen zunächst ein zusätzlicher Dialog geschaffen. Dieser Dialog gestattet es, die Sichtbarkeit von Komponenten zu verändern. Die Komponenten werden mit Hilfe eines *TTreeView* in Baumform dargestellt. Um die Komponenten entsprechend der bevorzugten Konfiguration auswählen zu können, wurden entsprechende Buttons hinzugefügt. Basis für die Auswahl ist die Prozedur *TEduCompPaletteFrame.ShowSelected(extended: boolean);*. Das folgende Listing zeigt auszugsweise das Setzen dieser Werte für Komponenten der erweiterten Konfiguration. Die vollständige Funktion kann im Anhang D eingesehen werden.

Listing 1: Ändern der Sichtbarkeit

```
1 for k := 0 to 25 do begin
2  if (CompareText (CompName , ExtendedComponents[k] )=0) then begin
3     EduComponentPaletteOptions.ComponentVisible[CompName]:=true;
4     Node.StateIndex:=ShowImgID;
5  end;
6 end;
```

Alle vom Nutzer ausgewählten Komponenten werden in der *Education.xml* eingetragen. Das im weiteren Verlauf folgende Listing zeigt den korrespondierenden Eintrag für die minimale Konfiguration. Die einzelnen Komponenten können (neben der Wahl über Buttons) auch per Mausklick in der Baumansicht an- oder abgewählt werden. **Die freie Skalierbarkeit der Komponentenpalette ist somit gewährleistet.** Abbildung 13 zeigt das Menü für die Einstellungen zur Komponentenpalette.

Listing 2: Komponenten in der minimalen Konfiguration

```
1 <ComponentPalette>
2   <Visible Item1="TButton" Item2="TCheckBox" Item3="TComboBox"
3       Item4="TEdit" Item5="TGroupBox" Item6="TImage" Item7="TLabel"
4       Item8="TListBox" Item9="TMainMenu" Item10="TMemo" Item11="TPanel"
5       Item12="TRadioButton" Item13="TRadioGroup" Count="13" />
6 </ComponentPalette>
```

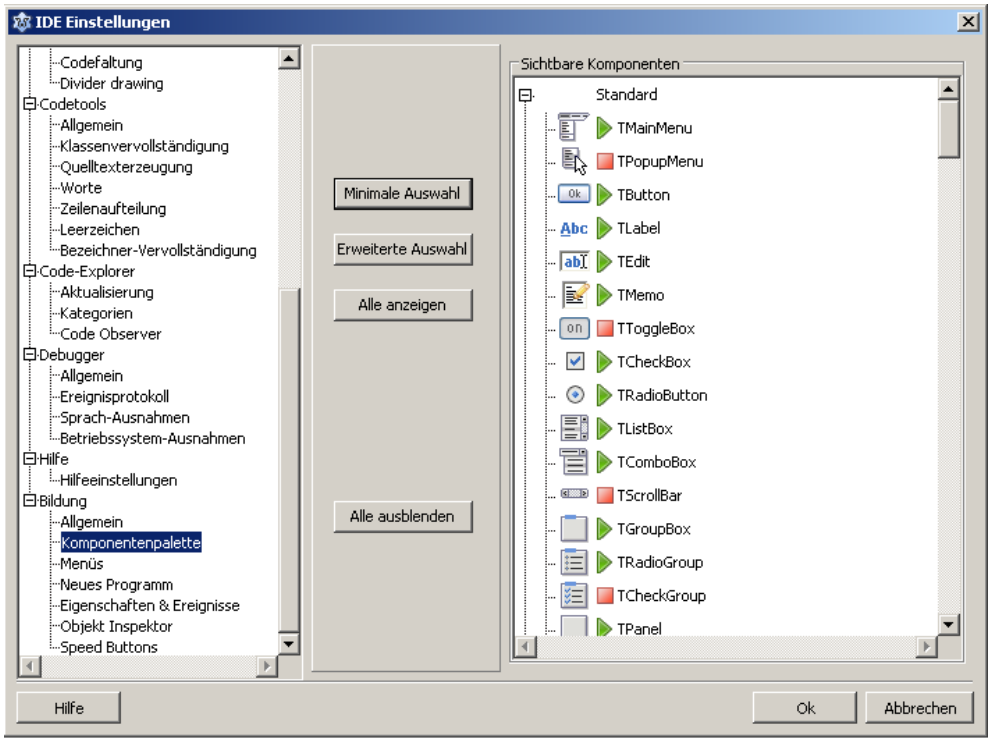

Abbildung 13: Das Menü zur Komponentenpalette

Anzeige von Eigenschaften und Ereignissen im Objektinspektor

(Anmerkung: Die Beschränkungen von Eigenschaften und Ereignisse beziehen sich, wie auch im Prototyp, nur auf die Anzeige im ObjektInspektor. Die Entwicklung kann weiterhin mit der gesamten Mächtigkeit der Programmiersprache erfolgen.)

Dank der Erweiterung besteht nun die Möglichkeit, für die Eigenschaften sowie Ereignisse eine separate Konfigurationsstufe auszuwählen. Die Auswahl erfolgt über RadioButtons. Die vom Nutzer gewählte Option wird in der Folge in der *Education.xml* gespeichert. Das folgende Listing zeigt jeweils die minimale Konfiguration.

Listing 3: Konfiguration der Eigenschaften und Ereignisse

```
1 <PropsEvents PropsMinimal="True" PropsExt="False" PropsFull="False"
2   EventsMinimal="True" EventsExt="False" EventsFull="False" />
```

Um Eigenschaften sowie Ereignisse im Objektinspektor ausblenden zu können, existiert gegenwärtig innerhalb des LAZARUS-Quellcode kein globales Flag. Deshalb waren an dieser Stelle Eingriffe am Quellcode der IDE notwendig. Hierfür wurde das Einlesen der Eigenschaften sowie Ereignisse innerhalb der Unit *PropEdits* modifiziert.

Um die gespeicherten Werte aus der XML-Datei einzulesen, wurde die Prozedur *getPropsAndEventsConfiguration();* hinzugefügt. Sie liest die Konfigurationsdatei aus und speichert in lokalen Variablen die jeweilige Belegung. Auf dieser Basis erfolgt anschließend das Einlesen von Eigenschaften und Ereignissen. Beide Funktionen sind im Anhang beschrieben. Abbildung 14 zeigt das Menü zur Auswahl der Konfigurationen.

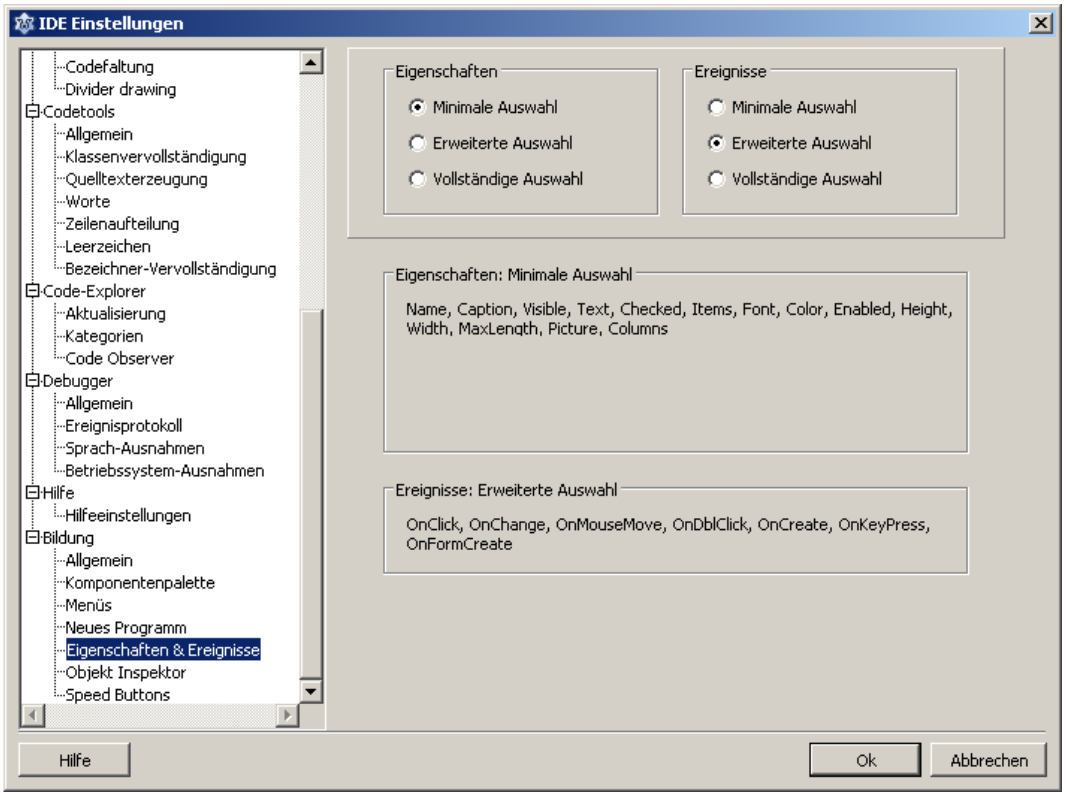

Abbildung 14: Das Menü für Eigenschaften und Ereignisse

Ausblenden von Registerkarten im Objektinspektor

Wie zuvor beim Prototyp, sollte auch hier wiederum das Ausblenden von Registerkarten ermöglicht werden. Dies beschränkt sich auf die Karten *Favoriten* sowie *Bedingte Eigenschaften*. Werden die Registerkarten Ereignisse oder Eigenschaften nicht verwendet, kann wahlweise der ObjektInspektor komplett abgeschaltet werden.

Für die Änderungen war auch hier der Eingriff in die Unit des ObjektInspektors nötig. Der Dialog des ObjektInspektors wurde um die Prozedur *TObjectInspectorDlg.getOIPageConfig();* erweitert. Sie liest analog die gewählte Einstellung aus und beschreibt entsprechend die in dieser Unit für die Sichtbarkeit der Registerkarten vorgesehenen Variablen. Auch diese Prozedur ist im Anhang aufgeführt. In der *Education.xml* wird für jede der beiden möglichen Registerkarten ein boolscher Wert gespeichert. In der minimalen Konfiguration sind beide ausgeblendet.

Listing 4: Konfigurationseintrag der Registerkarten

```
1 <OIPages OIPageFavs="False" OIPageRestricted="False" />
```

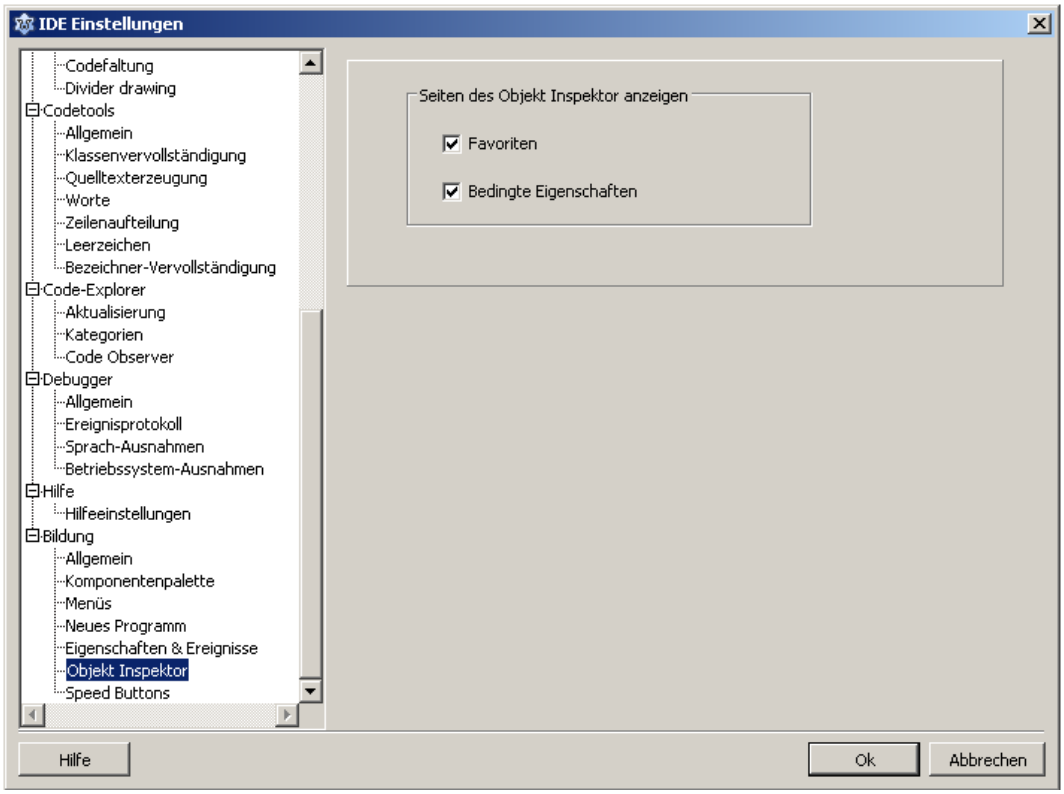

Abbildung 15: Das Menü der Registerkarten

SpeedButtons

Im Prototyp wurden *SpeedButtons* statisch hinzugefügt oder entfernt. Für die Erweiterung schuf der Autor die Möglichkeit, die verfügbaren SpeedButtons frei auswählen zu können. Die Anwender können dem zu Folge selbst entscheiden, welche *SpeedButtons* erforderlich bzw. geeignet erscheinen. Die Funktionsweise ist hierbei analog zur Klasse der Komponentenpalette. **Zusätzlich wurde eine spezielle Auswahl an *SpeedButtons* erstellt. Sie basiert auf der Auswahl der *SpeedButtons* im Prototyp.** Dieser sollte auf Projekte sowie Units beschränkt werden. Das separate Speichern, Laden oder Öffnen von Units sowie Formularen über *SpeedButtons* kann somit unterbunden werden. Gesonderte Konfiguration (minimal oder erweitert) sind nicht vorhanden. Wie bekannt, erfolgt die Speicherung der gewählten Buttons in der *Education.xml*:

Listing 5: Konfigurationseintrag der SpeedButtons

```
1 <SpeedButtons>
2   <Visible Item1="EduNewSingleFileProgramBtn" Item2="NewFormSpeedBtn"
3     Item3="OpenFileSpeedBtn" Item4="PauseSpeedButton" Item5="RunSpeedButton"
4     Item6="SaveAllSpeedBtn" Item7="StepIntoSpeedButton"
5     Item8="StepOverpeedButton" Item9="StopSpeedButton"
6     Item10="ToggleFormSpeedBtn" Count="10" />
7 </SpeedButtons>
```

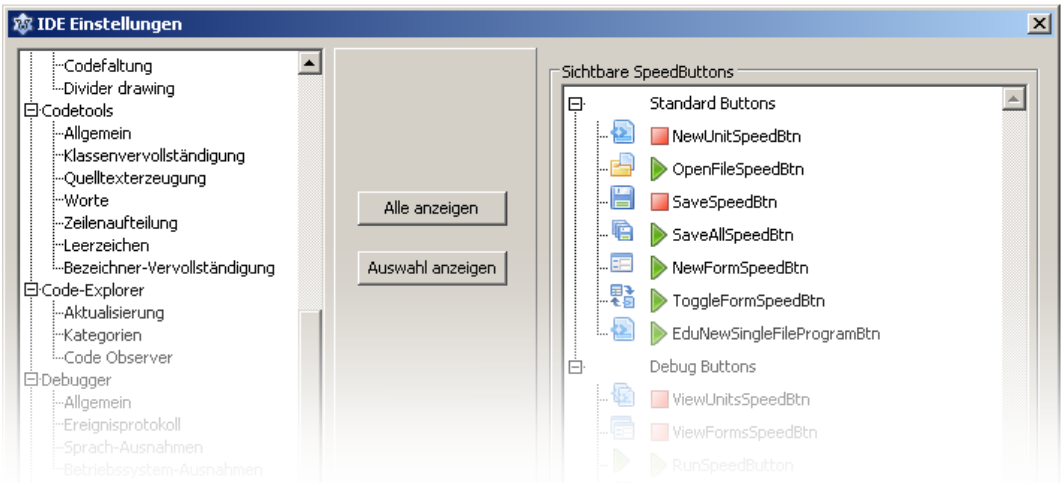

Abbildung 16: Das Menü der SpeedButtons

Menüeinträge

Die Möglichkeit des Ein- und Ausblendens von Menüs wurde bereits vom LAZARUS-Entwickler

für die Erweiterung realisiert. Vom Autor wird hierzu eine Vorkonfiguration im Rahmen der *Education.xml* vorgenommen. Diese basiert auf den bereits erwähnten Einschränkungen des Prototyps in Verbindung mit den *SpeedButtons*. Diese Konfiguration blendet folglich diejenigen Menüpunkte aus, die das Speichern, Öffnen oder Erstellen einzelner Units oder Formulare ermöglichen. In Verbindung mit der speziellen Auswahl der *SpeedButtons*, können somit nur noch Projekte geöffnet oder gespeichert werden. Der entsprechende Eintrag in der *Education.xml* lautet wie folgt:

Listing 6: Konfigurationseintrag der Menüs

```
1 <Menus>
2   <Hidden Item1="IDEMainMenu/File/itmFileNew/itmFileNewForm"
3     Item2="IDEMainMenu/File/itmFileNew/itmFileNewUnit"
4     Item3="IDEMainMenu/File/itmFileOpenSave/itmFileOpen"
5     Item4="IDEMainMenu/File/itmFileOpenSave/itmFileRecentOpen"
6     Item5="IDEMainMenu/File/itmFileOpenSave/itmFileRevert"
7     Item6="IDEMainMenu/File/itmFileOpenSave/itmFileSave"
8     Item7="IDEMainMenu/File/itmFileOpenSave/itmFileSaveAs"
9     Count="7" />
10 </Menus>
```

Die aufgelisteten Menüs werden jedoch nur in der minimalen sowie der erweiterten Konfiguration ausgeblendet.

Weiteres

Neben den bisherigen erwähnten Units/Dateien, findet auch eine Lokalisierung der Erweiterung statt. Diese wurde vom Autor zunächst speziell für die deutsche Sprache erstellt. Die Sprachauswahl hängt dabei zunächst von der in LAZARUS eingestellten Sprache ab. Optional kann jedoch zusätzlich bei der Installation des Packages eine andere Sprache ausgewählt werden (sofern weitere Sprachdateien erstellt wurden).

Der Prototyp bot darüber hinaus die Möglichkeit, einzelne Optionen (Editor, Umgebung) auf Standardwerte zurücksetzen zu können. Hintergrund dieser Modifikation war die Möglichkeit, die durch (neugierige) Schüler vorgenommenen Änderungen wieder rückgängig machen zu können. Dank der Sichtbarkeit von Menüs hat der Lehrer nunmehr die Möglichkeit, alle Menüs und

folglich auch die Einstellungen auszublenden. Das Verstellen kann den Schülern somit unmöglich gemacht werden. Das Hinzufügen weiterer Standardwerte ist mit der Erweiterung somit nicht mehr erforderlich.

4.3.3 Zukünftige Installation der Erweiterung

Wie bereits dargelegt, soll die fertige Erweiterung zukünftig als eigenes Paket angeboten und separat installiert werden können. Die Installation der Erweiterung erfolgt in wenigen Schritten:

Tabelle 21: Installation der Erweiterung

- **Schritt 1:** Nach Installation von LAZARUS befindet sich die (nicht installierte) Erweiterung im Verzeichnis *„Lazarus/components/education“*.
- **Schritt 2:** Nach dem Anwendungsstart kann über *Package -> Package-Datei öffnen* die bereits vorgestellte Package-Datei des Paketes geöffnet werden.
- **Schritt 3:** Über die Schaltfläche *Installieren* wird die Erweiterung zunächst kompiliert und anschließend installiert.
- **Schritt 4:** Nach erfolgreicher Installation startet die Entwicklungsumgebung neu und die Einstellungen können nun über die Umgebungseinstellungen vorgenommen werden.

4.3.4 Installation der mitgelieferten Version

Mit Abschluss der Arbeit wird die angepasste LAZARUS-Version bereitgestellt. Für eine einfache Installation wird erneut das in Anhang C erwähnte *NSIS*-Installationssystem verwendet. Die Installationsroutine unterscheidet sich hierbei nur geringfügig vom Prototyp. Auf die Unterschiede wird in der Folge kurz eingegangen.

Auswahl der Konfiguration

Der Prototyp stellte eine minimale LAZARUS-Version dar. Die Art der Einschränkungen konnte der Anwender nicht nachträglich ändern. Dies unterscheidet den Prototyp von der nun vorgelegten Version. Ab jetzt kann der Anwender umfangreiche Änderungen an der Oberfläche der IDE vornehmen. Dies kann bereits in einem ersten Schritt während der Installation erfolgen. Der Installer fragt neben den bekannten Schritten (z.B. Installationsverzeichnis) die bevorzugte Konfiguration der Bildungseinstellungen ab. Diese Stufen entsprechen den Einstellungen innerhalb der Entwicklungsumgebung. Gemäß der Wahl des Anwenders wird eine vorgefertigte *Education.xml* vorgemerkt und später in das LAZARUS-Konfigurationsverzeichnis kopiert. Nach dieser Auswahl beginnt der eigentliche Installationsprozess (das Kopieren der Dateien).

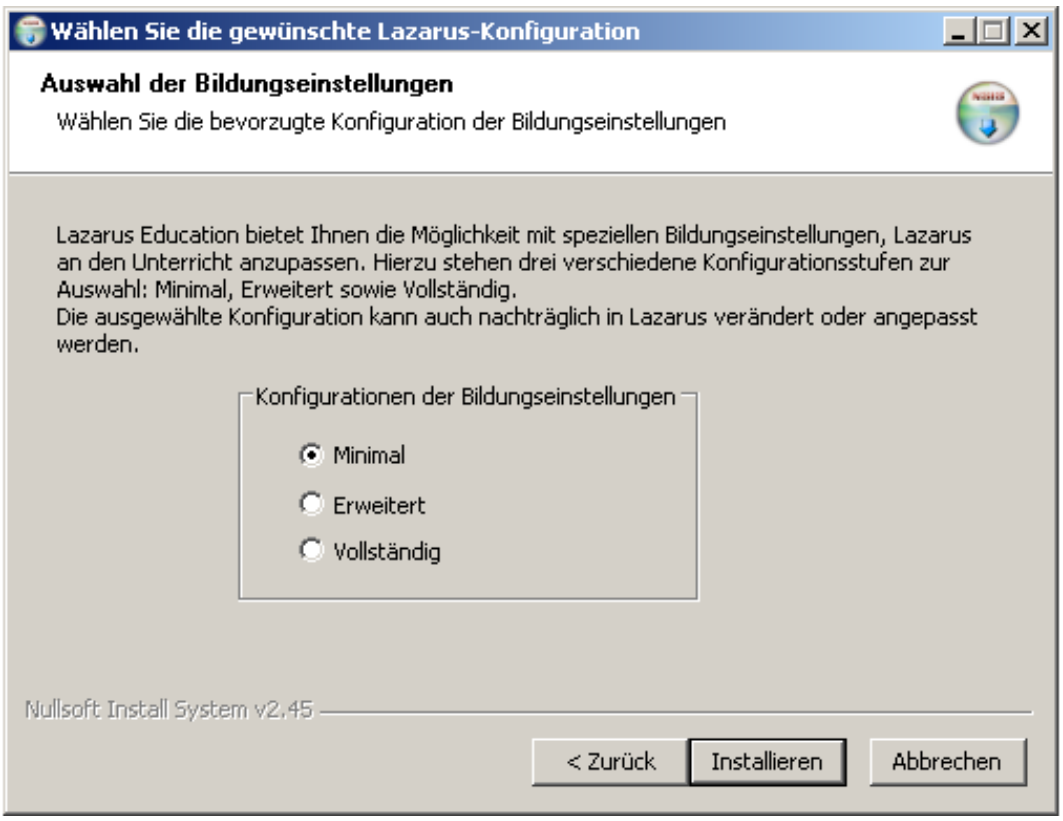

Abbildung 17: Auswahl der Konfiguration während der Installation

Zusätzliche Konfigurationsdateien

Neben den im Prototyp vorgestellten Konfigurationsdateien (*Environmentopions.xml*, *Editoroptions.xml*, *fpc.cfg*) benötigt die neu vorgelegte Version weitere vorbereitete Konfigurations-

dateien. Es handelt sich dabei zunächst um die bereits erwähnte *Education.xml*. In ihr werden die gewählten Bildungseinstellungen gespeichert. Die neue Version beinhaltet jedoch zwei zusätzlich installierte Erweiterungen. Dies ist einerseits das Education-Paket in seiner jetzigen Konfiguration. Andererseits wurde das Paket *SqlDB* installiert, da von einigen Lehrern Datenbank-Komponenten gewünscht wurden. Die Installation dieser zusätzlichen Packages wird in zwei Konfigurationsdateien vermerkt: *Packagefiles.xml* sowie *Miscellaneousoptions.xml*. Für einen reibungslosen Start der IDE müssen beide Konfigurationsdateien schon vor dem ersten Anwendungsstart vorhanden sein. In der *Packagefiles.xml* müssen darüber hinaus die Installationspfade für beide Packages automatisch korrigiert werden.

Dateizuordnungen

Lazarus arbeitet mit unterschiedlichen Dateiformaten. Um die Arbeit damit zu erleichtern, wird eine automatische Zuweisung der Dateiendungen mit Lazarus systemweit vorgenommen. So wurden Dateien mit der Endung **.pas*, **.pp*, **.lpi* sowie **.lpk* automatisch mit Abschluss der Installation mit Lazarus verknüpft. Wird also im Dateimanager der Wahl eine entsprechende Datei zum Öffnen ausgewählt, wird diese Datei in der Folge mit Lazarus geöffnet.

4.3.5 Fazit der Anpassungen

Wie im Abschnitt beschrieben, konnten vom Autor mehrere Möglichkeiten gefunden und realisiert werden, um Lazarus an den Schulunterricht anzupassen. Basis hierfür bieten drei Konfigurationsstufen, welche auf den Erfahrungen und Wünschen mehrerer Lehrer basieren. Somit erfolgt eine erste hilfreiche Vorkonfiguration der Entwicklungsumgebung. Das Ziel der Skalierbarkeit konnte in unterschiedlichen Ausprägungen erreicht werden: Die Komponentenpalette frei skalierbar, für Eigenschaften oder Ereignisse stehen separat Konfigurationsstufen zur Verfügung. Den Lehrern wird somit die Möglichkeit gegeben, die IDE einfach und bequem an ihre didaktischen Intentionen anzupassen. Dank der Zusammenarbeit mit den Entwicklern von Lazarus ist darüber hinaus die Nachhaltigkeit sowie der Fortbestand der Änderungen gewährleistet. Sämtliche Änderungen fließen in eine offizielle Erweiterung ein und stehen somit auch für künftige Lazarus-Versionen zur Verfügung. Ein langfristiger Einsatz in Schulen ist somit möglich.

4.4 Evaluation

Im Rahmen dieser Arbeit wurde bereits eine gezielte Befragung von Fachlehrern durchgeführt. Die Befragung verfolgte das Ziel, didaktische Anforderungen sowie benötigte Komponenten, Eigenschaften oder Ereignisse aus praxisnaher Sicht näher zu bestimmen und diese zu verifizieren. Letztendlich wurden in deren Ergebnis die vorgestellten Konfigurationsstufen entwickelt. Mit Abschluss der Arbeit war eine erneute Befragung der Fachlehrer zu den nun vorliegenden Ergebnissen vorgesehen. **So sollte mit dieser Evaluation ermittelt werden, ob die neue, frei skalierbare Lazarus-Version aus Sicht der Lehrer die gewünschte Unterstützung bietet und in ihren Unterricht einfließen kann.**

Der für diese Evaluation entwickelte Fragebogen (vgl. Anhang F) gliedert sich demzufolge in mehrere Teile. Neben einem **allgemeinen Teil** beinhaltet der Bogen zunächst vornehmlich **Fragen zu den neu geschaffenen Auswahlmöglichkeiten für Komponentenpalette sowie ObjektInspektor**. Darüber hinaus wird um eine **Einschätzung zu den jeweiligen Konfigurationsstufen** gebeten. Die Antworten sollen Aufschluss über die Qualität der Einteilung sowie die potenzielle Eignung für den Unterricht geben. Für die Bewertung kam wiederum eine Likert-Skala zum Einsatz. Die Antworten konnten im Bereich von 1 („Auf keinen Fall“) bis 4 („Auf jeden Fall“) gegeben werden.

Die Komponentenpalette ist nun frei skalierbar. Sie kann jedoch auch in Konfigurationsstufen verwendet werden. Hier erschien es dem Autor wichtig in Erfahrung zu bringen, wie die Lehrer diese freie Skalierbarkeit bewerten und ob sie diese dann überhaupt in ihrem Unterricht verwenden würden. Änderungen im ObjektInspektor sind derzeit noch nicht in dem Umfang möglich, wie sie die Komponentenpalette nun bietet. Es schien aber auch hier wichtig zu erfahren, ob die Einteilung den Erwartungen der Lehrer entspricht. Ferner wurden die Lehrkräfte befragt, ob sie die Trennung von Eigenschaften und Ereignissen ebenfalls als sinnvoll erachten. Dem Autor schien diese Unterteilung deshalb sinnvoll, weil sie den Lehrern seines Erachtens nun mehr Auswahlmöglichkeiten bietet. Insofern ist hier eine Einschätzung der Fachlehrer erwünscht.

Der letzte Teil der Befragung zielte auf den potenziellen Schuleinsatz ab. Diese Fragen stellen also gleichermaßen eine Zusammenfassung und das Ergebnis der Arbeit dar: **Ist**

es gelungen, den Lehrern mit dieser angepassten IDE eine Version zur Verfügung zu stellen, die sie dann auch Dank der Möglichkeiten zur individuellen Anpassung im Unterricht verwenden würden? In der ersten durchgeführten Evaluation wurden im Unterricht identifizierte Probleme abgefragt. So z.B. die Suche nach Komponenten, Eigenschaften oder nach prinzipiellen Problemen der Schüler bei der Programmierung. Diese Frage wurde nun erneut gestellt. Ihre Auswertung wird Schlussfolgerungen zulassen, ob mit der neuen frei skalierbaren Version eine signifikante Besserung hinsichtlich der o. g. Probleme erreicht werden konnte. Der Fragebogen endet mit der Möglichkeit, gegebenenfalls weite Änderungswünsche oder Anregungen für weitere Entwicklungsarbeiten zu geben.

Leider nahmen bisher jedoch nur wenige Lehrer an der zweiten Evaluation teil. Das ist insbesondere auf den zeitlichen Rahmen der Evaluation zurückzuführen. Sie begann kurz vor Beginn der Winterferien 2010 in Sachsen und erstreckte sich auch über diesen Zeitraum. Zwangsläufig hielt sich somit auch die Anzahl zurückgeführter Fragebögen in Grenzen. Eine umfangreiche Auswertung war somit nicht mehr möglich.

Der Autor verfolgte auch das Ziel, die Ergebnisse der zweiten Evaluation den Anforderungen bzw. Wünschen der Lehrer aus der vorhergehenden Evaluation gegenüberzustellen. Hieraus sollte abgeleitet werden, ob die Anregungen der Lehrer positiv umgesetzt und von LAZARUS nun erfüllt werden können. Vor allem der vorgesehene Vergleich der bei den Schülern im Unterricht beobachteten Probleme mit alter und neuer Version der Entwicklungsumgebung kann derzeit somit nicht realisiert werden. Dennoch lässt sich aus den wenigen, bisher vorliegenden Fragebögen sowie auch aus Lehrergesprächen im Rahmen einer Präsentation bereits jetzt eine **positive Tendenz** ableiten. Die Antworten der Lehrer zeigen, dass vor allem die **Möglichkeit zur Anpassung der Komponentenpalette sowie der Anzeige im ObjektInspektor von ihnen sehr positiv bewertet wird**. Die **freie Skalierbarkeit der Komponentenpalette** und die damit verbundene optimale Anpassungsmöglichkeit an den Unterricht fallen hierbei **besonders positiv** auf. Ein ähnliches Bild zeichnet sich auch nach zahlreichen Gespräche mit Lehrern im Rahmen einer Präsentation der neuen Entwicklungsumgebung ab. Auch hier wurde die vorgestellte Vorabversion in der Tendenz positiv bewertet. Ausgehend von diesen ersten positiven

Erkenntnissen erwartet der Autor auch bei Vorliegen einer größeren Datenmenge tendenziell positive Rückmeldungen. Dies wird insbesondere bei den Lehrern erwartet, die in ihrem Unterricht aktuell DELPHI verwenden. Eher skeptisch ist der Autor jedoch derzeit hinsichtlich des zukünftigen Einsatzes von LAZARUS bei den Lehrern, die bisher andere IDEs, wie beispielsweise ECLIPSE nutzen. Zeigte sich doch bereits in der ersten Evaluation, dass die Bereitschaft der Lehrer zum Wechsel auf eine andere IDE gegenwärtig eher gering ist.

Der Autor ist bemüht, z.B. im Rahmen eines Workshops, weitere Probanden für die neue Evaluation zu gewinnen, um somit die Basis für fundierte wissenschaftliche Aussagen weiter zu vertiefen. Diese Ergebnisse werden gesondert nachgereicht.

5 Fazit der Arbeit

Das Ziel der Arbeit bestand vorrangig in der Entwicklung einer frei skalierbaren Version der Entwicklungsumgebung Lazarus. Diese soll es dem versierten Lehrer künftig ermöglichen, in seinem Unterricht auf eine, seinen didaktischen Intentionen und der Leistungsfähigkeit seiner Schüler angepassten, Lazarus-Version zurückgreifen zu können.

Hierzu wurde zunächst eine theoretische Analyse der Entwicklungsumgebung Eclipse vorgenommen. Neben der Beschreibung der zu Grunde liegenden Plattform, erfolgte eine Untersuchung zur Gebrauchstauglichkeit der IDE. Diese Untersuchung zeigte, dass besonders die Grundsätze der *Lernförderlichkeit* sowie *Steuerbarkeit* vor dem Hintergrund eines schulischen Einsatzes funktionell gut umgesetzt wurden. Im weiteren Verlauf wurden Lösungsansätze aufgezeigt, wie potenzielle eigene Erweiterungen für mehrere Programmiersprachen auch nach der Aktualisierung der Gesamtumgebung weiter verwendet werden können.

Eine vom Autor durchgeführte Evaluation ergab, dass an sächsischen Gymnasien eine große Anzahl unterschiedlicher Entwicklungsumgebungen zum Einsatz kommen. Die überwiegende Mehrzahl der Lehrer greift jedoch auf die IDE Delphi zurück. Diese Tatsache zeigt, dass die Fachlehrer und Schulen häufig mit dem Problem der Wahl der richtigen IDE konfrontiert werden. Im Kapitel „Didaktische Anforderungen“ stellt der Autor zahlreiche Anforderungen vor, die aus seiner Sicht an eine Entwicklungsumgebung für den schulischen Einsatz zu stellen sind. Die Grundlage für diese Forderungsaufstellung bildete neben einer theoretischen Untersuchung die gezielte Befragung von Lehrkräften durch den Autor.

In Umsetzung dieser Befragung konnten zunächst drei Konfigurationsstufen für die zukünftige Lazarus-Version definiert werden. Diese bilden die Basis für die skalierbare Version der IDE. Die vom Autor entwickelten Änderungen fließen dabei in eine offizielle Erweiterung (der Entwicklungsumgebung Lazarus) ein. Somit ist auch die Nachhaltigkeit der Entwicklung gewährleistet. Auch wenn eine weitere Evaluation derzeit nur eine nicht zufrieden stellende Anzahl an Rückmeldungen erbrachte, kann aus den bisher gewonnenen Erkenntnissen durchaus eine positive Tendenz abgeleitet werden. Den Lehrern steht nunmehr eine Entwicklungsumgebung zur Verfügung, die noch besser und individueller an die didaktischen Erfordernisse des Unterrichts

angepasst werden kann. Ferner kann sie, Dank der freien Verfügbarkeit, im Unterricht sowie zur Heimarbeit verwendet werden. Sie stellt somit eine gute Alternative zur derzeit noch häufig verwendeten DELPHI-IDE dar.

Trotz des insgesamt positiven Resümees sieht der Autor Potenzial für weiterführende Arbeiten. So ist LAZARUS primär auf die Arbeit mit der Sprache *(Object-)Pascal* ausgerichtet. Es kann jedoch auch für andere Sprachen verwendet werden. Die entsprechende Syntaxhervorhebung ist bereits integriert. Die Komponentenpalette sowie der visuelle Editor sind bisher jedoch nicht für andere Sprachen geeignet. An dieser Stelle könnten künftige Arbeiten ansetzen und eventuelle Möglichkeiten zur Konvertierung prüfen. Der praktische Einsatz einer IDE mit mehreren Programmiersprachen könnte nach Auffassung des Autors zu einer weiteren Verbesserung der Informatikausbildung führen. Darüber hinaus sollte auch der ECLIPSE-Ansatz in weiterführenden Arbeiten verfolgt werden. Die theoretische Machbarkeit einer *Pascal*-Erweiterung wurde in dieser Arbeit aufgezeigt. Folglich liegt in der Entwicklung einer Erweiterung für sächsische Schulen ebenso erhebliches Potenzial für weitere Arbeiten.

A Anhang: Beispiel einer *plugin.xml*

Listing 7: plugin.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <plugin
3  id="pascaleditor"
4  name="Pascal-SourceditorPlug-In"
5  Version="0.0.1"
6  Class="pascaleditor.PascalEditorPlugin">
7  <runtime>
8   <library name="pascaleditor.jar"/>
9  </runtime>
10 <requires>
11  <import plugin="org.eclipse.ui"/>
12 </requires>
13 <extension
14  name="PascalSourceeditor"
15  point="org.eclipse.ui.editors">
16  <editor
17    name="PascalSourceeditor"
18    default="true"
19    icon="icons/pascal.gif"
20    extensions="pas,pascal,pp"
21    contributorClass="pascaleditor.editors.PascalEditorActionContributor"
22    class="pascaleditor.editors.PascalEditor"
23    id="pascaleditor.editors.PascalEditor">
24  </editor>
25 </extension>
26 </plugin>
```

B Anhang: Die *educationlaz.lpk* der Erweiterung

Listing 8: Die *educationlaz.lpk* der Erweiterung

```
1 <?xml version="1.0"?>
2 <CONFIG>
3   <Package Version="3">
4     <Name Value="EducationLaz"/>
5     <Author Value="Mattias Gaertner, Michael Kuhardt"/>
6     <CompilerOptions>
7       <Version Value="8"/>
8       <SearchPaths>
9         <UnitOutputDirectory Value="lib/$(TargetCPU)-$(TargetOS)"/>
10      </SearchPaths>
11      <Other>
12        <CompilerPath Value="$(CompPath)"/>
13      </Other>
14    </CompilerOptions>
15    <Description Value="IDE package for training, courses and education."/>
16    <License Value="LGPL-2 or any later"/>
17    <Version Major="1" Release="1"/>
18    <Files Count="7">
19      <Item1>
20        <Filename Value="eduenvoptsframe.pas"/>
21        <HasRegisterProc Value="True"/>
22        <UnitName Value="EduEnvOptsFrame"/>
23      </Item1>
24      <Item2>
25        <Filename Value="README.txt"/>
26        <Type Value="Text"/>
27      </Item2>
28      <Item3>
29        <Filename Value="eduoptions.pas"/>
30        <UnitName Value="EduOptions"/>
31      </Item3>
32      <Item4>
33        <Filename Value="edupkgsystem.pas"/>
34        <HasRegisterProc Value="True"/>
35        <UnitName Value="EduPkgSystem"/>
36      </Item4>
37      <Item5>
38        <Filename Value="educomppalette.pas"/>
```

```
39          <HasRegisterProc Value="True"/>
40          <UnitName Value="EduCompPalette"/>
41        </Item5>
42        <Item6>
43          <Filename Value="edumenu.pas"/>
44          <HasRegisterProc Value="True"/>
45          <UnitName Value="EduMenu"/>
46        </Item6>
47        <Item7>
48          <Filename Value="edunewprogram.pas"/>
49          <HasRegisterProc Value="True"/>
50          <UnitName Value="EduNewProgram"/>
51        </Item7>
52      </Files>
53      <i18n>
54        <OutDir Value="languages"/>
55      </i18n>
56      <Type Value="RunAndDesignTime"/>
57      <RequiredPkgs Count="5">
58        <Item1>
59          <PackageName Value="CodeTools"/>
60        </Item1>
61        <Item2>
62          <PackageName Value="SynEdit"/>
63        </Item2>
64        <Item3>
65          <PackageName Value="LCL"/>
66          <MinVersion Major="1" Valid="True"/>
67        </Item3>
68        <Item4>
69          <PackageName Value="IDEIntf"/>
70        </Item4>
71        <Item5>
72          <PackageName Value="FCL"/>
73          <MinVersion Major="1" Valid="True"/>
74        </Item5>
75      </RequiredPkgs>
76      <UsageOptions>
77        <UnitPath Value="$(PkgOutDir)/"/>
78      </UsageOptions>
79      <PublishOptions>
80        <Version Value="2"/>
```

```
81        <IgnoreBinaries Value="False"/>
82      </PublishOptions>
83    </Package>
84 </CONFIG>
```

C Anhang: Installationsroutine des Prototyps

Für den Einsatz eines Installers wählte der Autor das *Nullsoft Scriptable Install System* (NSIS) der Firma NULLSOFT aus. Diese *Open-Source*-Anwendung ermöglicht es, sowohl unter *Windows* als auch unter *Linux* Installer für *Windows*-Anwendungen zu erstellen. Bei der Erstellung dieser Installer kommt eine eigene Skriptsprache zur Anwendung. Sie ermöglicht die Anpassung eines Installers an die spezifischen Anforderungen der jeweiligen Anwendung. Weiterhin stehen die integrierten Kompressionsmethoden *ZLib*[4], *BZip2*[5] sowie *LZMA*[6] zur Verfügung. Für den Prototyp wurde LZMA verwendet. Dieser bietet nach Angaben des Anbieters die bestmöglichen Ergebnisse. Auf das eigentliche Skript zum Installer wird an dieser Stelle nicht näher eingegangen. Auf Grund des erheblichen Skriptumfanges wurde es auf der beiliegenden DVD abgelegt. Stattdessen soll der Installationsprozess und seine Vorgänge kurz dargelegt werden. Der Installationsprozess läuft im Wesentlichen in drei Schritten ab:

Schritt 1: Sprachauswahl

Nach Start der Installationsroutine fordert die Anwendung den Nutzer zunächst zur Sprachauswahl auf. Das NSIS-System unterstützt eine Vielzahl von Sprachen und bietet diese an. Die Wahl der im Installer unterstützten Sprachen unterliegt dabei dem Skriptautor. Im vorliegenden Fall nahm der Autor eine Reduzierung des Sprachumfanges auf die Sprachen Deutsch sowie Englisch vor. Abbildung 18 zeigt die Sprachauswahl.

Abbildung 18: Die Sprachauswahl

[4]Eine freie verfügbare Programmbibliothek. Sie dient zur De-/Kompression mit Hilfe des Deflate-Algorithmus.
[5]Ein frei verfügbares Komprimierungsprogramm zur verlustfreien Datenkompression.
[6]Ein freier Datenkompressionsalgorithmus von Igor Pavlov (Lempel-Ziv-Markow-Algorithmus).

Schritt 2: Wahl des Installationspfades

In der bisherigen Version des Prototyps war das manuelle Kopieren der Dateien und in der Folge auch das Bearbeiten von Pfaden durch den Nutzer erforderlich. Mit Hilfe des neuen Installers ist das nicht mehr nötig. Die Installationsroutine fragt dazu zunächst den gewünschten Installations-

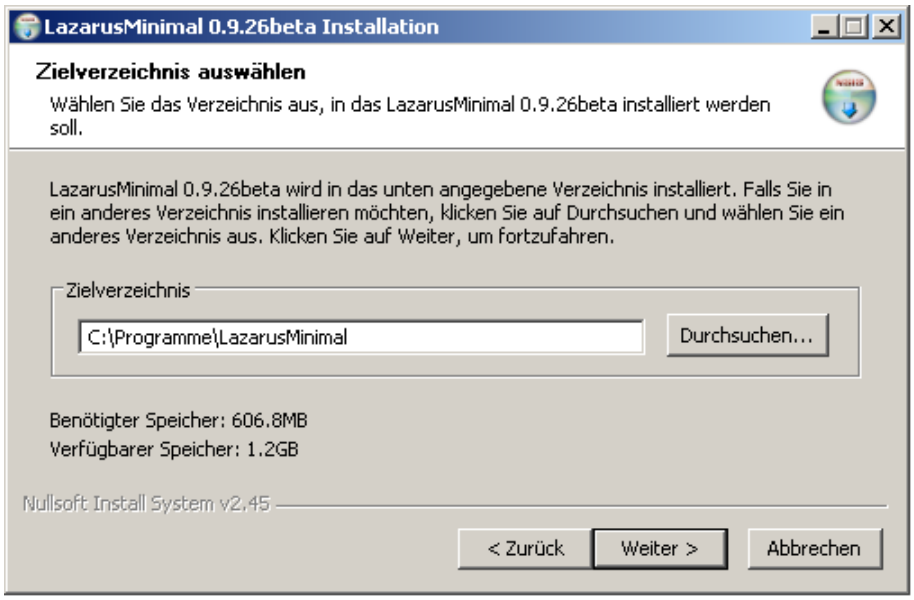

Abbildung 19: Wahl des Installationspfades

pfad ab. Als Vorgabe schlägt der Installer den Pfad *C:/Programme/LazarusMinimal/* vor. Dem Nutzer steht jedoch frei, diesen Pfad (auch mit Hilfe des Dateisystems) beliebig zu verändern. Als zusätzlich Hilfe zeigt der Installer dabei den verfügbaren sowie von LAZARUS benötigten Speicherplatz auf der gewählten Festplatte an. Nachdem der Benutzer das Verzeichnis seiner Wahl mit dem „Weiter“-Button bestätigt hat, beginnt der eigentliche Installationsprozess. Um dem Nutzer eine Rückmeldung über den aktuellen Verlauf der Installation zu geben, werden die zu entpackenden und kopierenden Dateien aufgelistet. Der Fortschritt des Installationsprozesses wird über einen Fortschrittbalken angezeigt (vgl. Abbildung 20). Währenddessen finden im Hintergrund verschiedene Operationen statt. Zunächst erfolgt das Entpacken der komprimierten Dateien. Diese wurden, wie bereits dargestellt, mit LZMA komprimiert. In der Praxis führte das zu einer deutlichen Reduzierung der Dateigröße des Installationspakets. Nach dem Kopieren erfolgt gleichfalls das automatische Bearbeiten der erforderlichen Dateien. Der Vorgang, der dem Nutzer im selben Fenster angezeigt wird, findet jedoch nicht in einem gesonderten Schritt

statt. Involviert sind hierbei die drei Dateien *Environmentoptions.xml*, *Editoroptions.xml* sowie *fpc.cfg*. Alle Dateien enthalten von LAZARUS die benötigten Pfadangaben. Durch die automa-

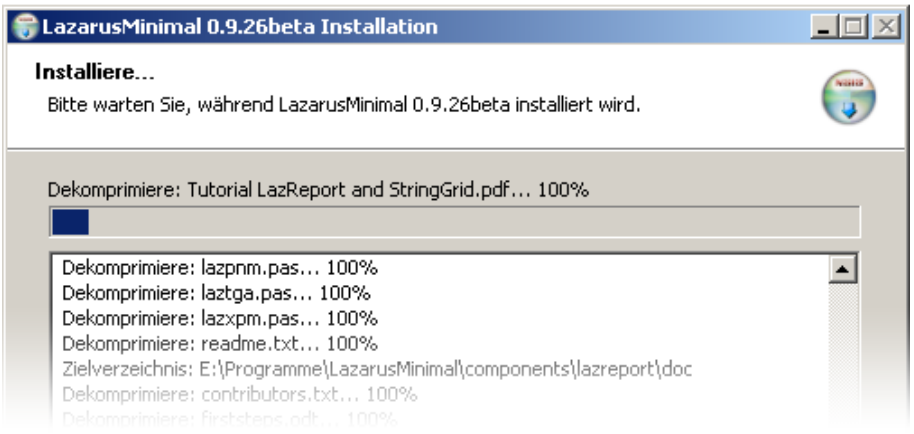

Abbildung 20: Der Installationsprozess

tische Bearbeitung mit dem gewählten Installationspfad müssen diese nicht mehr wie bisher nach der Installation manuell eingepflegt werden. Für die *XML*-Dateien erfolgte das bisher über die LAZARUS-IDE. Wesentlich umständlicher war die Ausführung dieses Prozesses für die vom *FreePascal*-Compiler benötigte Konfigurationsdatei *fpc.cfg*. Diese musste der Nutzer bisher mit Hilfe eines Texteditors manuell verändern. Auch dieser Schritt wird nun vom Installer ausgeführt.

Schritt 3: Abschluss der Installation

Nach erfolgreicher Installation wird der Installationsprozess über die Schaltfläche „Fertig stellen“ abgeschlossen. Optional kann von hier aus der Start der Anwendung bereits automatisch erfolgen.

Abschließend sei angemerkt, dass es bei einer beabsichtigten parallelen Nutzung mehrerer LAZARUS-Versionen jedoch auch weiterhin einer gewissen Zuarbeit des Nutzers bedarf. Sämtliche Versionen legen (unter *Windows XP*) im Verzeichnis *C:/Dokumente und Einstellungen/<Nutzername>/Lokale Einstellungen/Anwendungsdaten/Lazarus* wichtige Einstellungsdateien sowie die vorgestellten *XML*-Dateien mit Pfadangaben ab. Um die jeweils korrekten Dateien für die gewünschte Version verwenden zu können, müssen diese zuvor gesichert und dann in das o. g. Verzeichnis kopiert werden.

D Anhang: Exemplarische Funktionen

Listing 9: Sichtbarkeit der Komponenten (vollständige Funktion)

```
1  procedure TEduCompPaletteFrame.ShowSelected(extended: boolean);
2 var
3   Node: TTreeNode;
4   CompName: String;
5   MinimalComponents: array[0..12] of String;
6   ExtendedComponents: array[0..25] of String;
7   i,k: integer;
8 begin
9   MinimalComponents[0] :=  'TEdit';
10  MinimalComponents[1] :=  'TButton';
11  MinimalComponents[2] :=  'TCheckBox';
12  MinimalComponents[3] :=  'TLabel';
13  MinimalComponents[4] :=  'TListBox';
14  MinimalComponents[5] :=  'TComboBox';
15  MinimalComponents[6] :=  'TRadioGroup';
16  MinimalComponents[7] :=  'TRadioButton';
17  MinimalComponents[8] :=  'TPanel';
18  MinimalComponents[9] :=  'TMainMenu';
19  MinimalComponents[10] :=  'TMemo';
20  MinimalComponents[11] :=  'TGroupBox';
21  MinimalComponents[12] :=  'TImage';
22  ExtendedComponents[0] :=  'TBitBtn';
23  ExtendedComponents[1] :=  'TScrollBar';
24  ExtendedComponents[2] :=  'TPopupMenu';
25  ExtendedComponents[3] :=  'TCheckGroup';
26  ExtendedComponents[4] :=  'TActionList';
27  ExtendedComponents[5] :=  'TStringGrid';
28  ExtendedComponents[6] :=  'TSpeedButton';
29  ExtendedComponents[7] :=  'TTimer';
30  ExtendedComponents[8] :=  'TIdleTimer';
31  ExtendedComponents[9] :=  'TPageControl';
32  ExtendedComponents[10] :=  'TStaticText';
33  ExtendedComponents[11] :=  'TDBGrid';
34  ExtendedComponents[12] :=  'TOpenDialog';
35  ExtendedComponents[13] :=  'TSaveDialog';
36  ExtendedComponents[14] :=  'TSelectDirectoryDialog';
37  ExtendedComponents[15] :=  'TDataSource';
38  ExtendedComponents[16] :=  'TDBNavigator';
```

```
39   ExtendedComponents[17] :=  'TDBText';
40   ExtendedComponents[18] :=  'TDBEdit';
41   ExtendedComponents[19] :=  'TDBMemo';
42   ExtendedComponents[20] :=  'TDBImage';
43   ExtendedComponents[21] :=  'TDBListBox';
44   ExtendedComponents[22] :=  'TDBLookupListBox';
45   ExtendedComponents[23] :=  'TDBComboBox';
46   ExtendedComponents[24] :=  'TDBLookupComboBox';
47   ExtendedComponents[25] :=  'TDBCheckBox';
48
49   ComponentsTreeView.BeginUpdate;
50   Node:=ComponentsTreeView.Items.GetFirstNode;
51   while Node<>nil do begin
52     if Node.Parent<>nil then begin
53       CompName:=Node.Text;
54       for i := 0 to 12 do begin
55         if (CompareText (CompName , MinimalComponents[i] )=0) then begin
56             EduComponentPaletteOptions.ComponentVisible[CompName]:=true;
57             Node.StateIndex:=ShowImgID;
58         end;
59       end;
60       if extended then begin
61         for k := 0 to 25 do begin
62           if (CompareText (CompName , ExtendedComponents[k] )=0) then begin
63             EduComponentPaletteOptions.ComponentVisible[CompName]:=true;
64             Node.StateIndex:=ShowImgID;
65           end;
66         end;
67       end;
68     end;
69     Node:=Node.GetNext;
70   end;
71   ComponentsTreeView.EndUpdate;
72 end;
```

Listing 10: Auslesen der Konfiguration von Eigenschaften/ Ereignissen)

```
1 procedure TPropInfoList.getPropsAndEventsConfiguration();
2 var
3    Doc: TXMLDocument;
4    Node: TDOMNode;
5    test: boolean;
6    ConfFileName: string;
```

```
7  begin
8    ConfFileName:=GetAppConfigDir(false)+'education.xml';
9    if (not FileExistsUTF8(ConfFileName)) then begin
10     DebugLn('Note: config file not');
11     FVersionProps:='empty';
12   end else begin
13     ReadXMLFile(Doc, ConfFileName);
14     if not(Doc.FirstChild.HasAttributes) then begin
15     FVersionProps:='empty';
16   end else begin
17    test:=false;
18    Node:=Doc.DocumentElement.FindNode('PropsEvents');
19    while  (test=false) do begin
20      if (CompareText(Node.Attributes.GetNamedItem('PropsMinimal').NodeValue,
21       'True')=0) then begin
22         FVersionProps:='min';
23         test:=true;
24      end else if (CompareText(Node.Attributes.GetNamedItem('PropsExt').NodeValue,
25       'True')=0) then begin
26         FVersionProps:='ext';
27         test:=true;
28      end else if (CompareText(Node.Attributes.GetNamedItem('PropsFull').NodeValue,
29       'True')=0) then begin
30         FVersionProps:='full';
31         test:=true;
32      end;
33    end;
34    test:=false;
35    while   test=false  do begin
36      if (CompareText(Node.Attributes.GetNamedItem('EventsMinimal').NodeValue,
37       'True')=0) then begin
38         FVersionEvents:='min';
39         test:=true;
40      end else if (CompareText(Node.Attributes.GetNamedItem('EventsExt').NodeValue,
41       'True')=0) then begin
42         FVersionEvents:='ext';
43         test:=true;
44      end else if (CompareText(Node.Attributes.GetNamedItem('EventsFull').NodeValue,
45       'True')=0) then begin
46         FVersionEvents:='full';
47         test:=true;
48      end;
```

```
49     end;
50    end;
51    Doc.Free;
52   end;
53 end;
```

Listing 11: Auslesen der Konfiguration der Registerkarten

```
1  procedure TObjectInspectorDlg.getOIPageConfig();
2  var
3      Doc: TXMLDocument;
4      Node: TDOMNode;
5      ConfFileName: string;
6  begin
7    ConfFileName:=GetAppConfigDir(false)+'education.xml';
8    if (not FileExistsUTF8(ConfFileName)) then begin
9      DebugLn('Note: config file not');
10     FOIPageActive:=false;
11   end else begin
12     ReadXMLFile(Doc, ConfFileName);
13     if not(Doc.FirstChild.HasAttributes) then begin
14       FOIPageActive:=false;
15     end else begin
16      FOIPageActive:=true;
17      Node:=Doc.DocumentElement.FindNode('OIPages');
18      if (CompareText(Node.Attributes.GetNamedItem('OIPageFavs').NodeValue, 'True')=0)
19      then begin
20        FShowFavorites:=true;
21      end else begin
22        FShowFavorites:=false;
23      end;
24      if (CompareText(Node.Attributes.GetNamedItem('OIPageRestricted').NodeValue, 'True')=0)
25      then begin
26          FShowRestricted:=true;
27      end else begin
28          FShowRestricted:=false;
29      end;
30     end;
31   end;
32 end;
```

E Anhang: Befragung zum Einsatz von Entwicklungsumgebungen in Schulen

A Allgemeine Fragen

Welche Sekundarstufen betreuen Sie im Informatikunterricht?

☐ Sekundarstufe I ☐ Sekundarstufe II

Welche Profile betreuen Sie?

Welche Programmiersprachen werden an Ihrer Schule gegenwärtig vermittelt?

Welche Anforderungen stellen Sie an eine, den speziellen Belangen einer Schule angepassten Entwicklungsumgebung?

☐ Einfachheit
☐ Selbsterklärende Elemente und Symbole
☐ Eingeschränkter Sprachumfang
☐ Verringerte Komplexität
☐ Grafische Visualisierung des Systemzustandes
☐ Unterstützung mehrerer Programmiersprachen
☐ Individuelles Anpassen der Benutzerschnittstelle
☐ Robustheit
☐ Fehlertoleranz
☐ Angepasste Terminologie
☐ Syntaxhervorhebung
☐ Quelltextvorlagen
☐ Quelltextgenerierung

☐ Weitere:

B Derzeit Verwendete Entwicklungsumgebung

Welche Entwicklungsumgebung wird an Ihrer Schule gegenwärtig verwenden?

☐ Delphi ☐ Lazarus ☐ Weitere:

Welche Aspekte - insbesondere aus didaktischer Sicht - führten zur Entscheidung für den Einsatz der derzeit verwendeten IDE?

Welche Aspekte sprachen aus didaktischer Sicht gegen einen Einsatz der derzeit verwendeten IDE?

(Bitte nur eine Nennung pro Zeile)	Auf keinen Fall				Auf jeden Fall
	1	2	3	4	5
Würden Sie eine angepasste Lazarus-Version Ihrer derzeit verwendeten IDE vorziehen?	☐	☐	☐	☐	☐
Würden Sie via Einstellungen Änderungen (Komponenten, Eigenschaften, ...) vornehmen, um die IDE besser an den Unterricht anpassen zu können?	☐	☐	☐	☐	☐
Würden Sie Änderungen im Quellcode vornehmen, um die IDE besser an den Unterricht anzupassen?	☐	☐	☐	☐	☐
Würden Sie eine gut vorkonfigurierte IDE-Version (Minimal, Erweitert, Vollständig) einer kompletten Version mit der Möglichkeit zur Änderung vorziehen?	☐	☐	☐	☐	☐
Ziehen Sie die langfristige Arbeit mit einer stabilen Version einer Entwicklungsumgebung, der einer Variante mit regelmäßigen Aktualisierungen vor?	☐	☐	☐	☐	☐

Welche Probleme im Umgang mit der Entwicklungsumgebung Delphi oder Lazarus beobachten Sie bei Ihren Schülern am Häufigsten?

- ☐ Verwirrung über zu viele Funktionen
- ☐ Suche nach den benötigten Komponenten
- ☐ Suche nach den benötigten Eigenschaften oder Ereignissen
- ☐ Ausprobieren aller möglichen Komponenten oder Eigenschaften
- ☐ Mangelnde Fokussierung auf die Aufgabe
- ☐ Prinzipielle Probleme im Umgang mit der Entwicklungsumgebung
- ☐ Fehlendes Verständnis für die Programmierung

☐ Weitere:

C Komponentenpalette

Welche Standard-Komponenten benötigen Sie mindestens für den Unterricht?

☐ TMainMenu	☐ TPopupMenu	☐ TButton
☐ TLabel	☐ TEdit	☐ TMemo
☐ TToggleBox	☐ TCheckBox	☐ TRadioButton
☐ TListBox	☐ TComboBox	☐ TScrollBar
☐ TGroupBox	☐ TRadioGroup	☐ TCheckGroup
☐ TPanel	☐ TActionListener	

Welche Standard-Komponenten erachten Sie darüber hinaus als sinnvoll?

- ☐ TBitBtn
- ☐ TSpeedButton
- ☐ TStaticText
- ☐ TImage
- ☐ Weitere (exakter Name nicht notwendig):

D Objektinspektor

Welche Eigenschaften von Komponenten benötigen Sie mindestens für den Unterricht (exakter Name nicht notwendig)?

Welche erachten Sie darüber hinaus als sinnvoll?

Welche Ereignisse benötigen Sie mindestens für den Unterricht?

Welche sehen Sie darüber hinaus als sinnvoll an?

Sollte eine minimale Version weitere Ereignisse enthalten (derzeit wird nur das OnClick-Ereignis angeboten)?

- ☐ Ja
- ☐ Nein
- ☐ Wenn ja, welche:

Welche weiteren Änderungen möchten Sie darüber hinaus als wünschenswert anregen?

Vielen Dank für Ihre Teilnahme!

F Anhang: Befragung zum Einsatz einer skalierbaren Lazarus-Version

Befragung zum Einsatz einer skalierbaren Lazarus-Version

A Allgemeine Fragen

Welche Sekundarstufen betreuen Sie im Informatikunterricht?

☐ Sekundarstufe I ☐ Sekundarstufe II

Welche Profile betreuen Sie?

Welche Entwicklungsumgebung wird an Ihrer Schule gegenwärtig verwenden?

☐ Delphi ☐ Lazarus ☐ Weitere:

B Die Änderungen

(Bitte nur eine Nennung pro Zeile)	Auf keinen Fall	Eher nein	Eher ja	Auf jeden Fall
	1	2	3	4
Ich bewerte es positiv, Änderungen an Lazarus vornehmen zu können.	☐	☐	☐	☐
Ich finde es gut, bereits während der Installation zwischen verschiedenen Konfigurationen wählen zu können.	☐	☐	☐	☐
Die Gliederung in 3 Konfigurationsstufen ist zweckmäßig.	☐	☐	☐	☐
Ich betrachte es als positiv, auf Basis der Konfigurationsstufen weitere Anpassungen vornehmen zu können.	☐	☐	☐	☐

Komponentenpalette

(Bitte nur eine Nennung pro Zeile)	Auf keinen Fall	Eher nein	Eher ja	Auf jeden Fall
	1	2	3	4
Die Gliederung der Komponenten in die 3 Konfigurationsstufen ist zweckmäßig.	☐	☐	☐	☐
Die freie Skalierbarkeit der Komponentenpalette hilft mir, diese optimal an den Unterricht anzupassen.	☐	☐	☐	☐
Ich werde die Möglichkeit einer individuellen Komponentenauswahl für den Unterricht nutzen.	☐	☐	☐	☐

ObjektInspektor

(Bitte nur eine Nennung pro Zeile)	Auf keinen Fall	Eher nein	Eher ja	Auf jeden Fall
	1	2	3	4
Die Aufteilung der Eigenschaften auf die drei Konfigurationsstufen finde ich zweckmäßig.	☐	☐	☐	☐
Die Aufteilung der Ereignisse auf die drei Konfigurationsstufen finde ich zweckmäßig.	☐	☐	☐	☐
Ich sehe es als positiv an, Eigenschaften und Ereignisse getrennt von einander auswählen zu können.	☐	☐	☐	☐
Die Möglichkeit des Ausblendens einzelner Registerkarten im ObjektInspektor scheint mir angemessen.	☐	☐	☐	☐

Weiteres

(Bitte nur eine Nennung pro Zeile)	Auf keinen Fall	Eher nein	Eher ja	Auf jeden Fall
	1	2	3	4
Das Ausblenden von Menüs und Menüpunkten ist hilfreich.	☐	☐	☐	☐
Das Ausblenden von SpeedButtons ist nützlich.	☐	☐	☐	☐
Der Umfang der Änderungen ist für den Schulunterricht optimal.	☐	☐	☐	☐
Die Änderungen helfen mir, Lazarus bestmöglich an den Unterricht anzupassen.	☐	☐	☐	☐
Ich kann mir vorstellen, die angepasste Lazarus-Version künftig im Unterricht einzusetzen.	☐	☐	☐	☐

Welche Probleme im Umgang mit der angepassten Lazarus-Version vermuten Sie noch immer?

- ☐ Verwirrung über zu viele Funktionen
- ☐ Suche nach den benötigten Komponenten
- ☐ Suche nach den benötigten Eigenschaften oder Ereignissen
- ☐ Ausprobieren aller möglichen Komponenten oder Eigenschaften
- ☐ Mangelnde Fokussierung auf die Aufgabe
- ☐ Prinzipielle Probleme im Umgang mit der Entwicklungsumgebung
- ☐ Fehlendes Verständnis für die Programmierung
- ☐ Weitere:

Wünschen Sie sich weitere Anpassungsmöglichkeiten?

☐ Nein ☐ Wenn ja, welche:

Vielen Dank für Ihre Teilnahme!

G Anhang: DVD LazarusEducation

Die beiliegende DVD enthält LazarusEducation sowie den Prototyp in einer installierbaren Version. Darüber hinaus den Quellcode sowie diese Arbeit in elektronischer Form.

Literaturverzeichnis

Balzert, H. (2008): *Lehrbuch der Softwaretechnik: Softwaremanagement*, 2. Aufl., Spektrum Akademischer Verlag, Heidelberg, Deutschland.

Baumann, R. (1990): *Didaktik der Informatik*, 1. Aufl., Klett-Schulbuchverlag, Stuttgart, Deutschland.

Boehme, P. (1996): Programmiersprache Pascal. Überblick über Programmiersprachen, URL `http://www.fh-jena.de/contrib/fb/gw/gmueller/Kurs_halle/pas12.html`, Abruf am 23.02.2010.

Computerwoche (1975): Interaktives Programmieren als Systems-Schlager, URL `http://www.computerwoche.de/heftarchiv/1975/47/1205421/`, Abruf am 23.02.2010.

Computerwoche (1979): Programm-Entwicklung im Dialog, URL `http://www.computerwoche.de/heftarchiv/1979/18/1192688/`, Abruf am 23.02.2010.

Daum, B. (2006): *Das Eclipse-Codebuch. 182 Tipps, Tricks und Lösungen für Eclipse-spezifische Probleme*, 1. Aufl., dpunkt.verlag, URL `http://www.dpunkt.de/leseproben/2371/Kapitel_3.pdf`, Abruf am 23.02.2010.

Daum, B. (2007): *Rich-Client-Entwicklung mit Eclipse 3.2. Anwendungen entwickeln mit der Rich Client Platform*, 2. Aufl., dpunkt.verlag.

Dick, E. (2000): *Multimediale Lernprogramme und telematische Lernarrangements. Einführung in die didaktische Gestaltung.*, BW Bildung und Wissen. Verlag und Software GmbH, Nürnberg, Deutschland.

Ebner, M.; Klaun, C. (1998): *Pascal mit Delphi4*, Addison Wesley Longman Verlag GmbH, Bonn, Deutschland.

Eclipse Foundation (a): Eclipse Wiki. FAQ How do I upgrade Eclipse?, URL `http://wiki.eclipse.org/FAQ_How_do_I_upgrade_Eclipse%3F`, Abruf am 23.02.2010.

Eclipse Foundation (b): Eclipse Wiki. FAQ What is the Update Manager?, URL `http://wiki.eclipse.org/FAQ_What_is_the_Update_Manager%3F`, Abruf am 23.02.2010.

Eclipse Foundation (c): How To Keep Up To Date, URL http://www.eclipse.org/articles/article.php?file=Article-Update/index.html, Abruf am 23.02.2010.

Eclipse Foundation (2009): Eclipse 3.5 Plug-in Migration Guide, URL http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.platform.doc.isv/porting/eclipse_3_5_porting_guide.html?view=co, Abruf am 23.02.2010.

Engels, G.; Schäfer, W. (1989): *Programmentwicklungsumgebungen. Konzepte und Realisierung.*, Teubner Verlag, Stuttgart, Deutschland.

Fleischer, W. (1992): *EDV-Didaktik. Wie man EDV-Wissen richtig erklärt, vermittelt, aufbereitet und dokumentiert.*, 2. Aufl., IWT-Verlag, Vaterstettem, Deutschland.

Forneck, H. J.; Goorhuis, H. (1990): *Beiträge zur Didaktik der Informatik*, Verlag Moritz Diesterweg, Frankfurt am Main, Deutschland.

Gavab Research Group (): Eclipse Gavab, URL http://www.gavab.es/wiki/eclipsegavab/, Abruf am 23.02.2010.

Gerald T. Aitken (2007): Early PC Computing, URL http://earlypccomputing.com/, Abruf am 26.01.2010.

Gumm, H. P.; Sommer, M. (2008): *Einführung in die Informatik*, 8., volständig überarbeitete Aufl., Oldenbourg Verlag, München, Deutschland.

Herzwurm, G.; Hierholzer, A.; Kunz, M. (1994): CASE - Die Aufgaben haben sich gewandelt, URL http://www.systementwicklung.uni-koeln.de/fileadmin/wwww_Inhalte/forschung/artikel/1997undfrueher/CASE%20-%20Die%20Aufgaben%20haben%20sich%20gewandelt.pdf, Abruf am 23.02.2010.

Huberwieser, P. (2004): *Didaktik der Informatik. Grundlagen, Konzepte, Beispiele*, 3., überarbeitete und erweiterte Aufl., Springer-Verlag, Berlin / Heidelberg, Deutschland.

IBM Corporation (2005): Eclipse 3.1 Documentation. Platform Plug-in Developer Guide, URL http://eclipse.org/documentation/, Abruf am 23.02.2010.

IBM Corporation (2009): Eclipse Project Release Notes. Release 3.5.0, URL `http://eclipse.org/documentation/`, Abruf am 23.02.2010.

Keuffer, J.; Kublitz-Kramer, M. (2008): *Was braucht die Oberstufe? Diagnose, Förderung und selstständiges Lernen*, Beltz Verlag, Weinheim, Deutschland / Basel, Schweiz.

Klingen, D. L. H.; Otto, A. (1986): *Computereinsatz im Unterricht. Der pädagogische Hintergrund.*, J.B. Metzler, Stuttgart, Deutschland.

Lehrerinnen und Lehrer der Gymnasien in Zusammenarbeit mit dem Sächsischen Staatsinstitut für Bildung und Schulentwicklung (2005): *Lehrplan Gymnasium. Naturwissenschaftliches Profil*, Sächsisches Staatsministerium für Kultus, URL `http://www.sachsen-macht-schule.de/apps/lehrplandb/`, Abruf am 23.02.2010.

Leitenberger, B. (): Die Geschichte von Turbo Pascal, URL `http://www.bernd-leitenberger.de/turbo-pascal-history.shtml`, Abruf am 23.02.2010.

Microsoft Corporation (2001): Versionsverlauf von Microsoft QuickBasic für MS-DOS, URL `http://support.microsoft.com/kb/39730`, Abruf am 23.02.2010.

Nödler, J. (2005): Web Engineering: CASE-Tools. Software zur Entwicklung von Internet-Anwendungen, URL `http://www.swe.informatik.uni-goettingen.de/notes/WS2004/neukirchen/case-tools_Noedler.pdf`, Abruf am 23.02.2010.

NetApplications.com (): Operating System Market Share October 2009, URL `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=10&qpmr=24&qpdt=1&qpct=3&qptimeframe=M&qpsp=129`, Abruf am 23.02.2010.

Neumann, A. (2009): Crème de Cocoa: Eclipse Galileo veröffentlicht, URL `http://www.heise.de/newsticker/meldung/Creme-de-Cocoa-Eclipse-Galileo-veroeffentlicht-185447.html`, Abruf am 23.02.2010.

Normenausschuss Ergonomie (NAErg) im DIN, Normenausschuss Informationstechnik und Anwendungen im DIN (2008): Ergonomie der Mensch-System-Interaktion. Teil 110: Grundsätze der Dialoggestaltung (ISO 9241-110: 2006).

Object Technology International, Inc. (2003): Eclipse Platform Technical Overview, URL `http://eclipse.org/documentation/`, Abruf am 23.02.2010.

Pasclipse Project (): Pasclipse, URL `http://sourceforge.net/projects/pasclipse/`, Abruf am 23.02.2010.

Reichert, R.; Noevergelt, J.; Hartmann, W. (2005): *Programmieren mit Kara. Ein spielerischer Zugang zur Informatik.*, Springer-Verlag, Berlin / Heidelberg, Deutschland.

Schneider, U.; Werner, D. (2001): *Taschenbuch der Informatik*, 4., aktualisierte Aufl., Fachbuchverlag, Leipzig, Deutschland.

Schubert, S.; Schwill, A. (2004): *Didaktik der Informatik*, 1. Aufl., Spektrum Akademischer Verlag, Berlin / Heidelberg, Deutschland.

Shavor, S.; D'Anjou, J.; Fairbrother, S.; Kehn, D.; Kellerman, J.; McCarthy, P. (2003): *The Java Developer's Guide to Eclipse*, überarbeitete Aufl., Addison-Wesley, München, Deutschland.

Sommerville, I. (2007): *Software Engineering*, 8., aktualisierte Aufl., Person Education Limited.

SpiegelWissen (1983): Akten auf Knopfdruck. Eine Münchner Programmier-Firma feiert im Heimatland des Computers, den USA, erstaunliche Erfolge., URL `http://www.spiegel.de/spiegel/print/d-14020896.html`, Abruf am 23.02.2010.

Thomson, D. (2003): *The Java Developer's Guide to Eclipse. Vorwort*, überarbeitete Aufl., Addison-Wesley, München, Deutschland.

Thurner, R. (1995a): Eine kurze Geschichte der Software-Entwicklung (Teil 2 und Schluss) Homogene Programmerstellung bleibt ein unerfuellbarer Traum, URL `http://www.computerwoche.de/heftarchiv/1995/28/1115508/`, Abruf am 23.02.2010.

Thurner, R. (1995b): RAD/Ein kurze Geschichte der Softwareprogrammierung (Teil 1). Drei Jahrzehnte lang haben die Entwickler um Struktur gerungen, URL `http://www.computerwoche.de/heftarchiv/1995/27/1115369/`, Abruf am 23.02.2010.

Walezak, M. (2008): IDE-Unterstützung für graduelle und individuelle Spracherweiterung. Eine Untersuchung am Beispiel von Eclipse, Java und VJ. Diplomarbeit an der Universität Hamburg, URL `http://swt-www.informatik.uni-hamburg.de/publications/papers/Dipl/da_mw_20080505.pdf`, Abruf am 23.02.2010.

Wolmeringer, G. (2005): *Java lernen mit Eclipse 3*, 4. Aufl., Galileo Press GmbH, Bonn, Deutschland.

Wütherich, G.; Hartmann, N.; Kolb, B.; Lübken, M. (2009): *Die OSGi Service Platform - Eine Einführung mit Eclipse Equinox*, dpunkt.verlag, Heidelberg, Deutschland, URL `http://drupal.osgibook.org/`, Abruf am 23.02.2010.

Name: Michael Kuhardt

Matrikel-Nr.: 2936155

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Dresden, 23. Februar 2010

Unterschrift