# Kapitel 4: Programmiersprachen und –umgebungen

Zur Wiederholung: *Informatik* ist die Wissenschaft von der automatischen Verarbeitung von Informationen; ein *Algorithmus* ist ein präzises, schrittweises und endliches Verfahren zur Verarbeitung (Umformung) von Informationen, und ein *Programm* ist die Repräsentation eines Algorithmus zur Ausführung auf einer Maschine

Während v. Neumann noch der Ansicht war, dass „Rechenzeit zu wertvoll für niedere Aufgaben wie Übersetzung“ ist, sind heute dagegen die Personalkosten der *Hauptfaktor* bei den Softwarekosten. Darüber hinaus ist die Codierung in Maschinensprache oder Assembler sehr fehleranfällig (wie jeder, der die Haufgaben bearbeitet hat, wohl gemerkt haben dürfte). Daher sucht man nach problemorientierten statt maschinenorientierten Beschreibungsformen. Die Verständlichkeit eines Programms ist oftmals wichtiger als die optimale Effizienz. Vorteile höherer Programmiersprachen zeigen sich

- bei der *Erstellung* von Programmen:
  - schnellere Programmerstellung
  - sichere Programmierung.
- bei der *Wartung* von Programmen:
  - bessere Lesbarkeit
  - besseres Verständnis der Algorithmen.
- wenn Programme *wiederverwendet* werden sollen:
  - Verfügbarkeit auf vielen unterschiedlichen Rechnern; vom Zielrechner unabhängige Entwicklungsrechner.

## 4.1 Programmierparadigmen

Unter einem Programmierparadigma versteht man ein Sprachkonzept, welches als Muster prägend für die Programme einer bestimmten Gruppe oder Sprache ist. Gängige Programmierparadigmen sind

- funktional / applikativ
- imperativ
- objektorientiert
- logikbasiert
- deklarativ
- visuell / datenflussorientiert

- *funktionale* oder *applikative Sprachen* betrachten ein Programm als *mathematische Funktion* f, die eine Eingabe I in eine Ausgabe O überführt: O = f(I). Variablen sind Platzhalter im Sinne der Mathematik (Parameter).
  **Ausführung = Berechnung des Wertes mittels Termersetzung.**
  **Beispiele: Lisp, SML, Haskell, ...**

  SML:

```
fun fak (n) = if n < 1 then 1 else n * fak (n-1)
```

  LISP:

```
(defun fak (n)
  (cond (le n 1) 1 (* n (fak (– n 1)))))
```

- *imperative Sprachen* unterstützen das ablauforientierte Programmieren. Werte werden sog. *Variablen* (= Speicherplätzen) zugewiesen. Diese können ihren Zustand ändern, d.h. im Laufe der Zeit verschiedene Werte annehmen (vgl. von-Neumann-Konzept).
  **Ausführung = Folge von Variablenzuweisungen.**
  **Beispiele: Algol, C, Delphi, ...**

```
// Fakultätsfunktion in C:

#include <stdio.h>
int fakultaet(int n) {
  int i, fak = 1;
  for ( i=2; i<=n; i++ )
    fak *= i;
  return fak;
}

int main(void) {
  int m, n;
  for ( n=1; n<=17; n++ )
    printf("n = %2d n! = %10d\n", n, fakultaet(n));
}
```

- *objektorientierte Sprachen* sind imperativ, legen aber ein anderes, in *Objekten* strukturiertes Speichermodell zugrunde. Objekte können eigene lokale Daten und Methoden speichern, voneinander erben und miteinander kommunizieren.
  **Ausführung = Interaktion von Agenten.**
  **Beispiele: Smalltalk, C++, Java**

```
//Java Polymorphie-Beispiel
import java.util.Vector;

abstract class Figur
{ abstract double getFlaeche();
}

class Rechteck extends Figur
{ private double a, b;
  public Rechteck ( double a, double b )
  { this.a = a;
    this.b = b; }
  public double getFlaeche() {return a * b;}
}

class Kreis extends Figur
{ private double r;
  public Kreis( double r )
  { this.r = r; }
  public double getFlaeche()
  {return Math.PI * r * r;
  }
}

public static void main( String[] args )
  { Rechteck re1 = new Rechteck( 3, 4 );
    Figur kr2 = new Kreis( 8 );
 ...
```

- *logikbasierte Sprachen* betrachten ein Programm als (mathematisch-) *logischen Term* t, dessen Auflösung (im Erfolgsfall) zu gegebenen Eingabewerten I passende Ausgabewerte O liefert: t(I,O) $\rightarrow$ true.

**Ausführung = Lösen eines logischen Problems.**

**Beispiel: Prolog**

```
fak(0,1).
fak(N,X):- N > 0, M is N - 1, fak(M,Y), X is N * Y.


mutter (eva, maria).
mutter (maria, uta).
grossmutter(X,Y) :- mutter(X,Z), mutter(Z,Y).
?- grossmutter (eva, uta)
yes.
?- grossmutter (uta, eva)
no.
```

- *deklarative Sprachen* betrachten ein Programm als eine Menge von *Datendefinitionen* (Deklarationen) und darauf bezogenen *Abfragen*.

  **Ausführung = Suche in einem Datenbestand.**

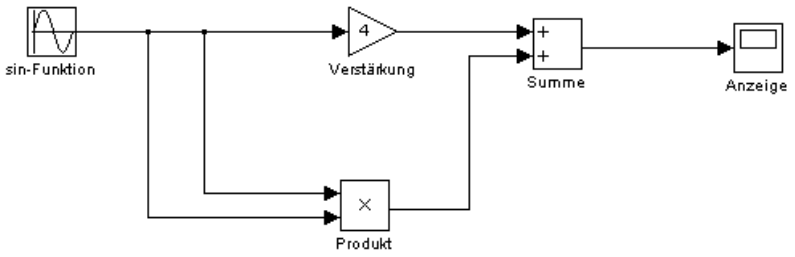
  **Beispiel: SQL**

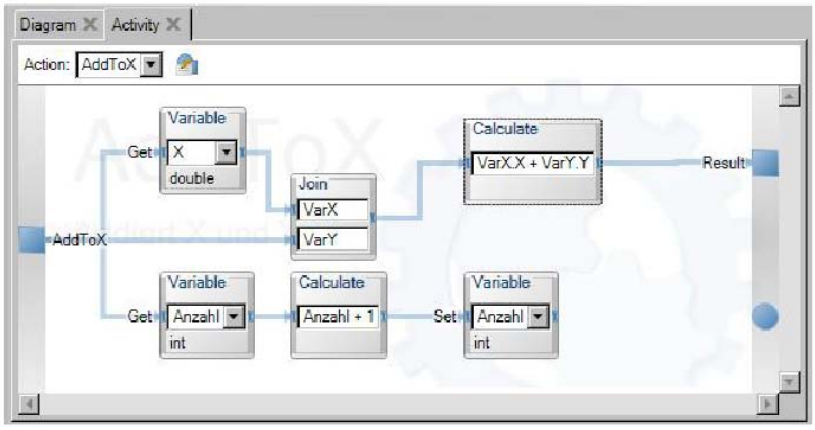
```
SELECT A_NR,
        A_PREIS As Netto,
        0.16 As MwSt,
        A_PREIS * 1.16 As Brutto,
FROM ARTIKEL
WHERE A_PREIS <= 100
ORDER BY A_PREIS DESC
```

- *datenflussorientierte Sprachen* stellen ein Programm dar, indem sie den *Fluss der Signale* oder *Datenströme* durch Operatoren (Addierer, Integratoren) beschreiben

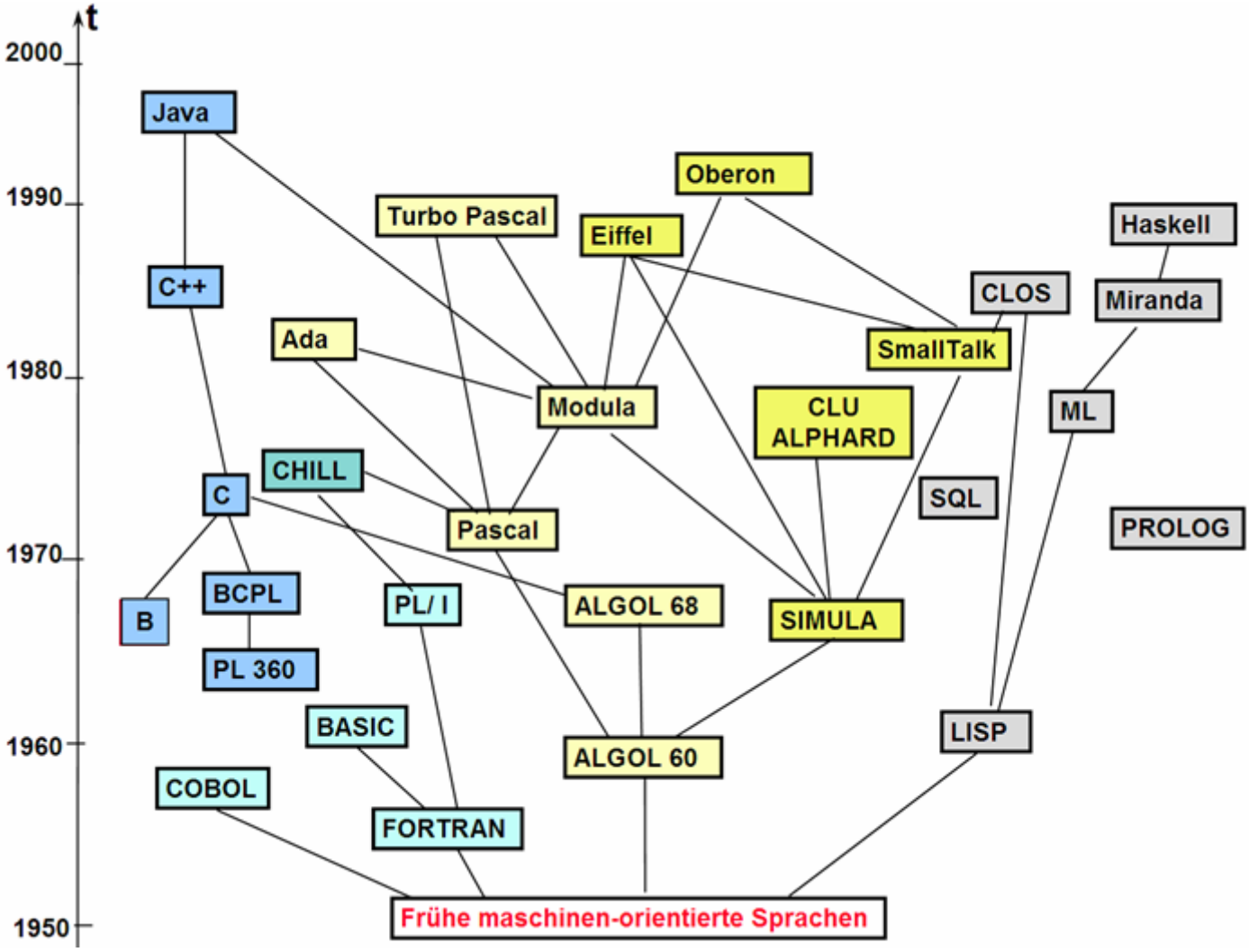
  **Ausführung = Transformation von Datenströmen**

  **Beispiel: Simulink, Microsoft Visual Programming Language (VPL)**






## 4.2 Historie und Klassifikation von Programmiersprachen

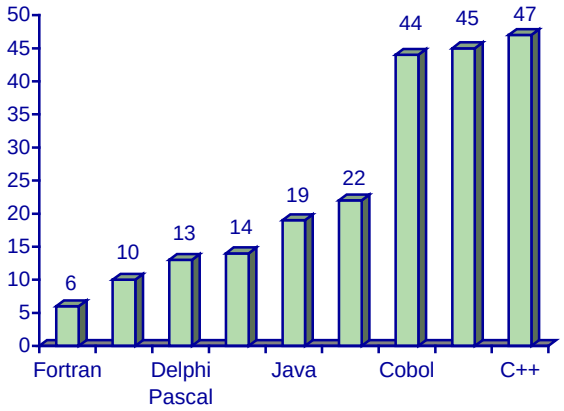



Zur obigen Grafik gibt es viele Varianten und Alternativen, je nach Vorliebe des Diagramm-Erstellers.

Programmiersprachen lassen sich *klassifizieren* nach

- Anwendungsgebiet: Ingenieurwissenschaften (Fortran), kommerzieller Bereich (Cobol), künstliche Intelligenz (Prolog, Lisp), Systemsoftware (Assembler, C), Steuerungssoftware (C, Ada, Simulink), Robotik (C, NQC, VPL), Internet / Arbeitsplatzsoftware (Java, C++), Datenbanken (SQL), …

    - Spezialsprachen („DSL, domain specific languages“): SPSS (Statistik); TeX/LaTeX, PostScript (Textverarbeitung); Lex, Yacc (Compilerbau); Z, B, CSP (Programmspezifikation); UML, SimuLink (Modellierung)
- Verbreitungsgrad: (derzeit )industrierelevante, überlieferte und akademische Sprachen
- Historischer Entwicklung: „Stammbaum“ der Programmiersprachen, siehe oben
- Programmiersprachengeneration: Abstraktionsgrad;
    - 1. Generation: Maschinensprachen
    - 2. Generation: Assemblersprachen (x86-MASM, ASEM-51)
    - 3. Generation: Algorithmische Sprachen (Algol, Delphi, …)
    - 4. Generation: Anwendungsnahe Sprachen (VBA, SQL, …)
    - 5. Generation: Problemorientierte Sprachen, KI-Sprachen (Prolog, Haskell, …)
- Programmierparadigmen (siehe Kapitel 4.1)
- Verfügbaren Sprachelementen
    - Kontrollfluss, Datentypen
    - Rekursion oder iterative Konstrukte
    - Sequentielle oder parallele Ausführung
    - Interpretiert oder compiliert
    - Realzeitfähig oder nicht realzeitfähig
    - Streng typisiert, schwach typisiert oder untypisiert
    - Statisch oder dynamisch typisiert
    - Textuell oder graphisch
    - …




Das Balkendiagramm zeigt die in deutschen Softwarehäusern überwiegend verwendeten Sprachen, zitiert nach (Bothe, Quelle: Softwaretechnik-Trends Mai 1998). Mehrfachnennungen waren bei der Befragung möglich.

Die „babylonische Sprachvielfalt“ der verschiedenen Programmiersprachen verursacht Kosten: Portabilitätsprobleme, Schulungsmaßnahmen, Programmierfehler. Daher gab es immer wieder Versuche, Sprachen zu vereinheitlichen (ADA, ANSI-C, …). Trotzdem kam es in der Forschung immer wieder zu neuen Ideen, neuen Konzepten und in der Folge zu neuen Programmiersprachen. Daher wird es auf absehbare Zeit viele verschiedene Sprachen und Dialekte geben. Für Informatiker ist es deshalb wichtig, die verschiedenen Konzepte zu kennen und sich schnell in neue Sprachen einarbeiten zu können.

## 4.3 Java und Groovy

*Java* wurde seit 1995 bei der Firma *Sun Microsystems* entwickelt. Ursprünglich war die Sprache zur Implementierung von sicheren, plattform-unabhängigen Anwendungen gedacht, die über das Internet verbreitet und bezogen werden können. Ein wichtiges Konzept ist das

sog. *Applet*, eine Mini-Anwendung, die innerhalb einer Web-Seite läuft. Inzwischen sind praktisch alle Internet-Browser „Java-fähig“ und können Applets ausführen. Künftige Anwendungen von Java liegen voraussichtlich im Bereich der kommunizierenden Geräte („Internet der Dinge“). Vorzüge von Java sind:

- Java ist eine *mächtige Programmiersprache*, die die Sprachkonzepte herkömmlicher Programmiersprachen wie C oder Pascal mit OO-Konzepten und Konzepten zur Parallelverarbeitung und Netz-Verteilung verbindet. Zielsetzung war es, eine möglichst schlanke Sprache zu schaffen, die das Klassenkonzept ins Zentrum der Sprache stellt.
- Das Konzept des *Java Byte Code* und der *Java Sandbox* erlauben eine einfache Portierbarkeit und sichere Ausführbarkeit auf den unterschiedlichsten Plattformen. Die Offenheit und hohe Portabilität von Java macht die Sprache zu einem guten Werkzeug für das Programmieren von Netz-Anwendungen.
- Java gilt als *robuste* Sprache, die viele Fehler vermeiden hilft, z.B. aufgrund ihres Typkonzepts und der automatischen Speicherbereinigung. Java-*Compiler* sind im Allgemeinen vergleichsweise schnell und produzieren effizienten Code.
- Durch die große Beliebtheit in akademischen Kreisen und bei Open-Source-Anwendern hat Java eine *umfangreiche Klassenbibliothek*, und es gibt viele *Unterstützungswerkzeuge*.

Im Kern ist Java jedoch eine Programmiersprache mit imperativen und rekursiven Konzepten ähnlich wie Pascal oder C.

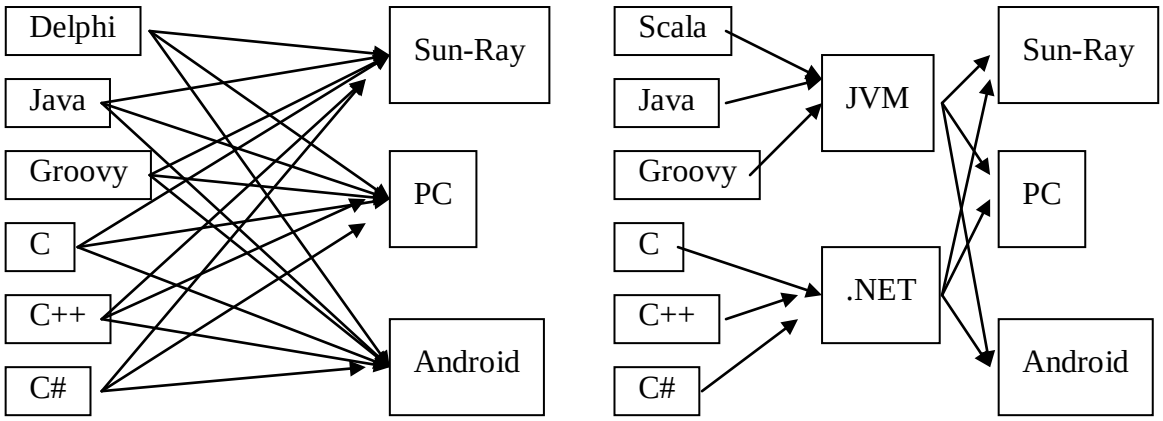
- So weist Java die meisten aus diesen Sprachen bekannten Konzepte wie Variablen, Zuweisungen, Datentypen, Ausdrücke, Anweisungen etc. auf - mit einer keineswegs verbesserten Syntax
- Viele syntaktische und strukturelle Mängel und Ungereimtheiten sind nur mit der C / C++ Historie zu erklären

Die Ausführung eines Java-Programms geschieht im Allgemeinen in folgenden Schritten:

(1) Eingabe des Programmtextes in einem Editor oder einer IDE (integrierten Entwicklungsumgebung).
(2) Compilation, d.h. Übersetzung des Programms (oder von Teilen des Programms) in Byte-Code für die „virtuelle Java-Maschine“ (JVM).
(3) Binden und Laden, d.h. Zusammentragen aller verwendeten Bibliotheksklassen, Ersetzung von Sprungadressen usw., und ggf. Übertragung des Programms auf die Zielplattform.
(4) Aufruf des Programms und Start des Prozesses.

Bei der so genannten „just-in-time-compilation“ wird Schritt (2), (3) und (4) verschränkt zur Laufzeit ausgeführt, d.h., es werden immer nur die Teile übersetzt (z.B. von ByteCode in Maschinencode), die gerade benötigt werden. Die meisten modernen Compiler bzw. Laufzeitumgebungen unterstützen diese Methode.

**Konzept der virtuellen Maschine:**
Bis in die 1990-Jahre musste für jede Sprache und jede mögliche Hardware-Plattform ein eigener Compiler erstellt werden. Das Konzept einer „virtuellen Maschine“, d.h., eines standardisierten Befehlssatzes, erlaubt es, Compiler zu erstellen, die Code für diese idealisierte Maschine erstellen (in Java: Byte Code für die Java Virtual Machine). Auf den einzelnen Plattformen muss dann nur noch eine „Laufzeitumgebung“ definiert werden, die diese virtuelle Maschine mit der realen Hardware simuliert.




**Java-Historie**

| | |
|---|---|
| ab 1991 | Bei der Firma Sun wird von J. Gosling und Kollegen auf der Basis von C++ eine Sprache namens Oak (Object Application Kernel) für den Einsatz im Bereich der Haushalts- und Konsumelektronik entwickelt. Ziele: Plattform-Unabhängigkeit, Erweiterbarkeit der Systeme und Austauschbarkeit von Komponenten. |
| 1993 | Oak wird im Green-Projekt zum Experimentieren mit graphischen Benutzer-Schnittstellen eingesetzt und später (wegen rechtlicher Probleme) in Java umbenannt. Zu diesem Namen wurden die Entwickler beim Kaffeetrinken inspiriert. |
| 1994 | Das WWW beginnt sich durchzusetzen. Java wird wegen der Applet-Technologie „die Sprache des Internets“ |
| seit 1995 | Sun bietet sein Java-Programmiersystem JDK (Java Development Kit) mit einem Java-Compiler und Interpreter kostenlos an. |
| ab 1996 | Unter dem Namen JavaBeans wird eine Komponenten-Architektur für Java-Programme entwickelt und vertrieben. |
| ab 2001 | Eclipse-Projekt: integrierte Entwicklungsumgebung für Java und (darauf aufbauend) andere Sprachen und Systeme. |
| 2006-2007 | JDK als Open-Source freigegeben (OpenJDK) |

**Groovy**

Groovy ist eine Erweiterung von Java, die 2003 definiert wurde mit den folgenden Zielen:

- skriptartige Sprache, d.h., einzelne Anweisungen können sofort ausgeführt werden
- dynamische Typisierung, d.h., es ist möglich, den Typ von Objekten zur Laufzeit vom System bestimmen zu lassen
- funktionale Programmierung mit Closures, d.h. Auffassung von Code als Daten, der zur Laufzeit analysiert und übersetzt wird
- originäre Unterstützung von Listen, Mengen, endlichen Funktionen; reguläre Ausdrücke und Mustervergleich für Textbearbeitungsaufgaben
- Schablonensystem für HTML, SQL; Scripting von Office- und anderen Anwendungen
- einfachere, „sauberere“ Syntax als Java
- weitgehende Kompatibilität zu Java, Code für die Java Virtual Machine

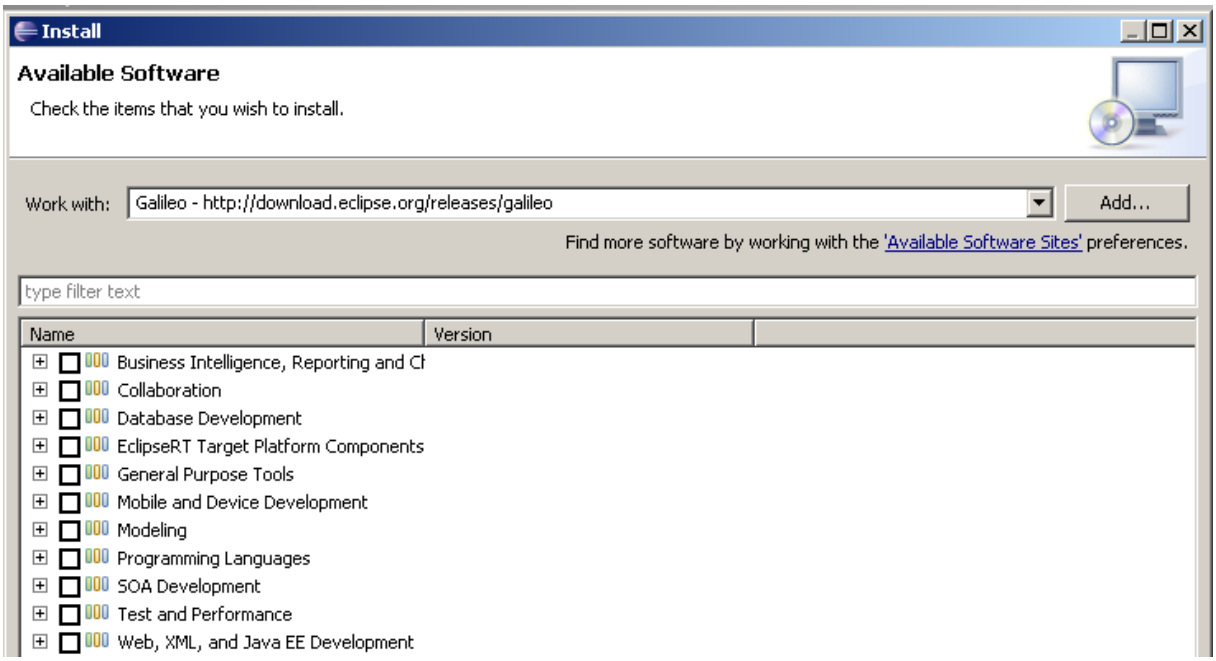
Auf Grund der einfachen Handhabung in der Groovy-Konsole eignet sich Groovy besonders für Programmieranfänger und für die „schnelle“ Erstellung von Programmen („agile Software-Entwicklung“).

## 4.4 Programmierumgebungen am Beispiel Eclipse

Während der Erstellung eines Programms sind vom Programmierer verschiedene Aufgaben zu erledigen. Dafür stehen verschiedene Werkzeuge zur Verfügung:

- Eingabe des Textes – Texteditor, syntaxgesteuerte Formatierer
- Übersetzung in Maschinencode – Compiler bzw. Interpreter
- Binden zu einem lauffähigen Programm – Linker, Object Code Loader
- Finden von semantischen Fehlern – Debugger, Object Inspector
- Optimierung des Programms – Profiler, Tracer
- Auffinden von Bibliotheksroutinen – Library Class Browser
- Design der graphischen Benutzungsoberfläche – GUI-Builder
- Modellierung des Problems – Modeling Tools
- Verwaltung verschiedener Versionen – Versionskontrollsystem
- Dokumentation – Dokumentationsgeneratoren, Klassenhierarchieanzeiger
- Testen – Testgeneratoren

Ursprünglich waren alle diese Funktionen in separaten Werkzeugen realisiert. Das ist recht umständlich, weil der Programmierer ständig zwischen den Werkzeugen wechseln muss. Daher begann man bereits in den 1970-er Jahren, die verschiedenen Aktivitäten beim Übersetzen und Binden durch Skripten zusammenzufassen (Make-files). Später wurden integrierte Entwicklungsumgebungen (integrated development environments, IDE) geschaffen, die den Texteingabe-, Übersetzungs- und Ausführungsprozess zusammenfassten. 2001 begann, initiiert durch die IBM, die Entwicklung der Eclipse IDE, einer freien, erweiterbaren Entwicklungsumgebung. Ursprünglich war Eclipse nur eine IDE für Java. Durch den Plug-In-Mechanismus ist es beliebigen Entwicklern möglich, Erweiterungen (auch für andere Programmiersprachen) vorzunehmen, so dass heute über 100 verschiedene integrierbare Werkzeuge vorliegen.




Eclipse zeichnet sich vor allem aus durch:

- minimale Kernfunktionalität, extreme Erweiterbarkeit
- Persistenz (gesamter Entwicklungszustand bleibt erhalten)
- verschiedene Sichten auf ein Projekt („Views“), projektspezifisch konfigurierbare Perspektiven (Fenster, Leisten,…)

- syntaxgesteuerte Editoren, Just-in-Time Compiler, …
- JDT, CDT: Java / C++ Development Tools
- EMF, GEF: Eclipse Modelling Frameworks, Graphical Editing Framework

Zu Eclipse gibt es eine umfangreiche online-Dokumentation im Programm selbst.


Die aktuelle Version von Eclipse kann bezogen werden unter http://www.eclipse.org/
Zur Installation des Groovy-Plugins ist in Eclipse der folgende Server anzugeben:
http://dist.springsource.org/milestone/GRECLIPSE/e3.5/