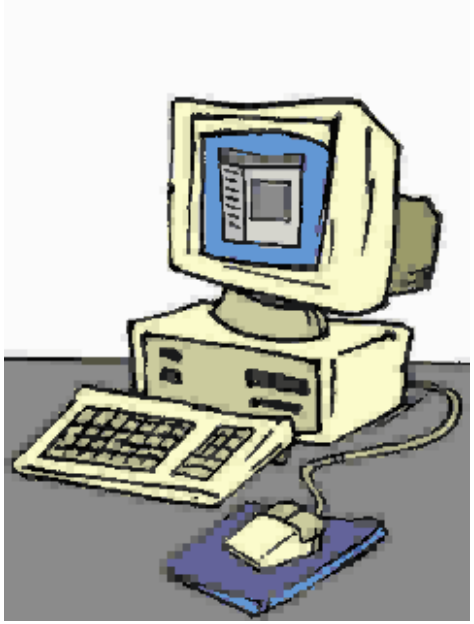
7. Implementierung



Dr.-Ing. Clemens Reichmann

Clemens.Reichmann@vector.com,
Tel. 0721-91430-200

Institut für Technik der Informationsverarbeitung
Fakultät für Elektrotechnik & Informationstechnik
Universität Karlsruhe (TH)

Karlsruher Institut für Technologie

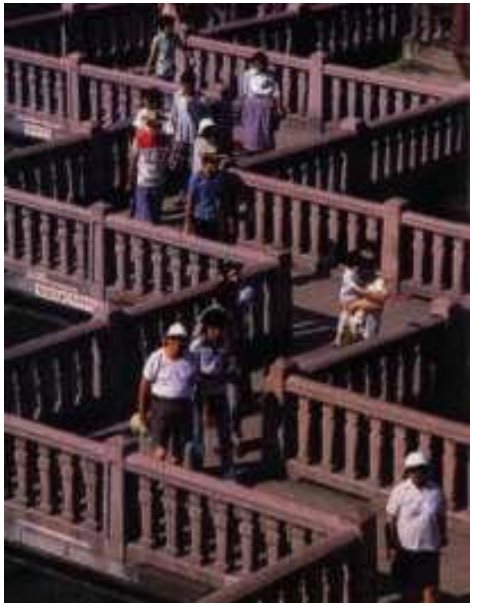
Inhalt 7. Implementierung

7.1 Entwicklungswerkzeuge (I)

- Programmier-Werkzeuge:
  - **Editor** Text-Editor zum Erstellen des Programm-Quelltextes (Quellcode) (Notepad, vi, Emacs, NoteTab, joe, ...)
  - **Compiler** übersetzt den Quelltext und liefert als Ergebnis eine Datei mit Binärcode (oder Fehlermeldungen).
  - **Interpreter** führt vorliegenden Quelltext (oder "Zwischencode" wie z.B. Java-Byte-Code) auf einem Rechnersystem aus.
  - **Debugger** Programm zur Fehlersuche
  - **Linker** Verbinden von Binärprogrammen (Bibliotheken) zu einem ausführbaren Programm.
  - **IDE** (Integrated Development Environment) Programmsysteme zur professionellen/komfortablen Programmentwicklung

Entwicklungswerkzeuge (II)

- Compiler, ggf. Interpreter, Debugger, Linker sind Programme, die
  - zum Programmiersystem einer Programmiersprache gehören
  - je nach Programmiersprache und Entwicklungssystem (IDE)
    - kommerziell erhältlich,
    - frei verfügbar sind oder
    - zum Betriebssystem gehören.



JAVA Werkzeuge

- **JDK/SDK (Java Developers Kit/Software Developers Kits)**
- **Standard-Werkzeuge für die Java-Programmierung (frei bei Sun Microsystems erhältlich „java.sun.com“).**

`javac`	**Java-Compiler**. Erzeugt aus dem Java-Quelltext Java-Byte-Code.
`java`	Java-Byte-Code-**Interpreter** zum Ausführen von Applikationen (startet die Java Virtuelle Maschine).
`appletviewer`	Ausführen und Anzeigen von Applets
`javadoc`	Generieren von **Dokumentation** aus Java-Quelltext
`jdb`	**Debugger**

7.1.1 Compiler

- Übersetzer
- Prüft Syntax
- Erzeugt Maschinencode/Bytecode

Maschinensprache (Binärcode)	Assembler-Sprache (Prozessorsteuerung)	Höhere Programmiersprachen

0011001001001010 1	MOVEM.L D3-D5,-(SP)
	MOVE.L D1,D3
	ADD D1,D2
	SWAP
		float x,y;
		x = 7.9;
		y = x * 4.001;

Fehlersuche – grundlegendes Vorgehen

Fehler-detektion

- Beobachten eines unerwünschten oder unspezifizierten Zustands des Systems (**error**) oder Versagen des Systems (**failure**).

Testen / Monitoring

Fehler-lokalisierung

- Auffinden der Stelle an welcher das Versagen des Systems sichtbar wird.
- Suchen der Stelle, an der das System einen unerwünschten Zustand zuerst einnimmt.

Fehler-analyse

- Finden der Fehlerursache (**defect**)

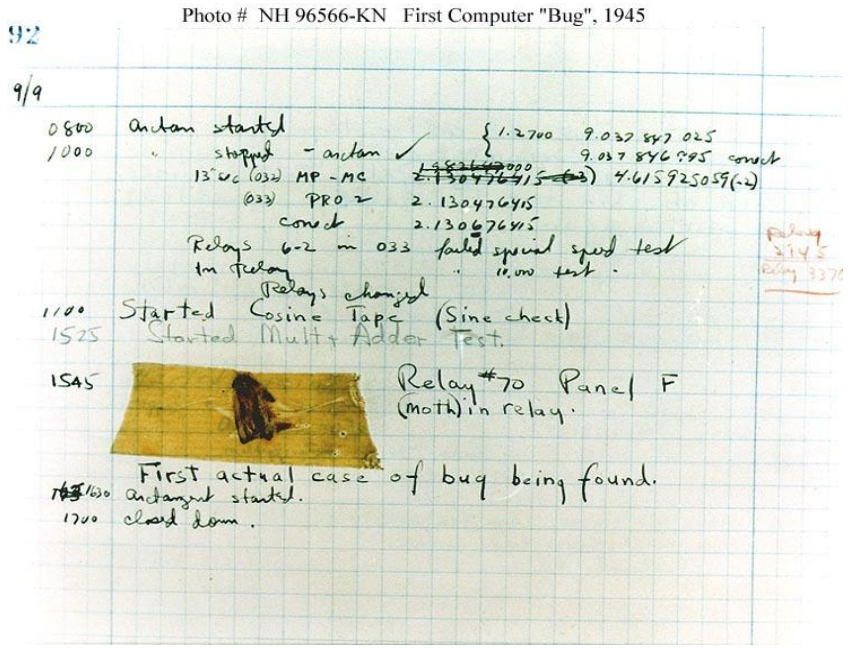
Debugging / Simulation

- Beheben des Fehlers

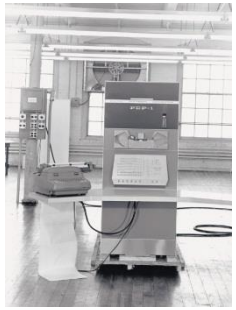
Debugging - Etymologie

- Edison in einem Brief von 1878:
  - It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise -- this thing gives out and [it is] then that **"Bugs"** -- as such little faults and difficulties are called -- show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

- Erster tatsächlicher Computer-Bug:
  - 1945 verfängt sich eine Motte *(Bug)* in einem Schaltrelais eines Computers der US Navy
  - Insekt wird gefunden und in Logbuch dokumentiert



Klassisches Debugging



- Geeignet für imperative Sprachen
- Erstmals 1961: „DEC Debugging Tool “ (DDT) von Digital Equipment Corporation (DEC) für PDP-1

- **Ablaufkontrolle *(run-control)***
  - Setzen von Haltepunkten (*breakpoints*)
  - Einzelschritt
  - Starten/Stoppen

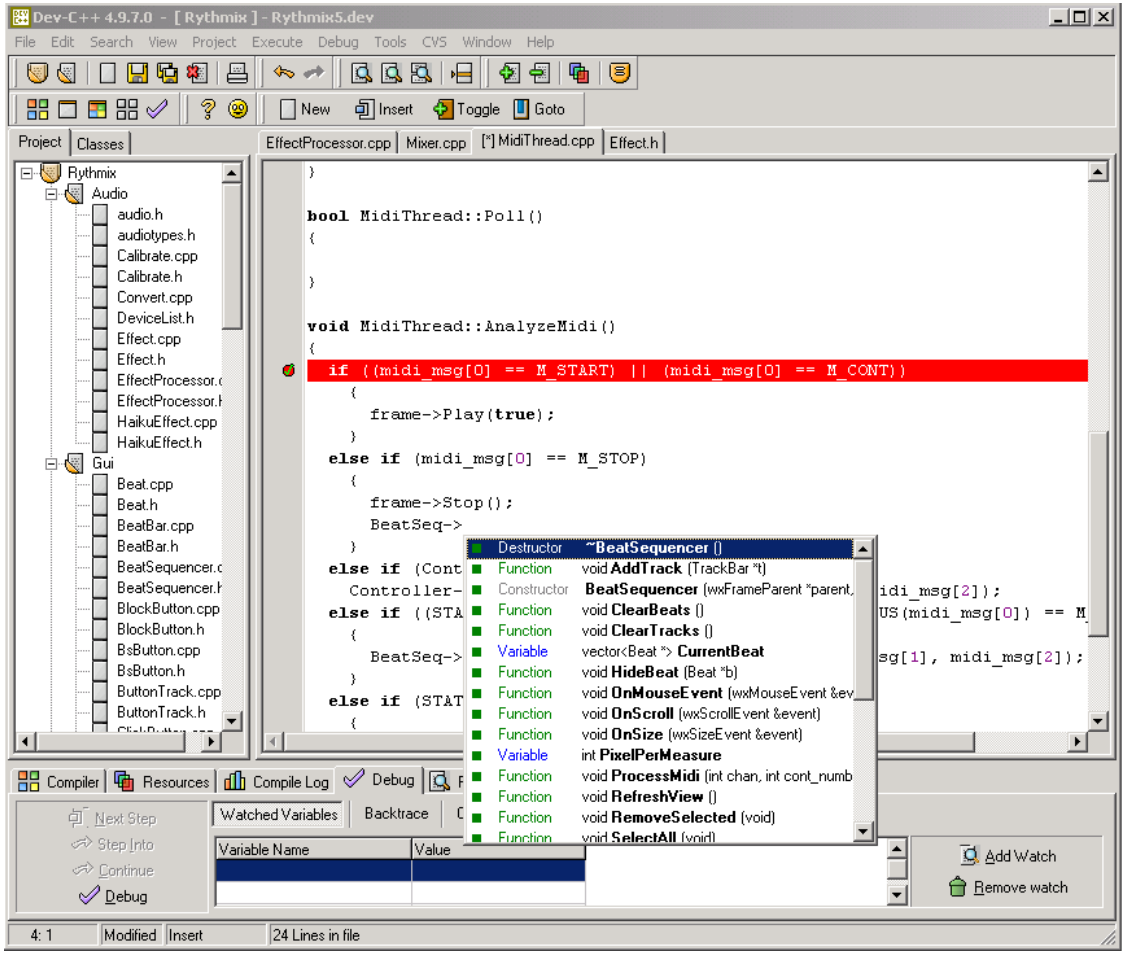
- **Inspektion**
  - Quelltextansicht
  - Programmzustand (Register, Variablen)
  - Aufrufliste (*call stack*)
  - Speicher
  - Threads / Prozesse

- **Vorgehen**
  1. Aufstellen einer These über mögliche Position des Fehlers
  2. Setzen eines Haltepunkts vor der vermuteten Position
  3. Annäherung mit Hilfe von Breakpoints / Einzelschritt, dabei Überprüfung des Programmzustands
  4. These falsch, weiter mit 1.
  5. Ansonsten: Korrigieren des Fehlers

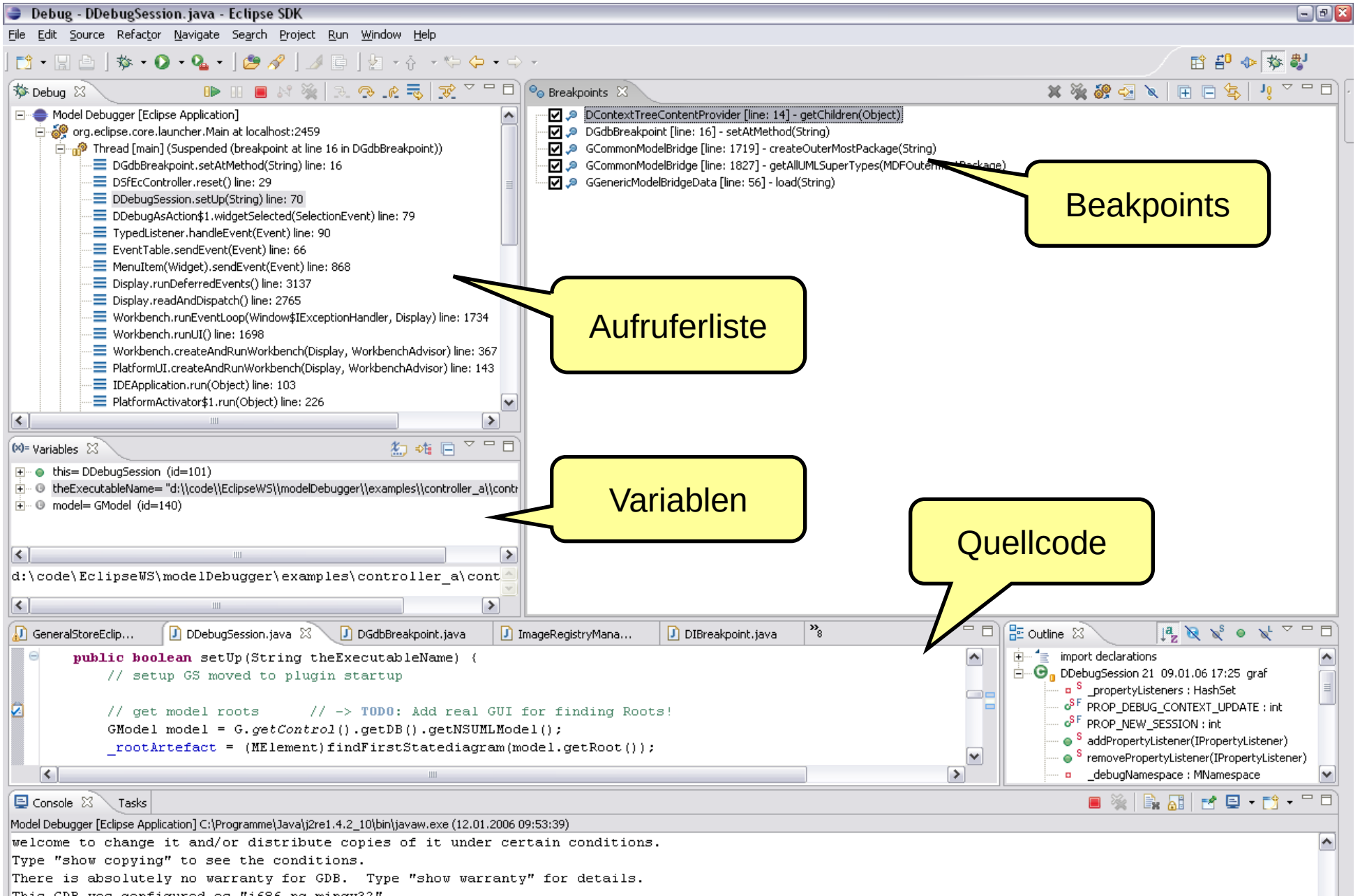
Debug eines Programms

- Möglichkeiten im Debug:
  - Anhalten an Breakpoints
  - Anschauen des aktuellen Wertes von Variablen

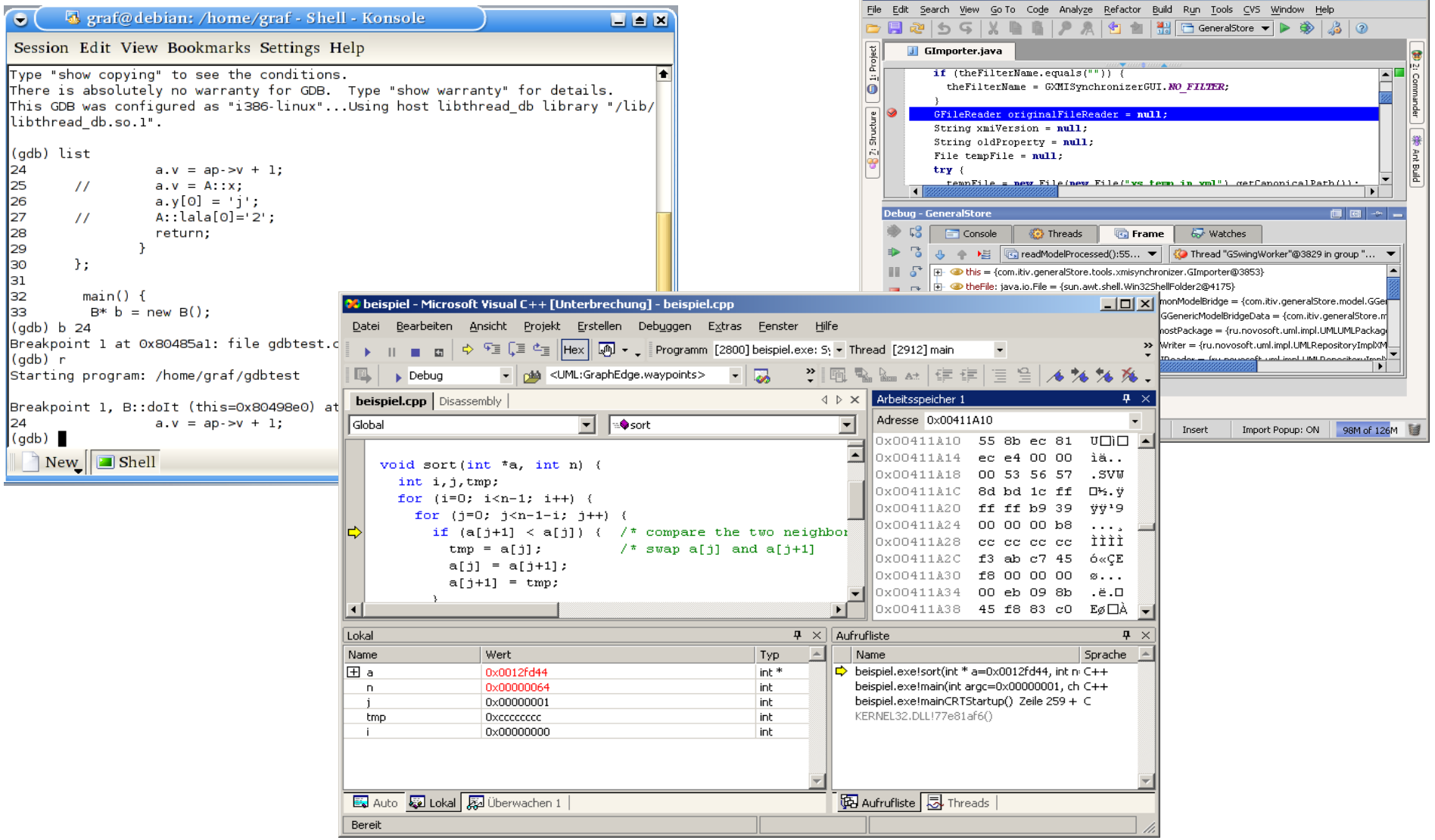
- Durchführung des Programms:
  - „Step by Step“
    - „Step into“
    - „Step over“



Beispiel: Debugger



Klassisches Debugging - Werkzeuge



Tracebasierte Ansätze

- Klassisch: Stoppen des Programms und Inspektion des Zustands
- Probleme
  - Wann und von wem wurde Zustand (Variable, Register...) verändert?
  - Zu weites Durchlaufen des Programms bedingt Neustart
  - Eingebettete Systeme: Peripherie und Umgebung läuft weiter
- Trace
  - Aufzeichnen des Programmablaufs
  - Rekonstruktion des Systemzustands in Vergangenheit möglich
- Vorteile
  - Keine Unterbrechung des Programmablaufs
  - „Rückwärts“-Debugging
  - Untersuchung des Trace auch Offline möglich, „Post-Mortem“

Tracebasierte Ansätze

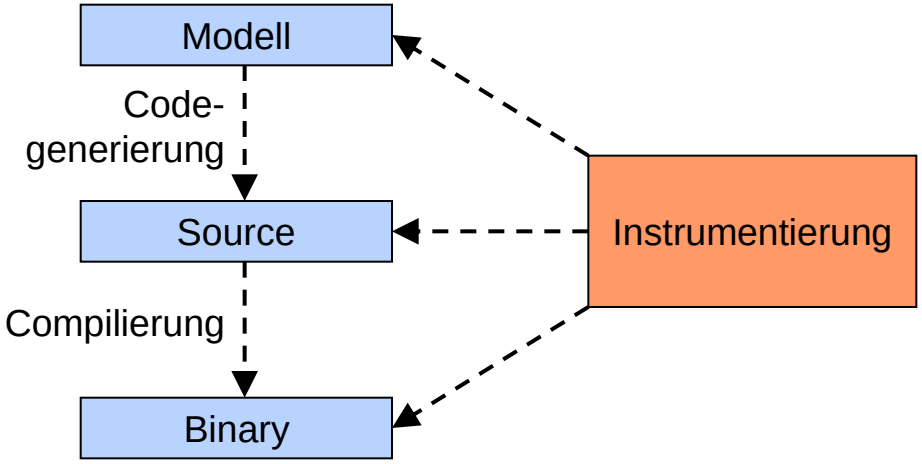
- Dedizierte Hardware

iSystem iC4000 ActiveEmulator

  - Emuliert CPU
  - Überwacht Pins
  - Nutzt On-Chip Schnittstellen (NEXUS 5001, BDM, JTAG)

+ Echtzeitfähig
+ Keine Beeinflussung des Programms
- Oft sehr teuer
- Bandbreite endlich

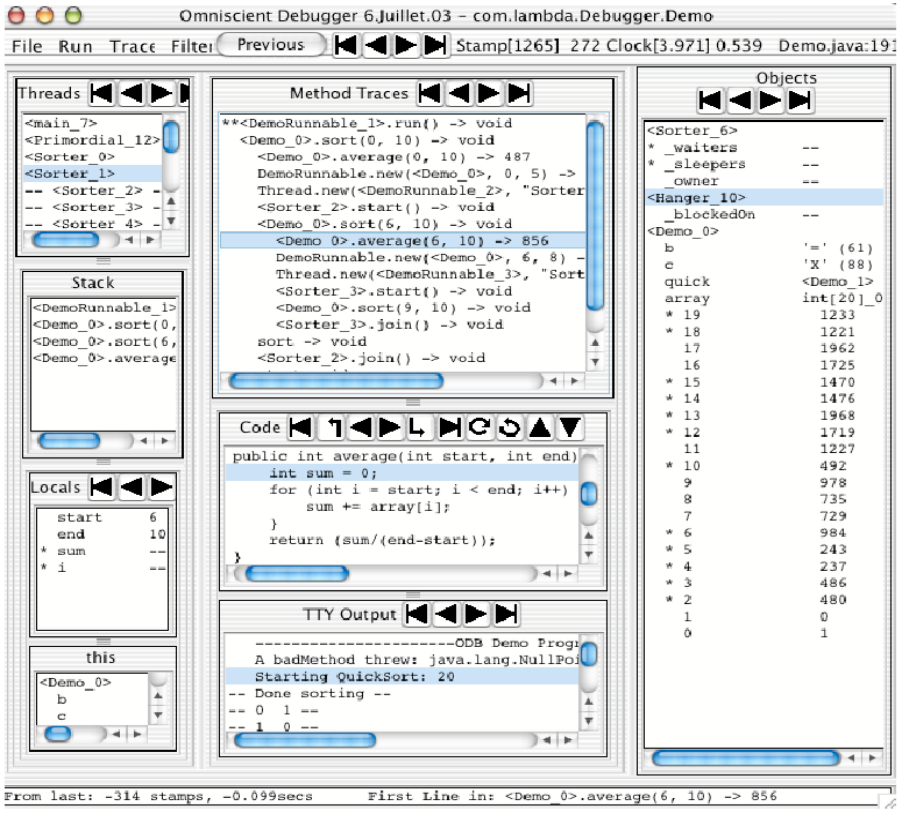
- Instrumentierung

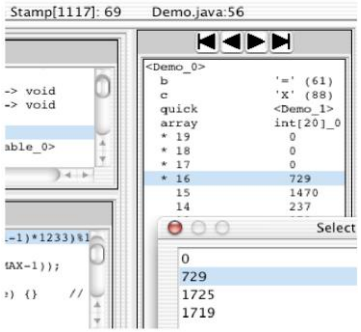


  - Modell, Source oder Binary
  - Statisch oder dynamisch

+ Beliebige Aquisition von Kontrollfluss und Daten
- Overhead Speicher und Performanz
- Veränderung des Programms („Heisenberg“-Effekt)

Omniscient Debugger (B. Lewis)





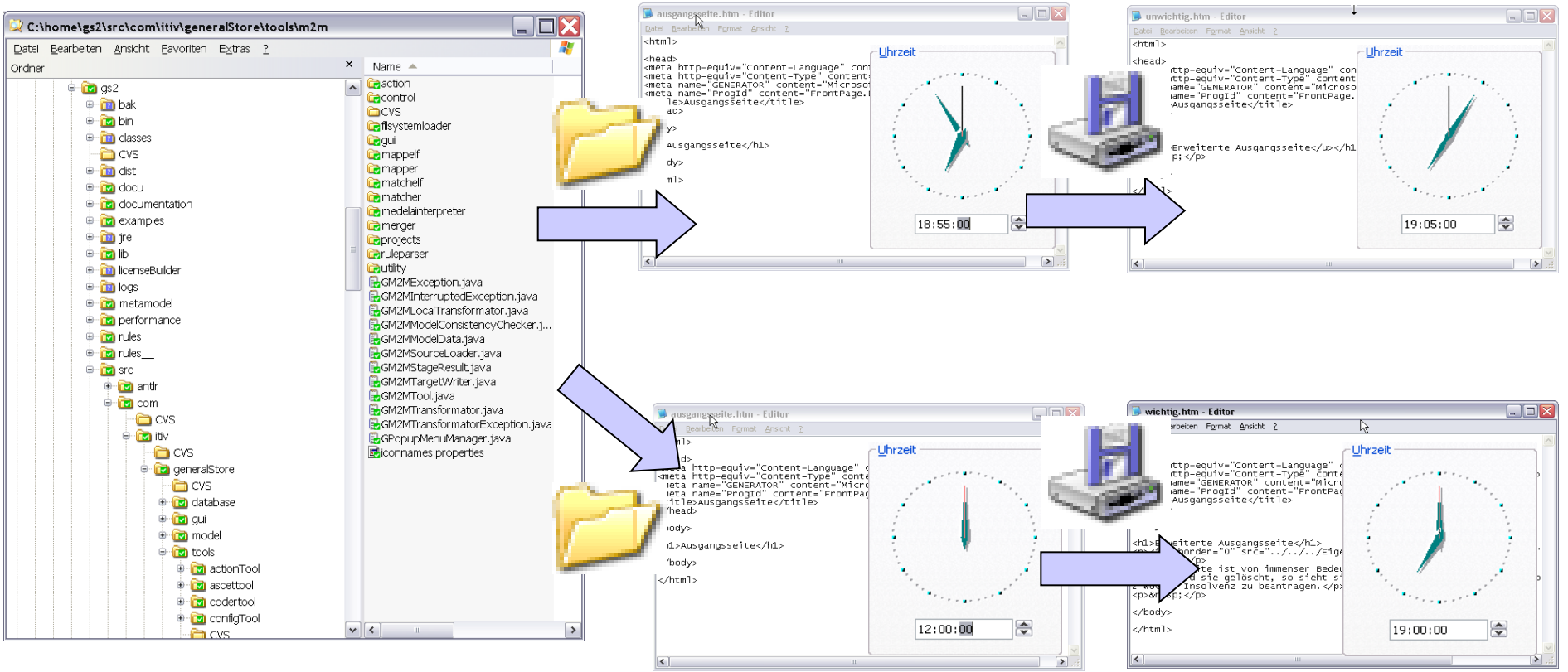
- Konzept: **Rückwärts in der Zeit**
- Eigener *class-loader* instrumentiert JAVA-Bytecode
- Protokolliert
  - Kontrollfluss
  - Variablenänderungen
  - Umschalten zwischen Threads
- Präsentation verknüpft
  - Abfolge der Methodenaufrufe
  - Quelltextansicht
  - Überwachungsfenster
  - Ausgabe
- Abfolge der Werte einer Variable und Springen zum Zeitpunkt der Änderung

7.1.2 Quellcode Versionsverwaltung

- Concurrent Versions System (CVS)
- Aegis
- Arch
- BitKeeper
- CMSynergy
- Co-Op
- Darcs
- GIT
- Monotone
- OpenCM
- Perforce
- PureCM
- Subversion
- Superversion
- svk
- Vesta
- Visual SourceSafe
- usw.

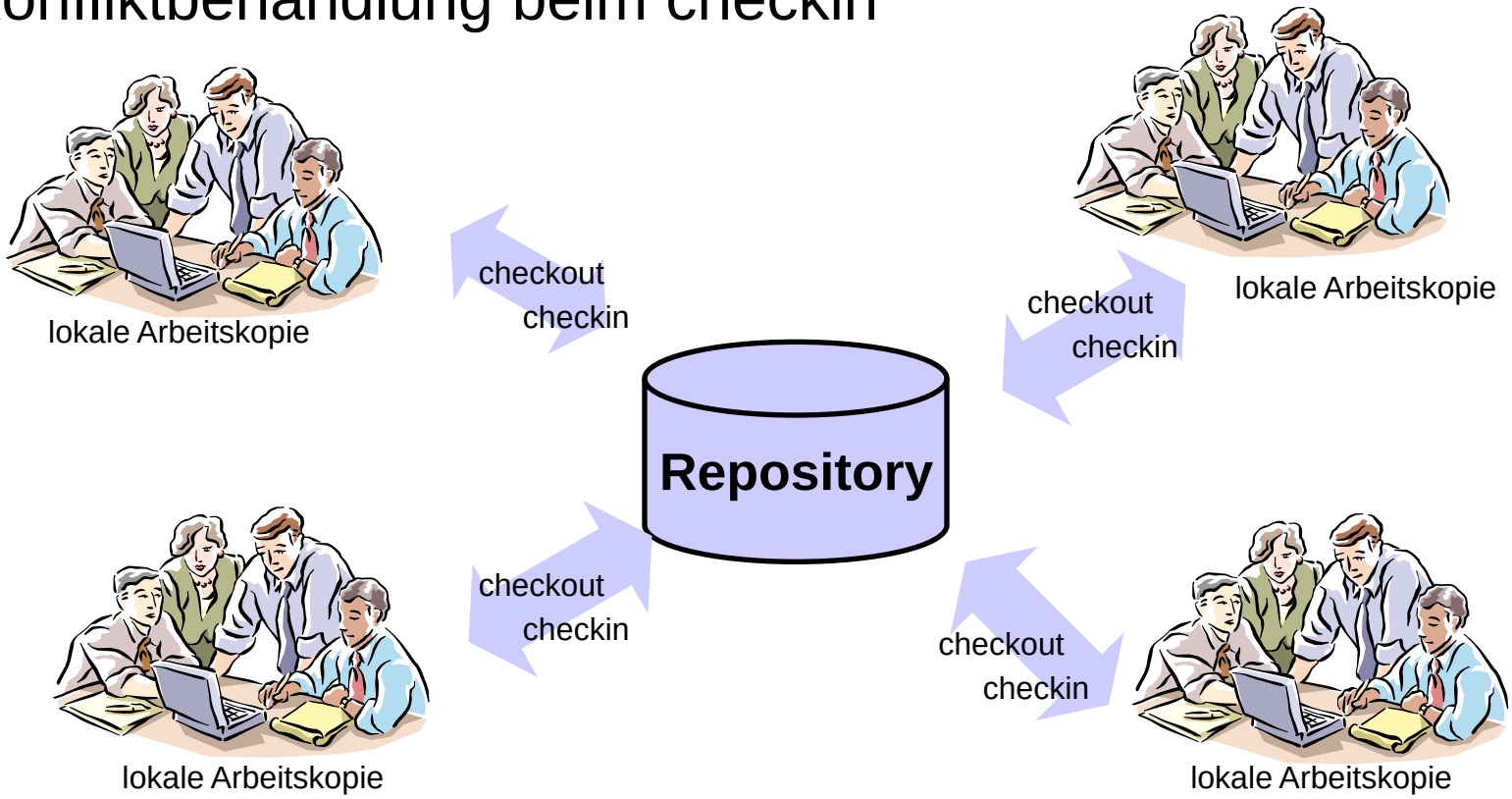
Wozu Quelltext Versionsverwaltung?

- Koordination im Team
- Nachvollziehbarkeit
- Parallele Entwicklung mehrerer Versionen
- Disziplin



Der Ansatz von CVS (Concurrent Versions System)

- Zentrales Repository
- Beliebige Versionen abrufbar
- Jeder Entwickler hat eine lokale Arbeitskopie (checkout)
- Parallele Änderungen möglich (concurrent), Konfliktbehandlung beim checkin



Merging

- Der Befehl `cvs update` bewirkt ein Versionsabgleich zwischen Repository und Workspace. Datei für Datei und Zeile für Zeile werden Änderungen ermittelt und „gemerged“:

	Repository	Workspace
Fall A		
Fall B		
Fall C		
Fall D		

Merging (Update)

	Repository	Workspace
Fall A		
Fall B		
Fall C		
Fall D		

Fall A:

- Nix passiert

Fall B:

- Update meldet, dass eine Änderung im Workspace vorgenommen wurde

Fall C:

- Kopie im Workspace wird durch Version im Repository ersetzt

Falll D:

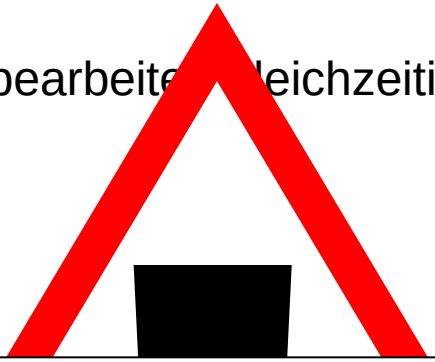
- Drei Dateiversionen werden zusammengeführt
  - Ursprüngliches Repository Checkout
  - Neue Version im lokalen Workspace
  - (geänderte) Version im Repository
- Sie werden durch „Three Way Merge“ zusammengeführt

Benutzung von CVS Checkout, Commit, Update

- Checkout
  - Erstellt eine private Kopie der Dokumente innerhalb des lokalen Arbeitsverzeichnisses.
  - Die Position ist beliebig
  - Mehrere lokale Kopien mit unterschiedlichen Versionen sind möglich
- Commit
  - Übermittelt am Ende der Arbeit die Änderungen an den Dokumenten an den Server
  - Die lokalen Kopien müssen die aktuellen Versionen sein
- Update
  - Vergleicht die lokalen Kopien mit dem Repository und aktualisiert sie bei Bedarf

Three-Way-Merge

- Geht meistens gut, außer:
  - Teammitglieder A und B bearbeiten gleichzeitig das gleiche Programm.

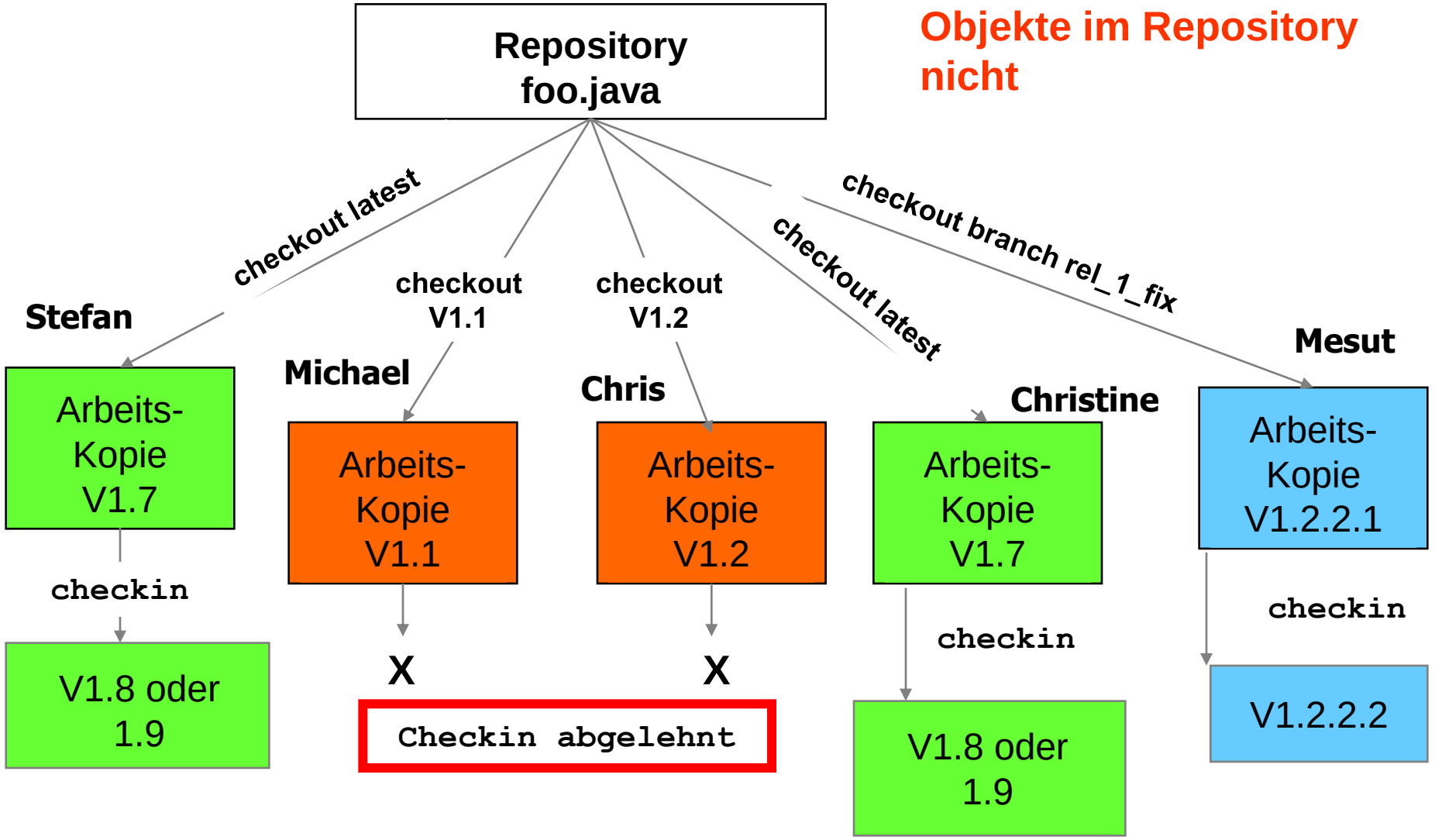


Problem: Wenn zwischen Update und Commit eine größere Zeitspanne liegt, besteht die Gefahr, dass ein anderer während dessen ebenfalls updatet und einen Commit durchführt, wodurch das Update des ersten Teilnehmers veraltet und das System in ein „Versionsdilemma“ gerät

Deshalb: Vor Commit ***IMMER*** Update durchführen

Gleichzeitiger Checkout

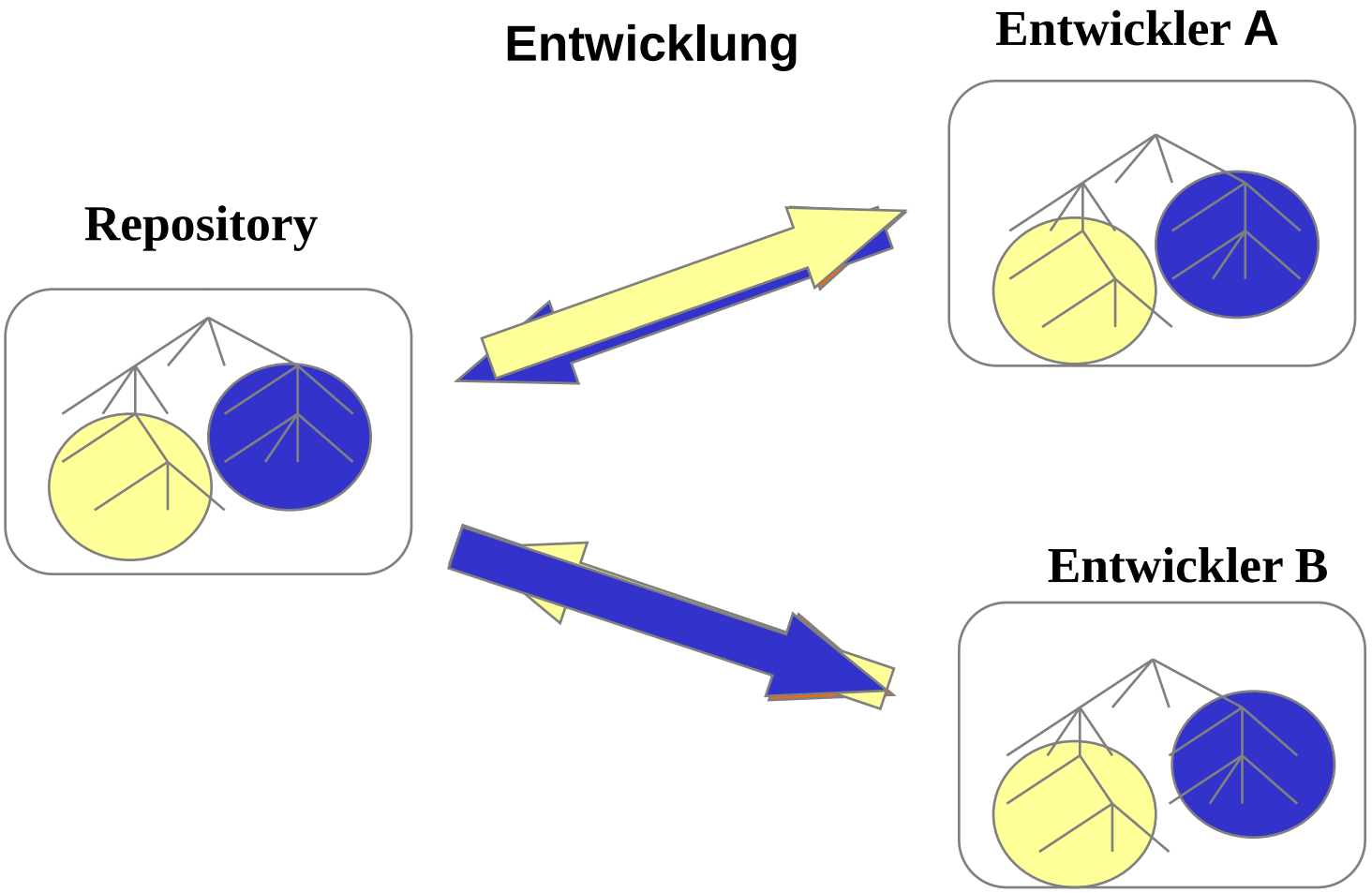
Checkout sperrt die Objekte im Repository nicht



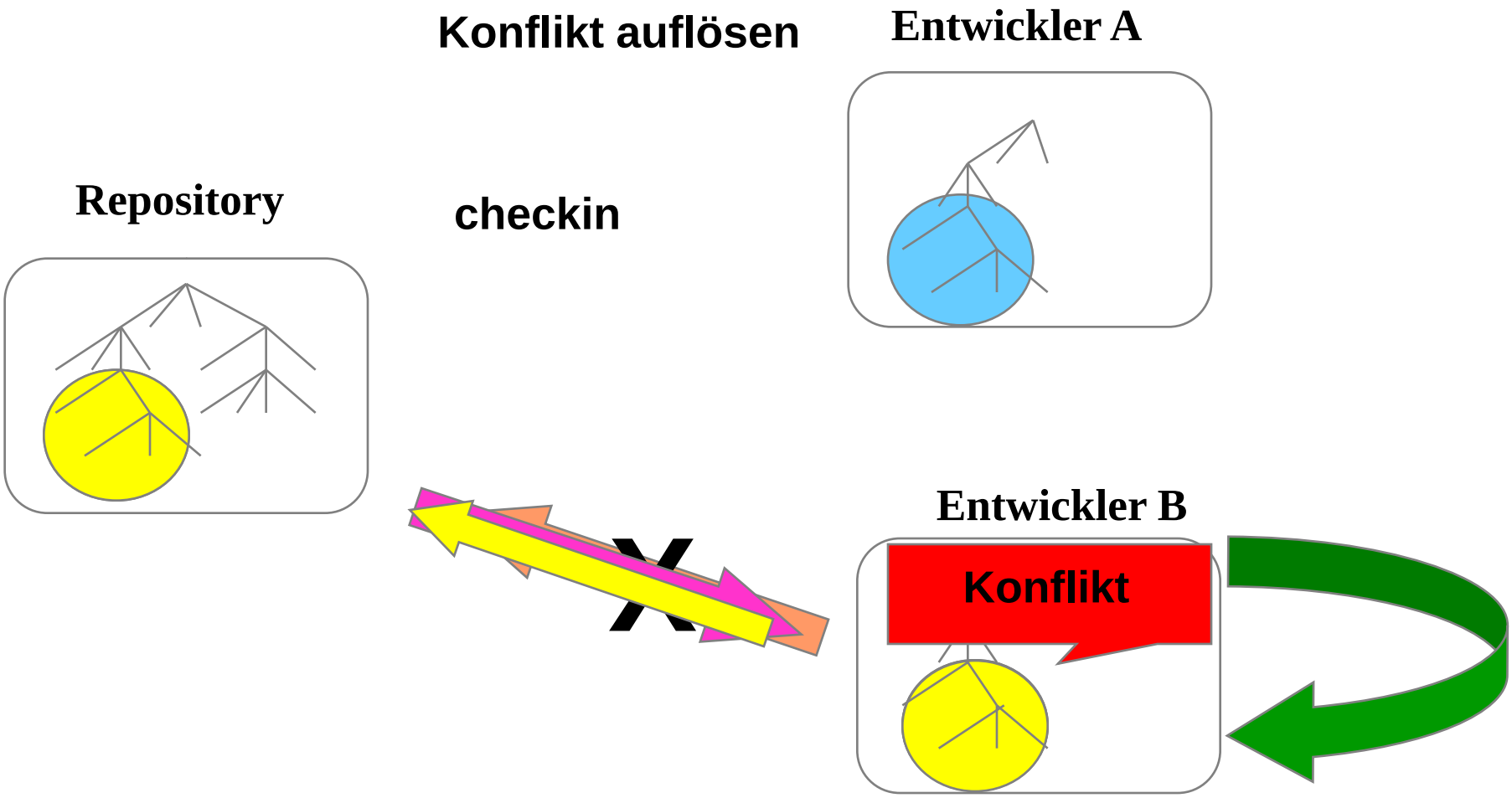
Der Entwicklungszyklus mit CVS

1. Dateien im lokalen Arbeitsverzeichnis “auschecken”
2. Dateien editieren
3. Testen (lokaler Build, Syntaktische Analyse, …)
4. Aktualisierung der lokalen Kopien mit **update**
   dabei werden die Änderungen der anderen Entwickler eingefügt (wenn nötig)
5. Nochmal testen, wenn in Schritt 4 gepatcht wurde
6. Änderungen übermitteln mit **commit**
7. Ab Schritt 2 wiederholen, bis zur nächsten Release
8. Release markieren (**tag**)
9. Modulnamen und Release-Tag zur System-Synthese übermitteln

Ideale Entwicklung mit CVS



Reale Entwicklung mit CVS



Einchecken

- Zum Markieren eines Status zur späteren Wiederherstellung
- Einchecken zusammenhängender Dateien mit einer einzelnen Operation
  → Dabei eine Log-Nachricht angeben
- Einchecken ist ein Backup
- In der Uni einchecken, um dann zu Hause schnell auschecken zu können
- Vorher immer ein Update

Richtlinien

- „Checke nur **kompilierbaren und getesteten Code** ein — die anderen werden es dir danken
- Mache **vor einem commit noch ein update** und achte auf zu lösende Konflikte
- Führe **regelmäßig commit** aus und gib **sinnvolle Log-Einträge** an

  (...)
- Denke immer daran, daß andere von deiner Arbeit abhängig sind“

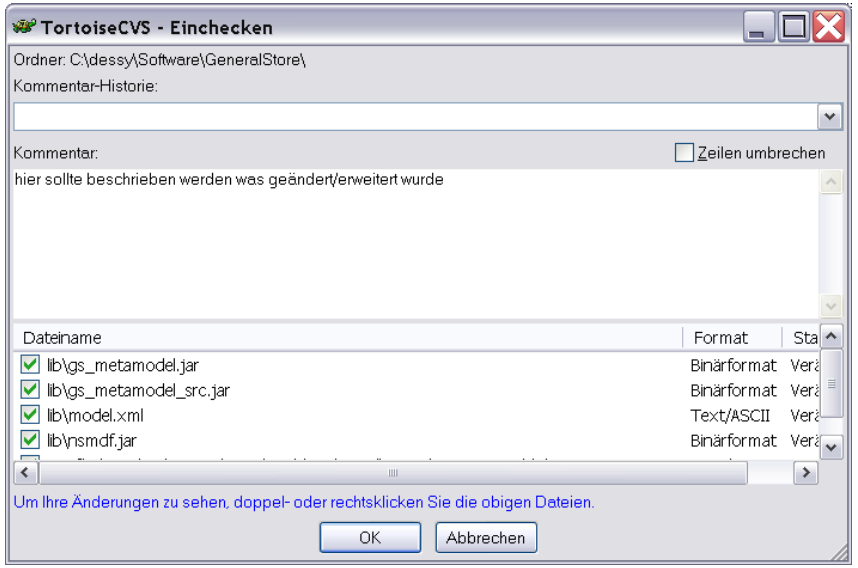
Die Schwächen von CVS

- Schlechter Umgang mit Verzeichnissen
- Schlechter Umgang mit Binärdateien
- Umbenennen von Dateien ist schwierig
- Keine Sicherheitsmechanismen
- Gewisse Features erfordern Disziplin

Weiterentwicklung:
Subversion (SVN)

Grafische Clients für CVS

- Gibt es in allen Größen und Formen
- Potenzielle Vorteile:
  - Übersichtliche Darstellung
  - Grafische Revisionsbäume
  - Grafische Aufbereitung von Änderungen
  - **Integration in IDE**
- Nachteil: Funktionsweise und Features stark unterschiedlich



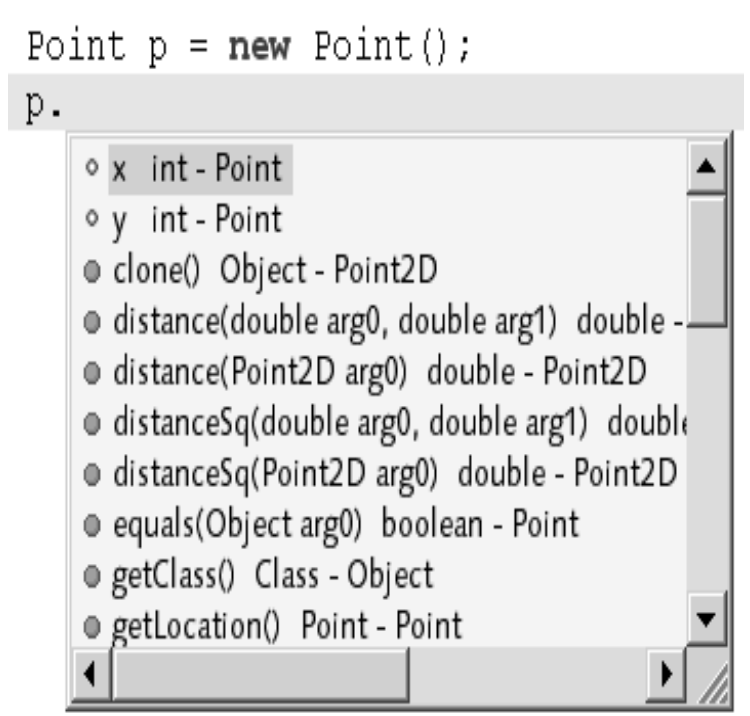
Fragen ?

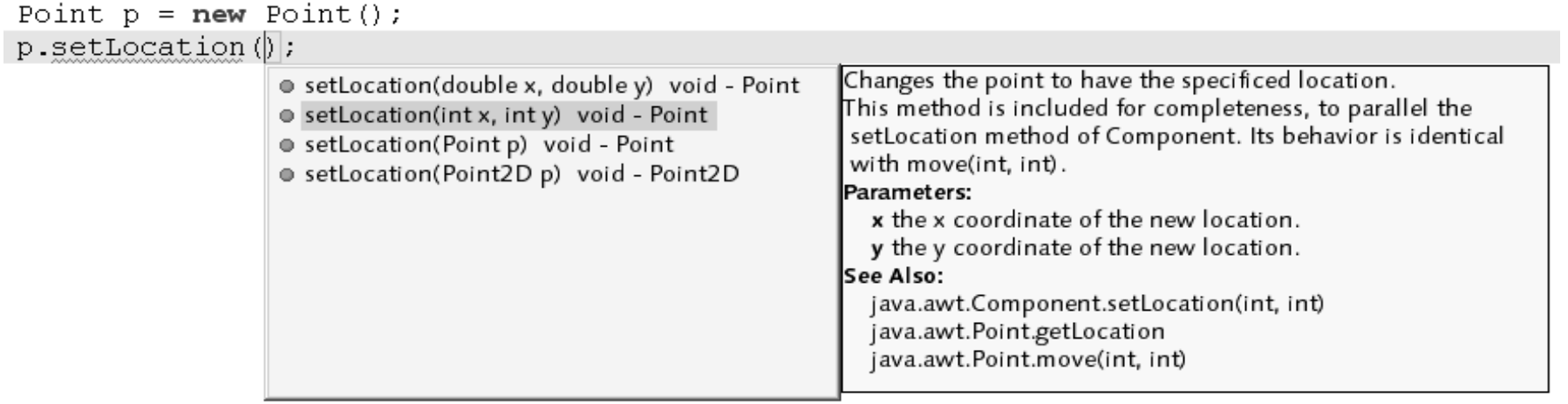
7.1.3 Integrated Development Environment (IDE)

- integrierte Entwicklungsumgebung zur Entwicklung von Software
- Komponenten:
  - Texteditor
  - Compiler bzw. Interpreter
  - Linker
  - Debugger
  - Quelltextformatierungsfunktion
  - Versionsverwaltung
  - Projektmanagement
  - UML-Modellierung
  - Erstellung von grafischen Benutzeroberflächen.
- Beispiele
  - Borland C++ Builder, IntelliJ IDEA (JAVA, .NET), Microsoft Visual Studio, Eclipse, KDevelop

Typische Features von IDEs (I)

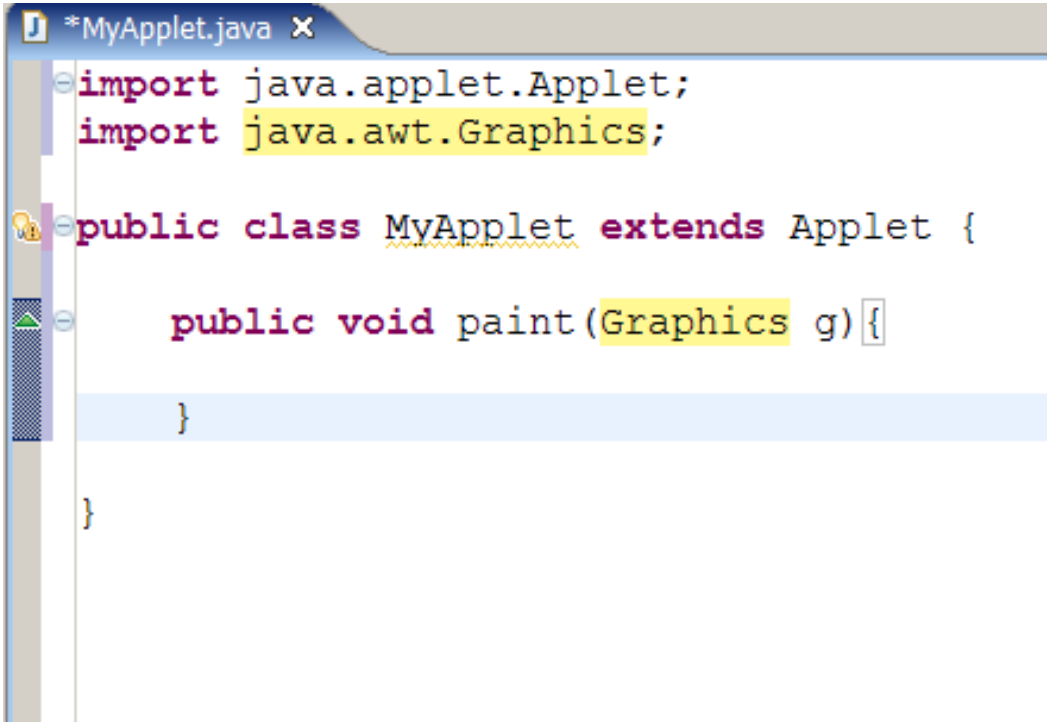
- Autovervollständigung





Typische Features von IDEs (II)

- Syntax Highlighting
- Autocompilierung
- Refactorings
- Suche
- Navigation durch den Quellcode
- Codegeneratoren
- Hervorhebungen



Beispiel: eclipse - Entwicklungsumgebung und mehr …

- Motivation
- Geschichte
- Architektur
  - Platform Runtime
  - Eclipse Platform
  - Java Development Tools (JDE)
  - Plugin Development Environment (PDE)
- Zusammenfassung

Motivation

- Was ist eclipse?
  - open source Entwicklungsumgebung
    - Deckt viele OS ab (Windows, Linux, Solaris,...)
    - Sprachneutral (Java, C, Cobol, …)
  - erweiterbare Plattform für die Werkzeugintegration
    - gesamter Softwarelebenszyklus abdeckbar
    - bringt Entwicklungsumgebung für Erweiterungen gleich mit
  - beinhaltet neue GUI für Java-Applikationen
  - Framework für Anwendungsentwicklung

Geschichte

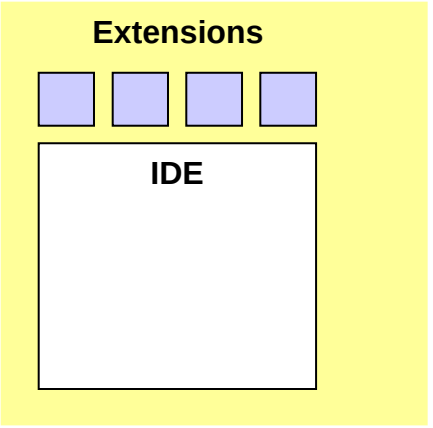
- Als kommerzielle Version von OTI + IBM geplant
- Entwicklung ab April 1999
- 2001 Version 1.06 an open source -Gemeinde übergeben
- aktuell: 3.1.1
- Kommunikationsplattform: http://www.eclipse.org
- große + sehr aktive Community
- Eclipse foundation: IBM, Borland, BEA, Intel, HP, SAP, ...

Entwicklungsumgebungen

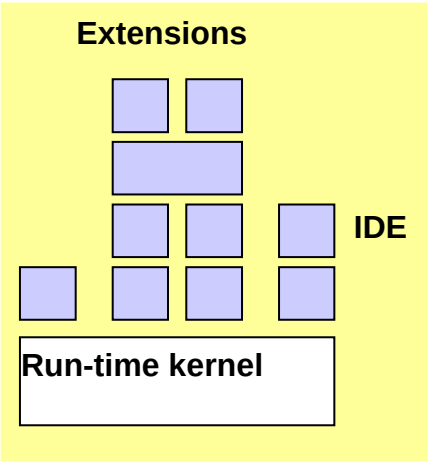
- Welche IDEs gibt es noch?
  - JBuilder, NetBeans, IntelliJ IDEA, WSAD, …
- Wann bzw. warum setzt man IDEs anstatt einfacher Editoren ein?
  - große Projekte
  - heute Software benötigt, die gesamten Software-lebenszyklus möglichst nahtlos unterstützt
    - Installation, Einarbeitung, Datenaustausch, Teamwork, ... erleichtert
    - Anpassbarkeit
  - Refactoring, Debugging, Code-Schablonen, Syntaxcheck, Code Completion, Hovering

Architektur (I)

- Übliche IDE-Architektur
  - monolithisch => Erweiterungen nur wie vorgesehen
  - Erweiterungen wirken oft fremd



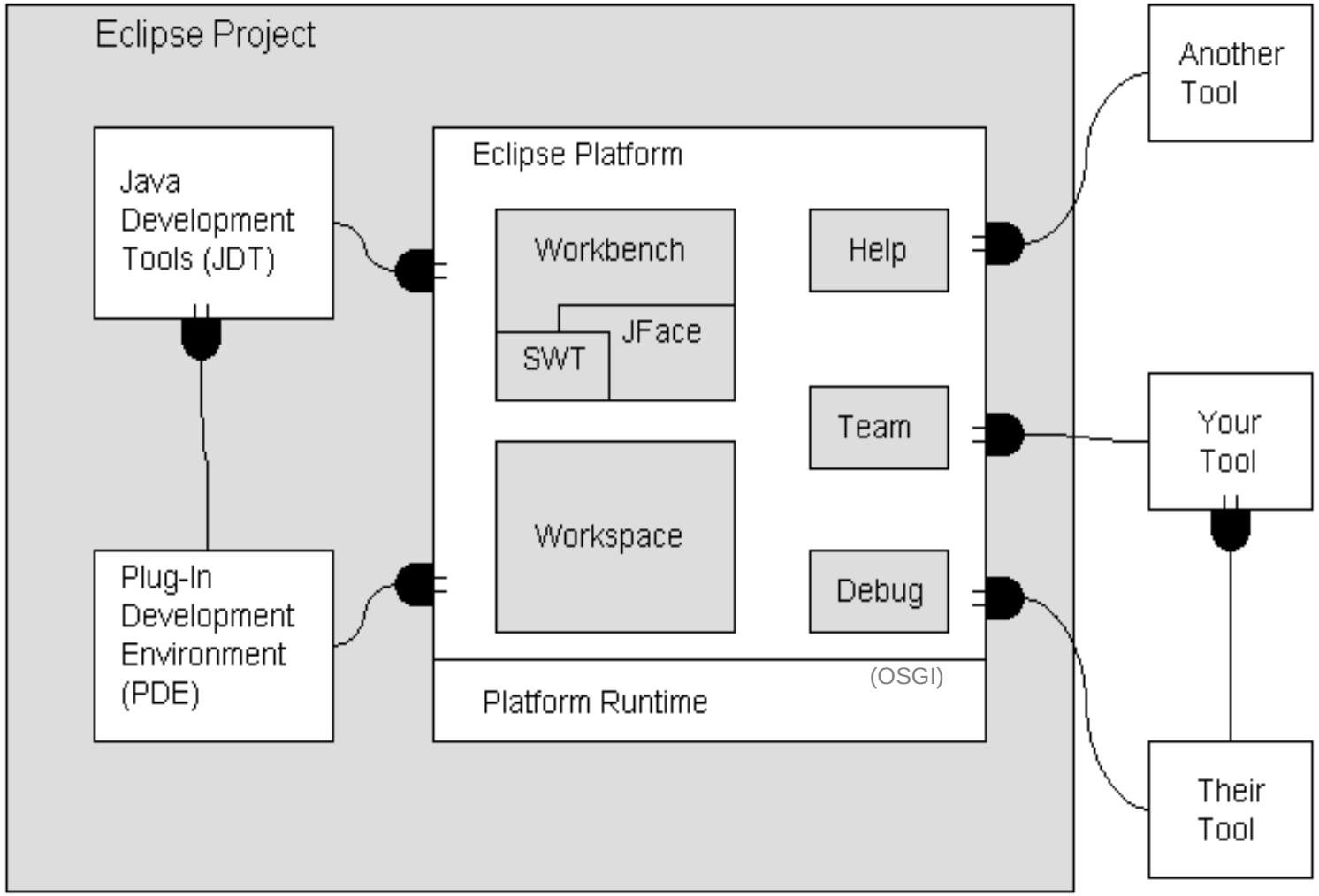
- Eclipse:
  - Bestandteile: Plugins + Platform Runtime
  - Plugins nutzen Plugins
  - Endanwender richten eigene Umgebung ein (Installieren + Deinstallieren Plugins)
  - Erweiterung Teil der Philosophie



Architektur (II)

- Plugins:
  - zwingend: Plugin-Manifest (plugin.xml)
    - Deklaration
    - beschreibt Konfiguration des Plugins
    - beschreibt Integration des Plugins in die Plattform, d.h. welche Erweiterungspunkte genutzt, welche bereitgestellt
  - optional: Java-Archiv
    - Implementierung
  - optional: Resourcen
    - Bilder
    - Hilfetexte
    - ...

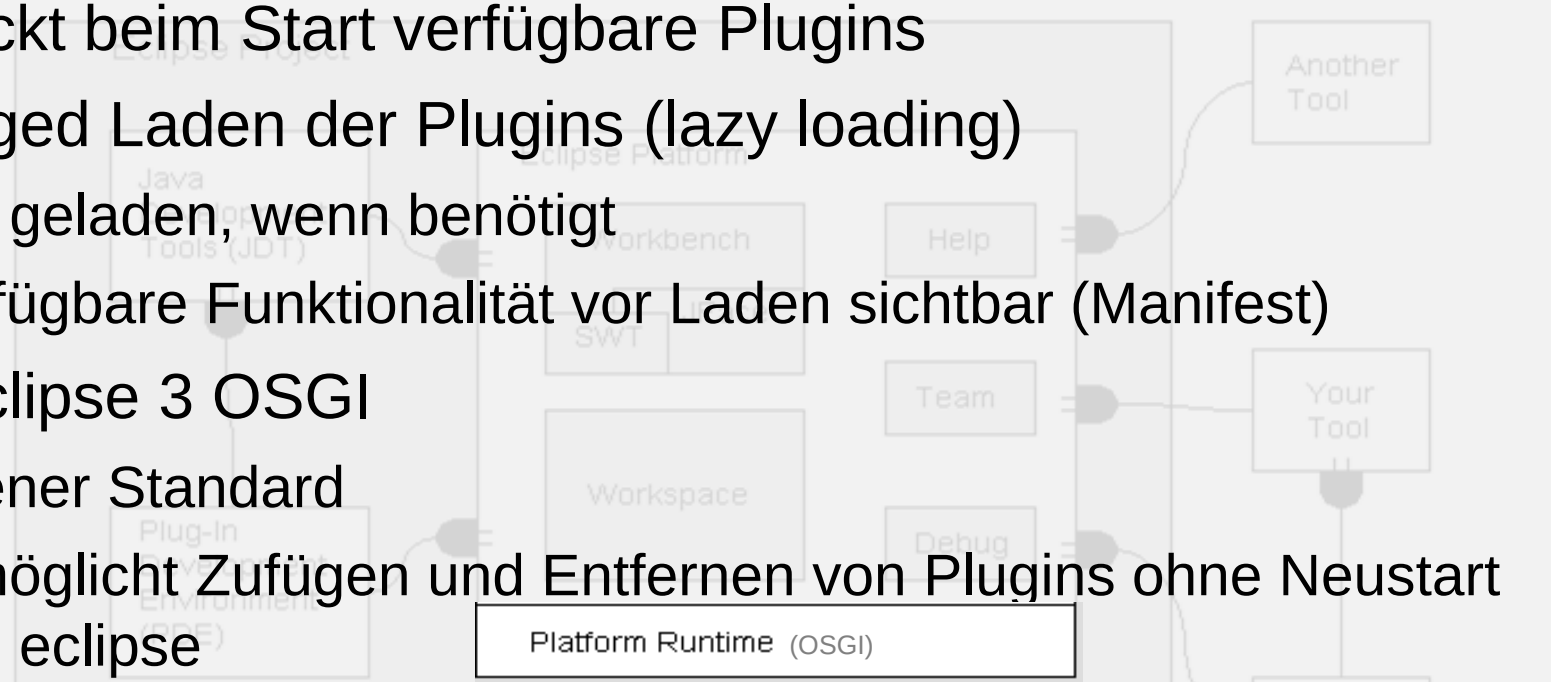
Architektur (III)



SWT: Standard Widget Toolkit

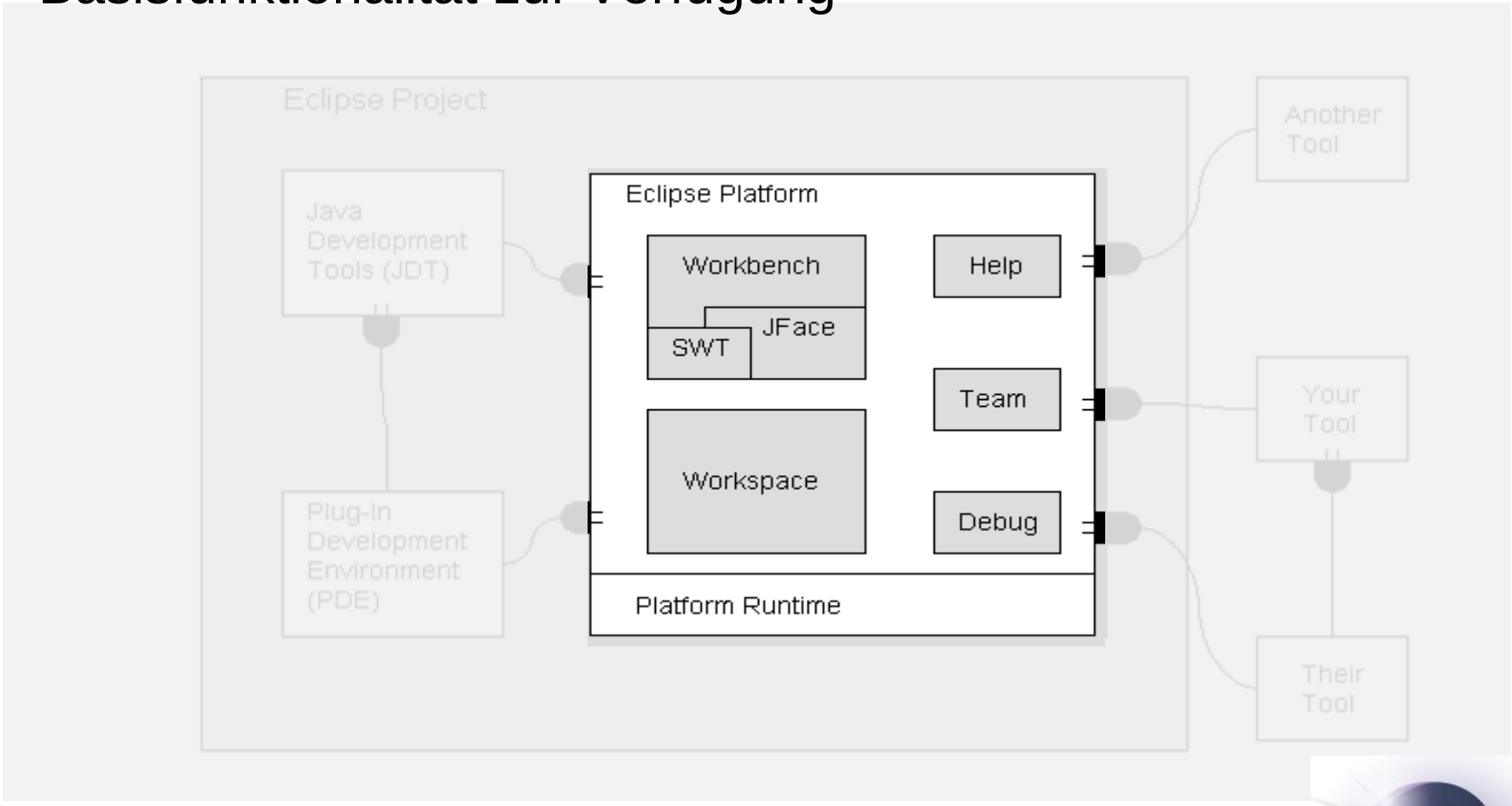
Platform Runtime

- Kern, kein Plugin (aber formal als Plugin angelegt)
- definiert Plugin Infrastruktur
- entdeckt beim Start verfügbare Plugins
- managed Laden der Plugins (lazy loading)
  - nur geladen, wenn benötigt
  - verfügbare Funktionalität vor Laden sichtbar (Manifest)
- seit eclipse 3 OSGI
  - offener Standard
  - ermöglicht Zufügen und Entfernen von Plugins ohne Neustart von eclipse



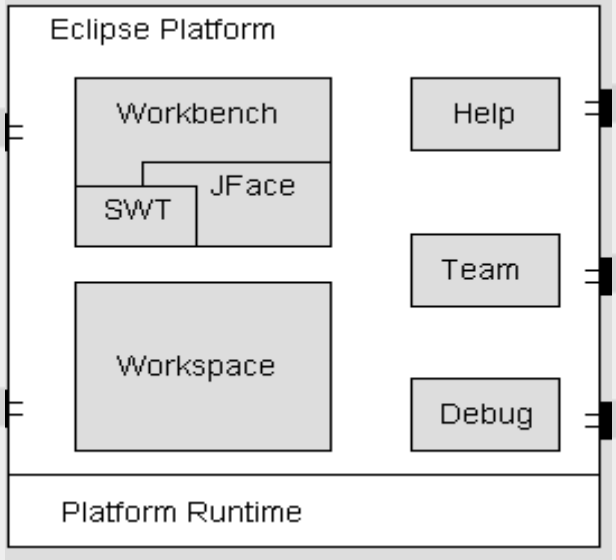
Eclipse Platform

- Kernkomponenten, stellen domänen-spezifische Basisfunktionalität zur Verfügung



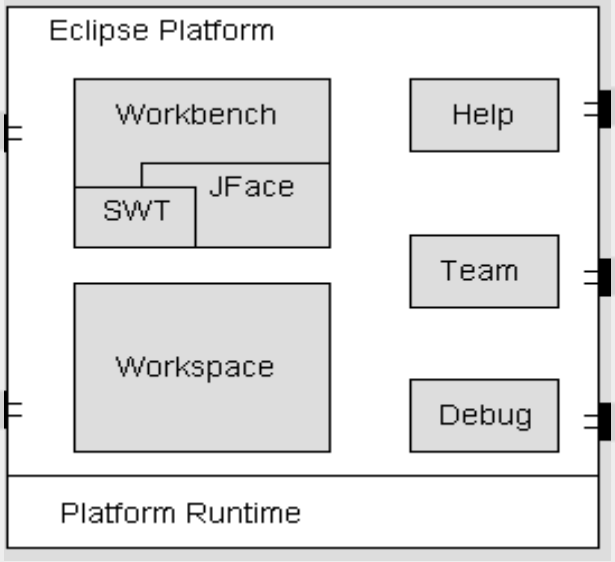
Eclipse Platform - Workspace

- Workspace
  - besteht aus Projekten (jeweils auf ein Verzeichnis im Workspace abbildbar)
  - alle Werkzeuge arbeiten mit Ressourcen aus Workspace des Nutzers

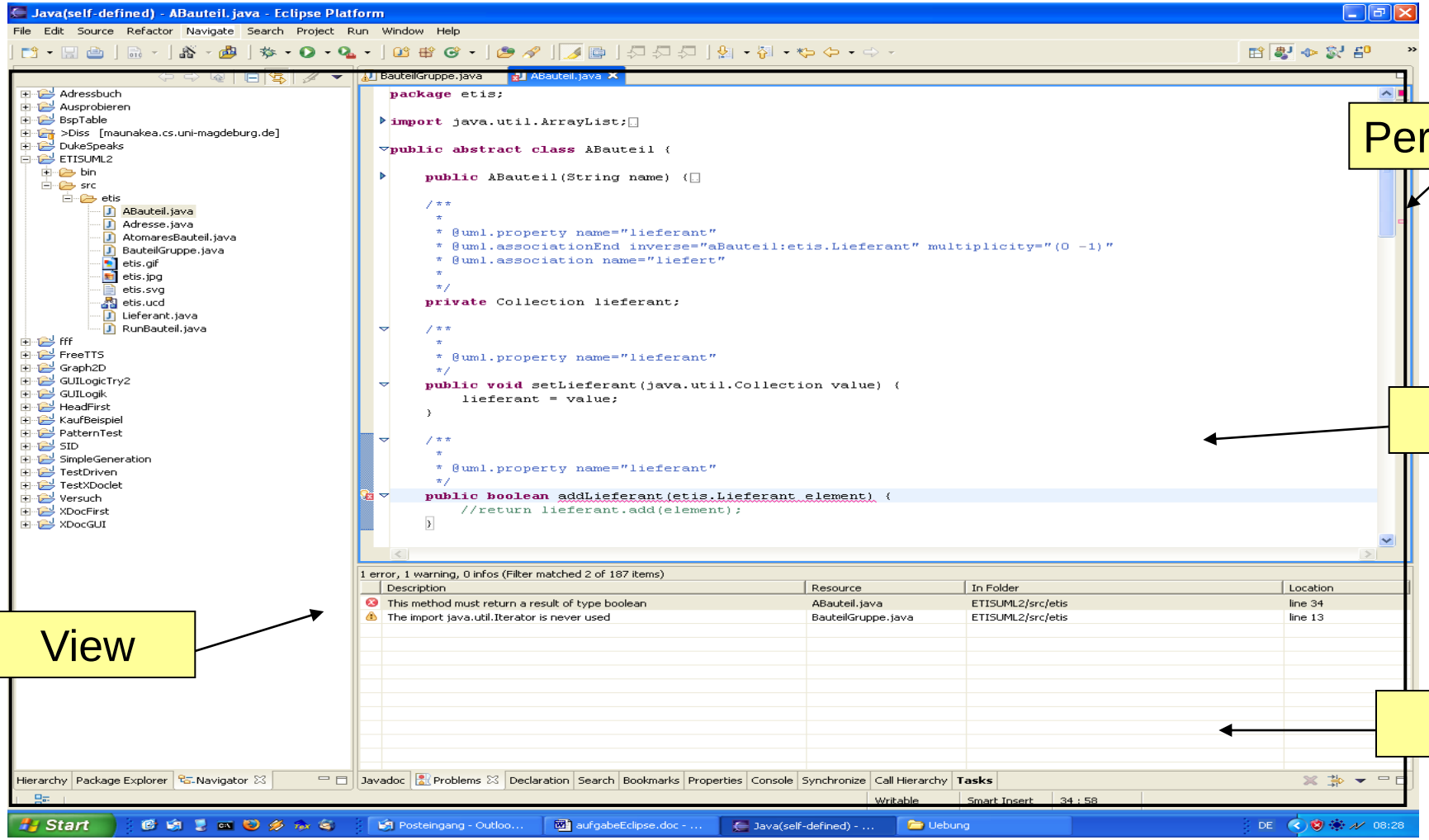


Eclipse Platform - Workbench

- Workbench
  - Implementiert BNO
  - Zusammenspiel von Editoren, Sichten, Perspektiven

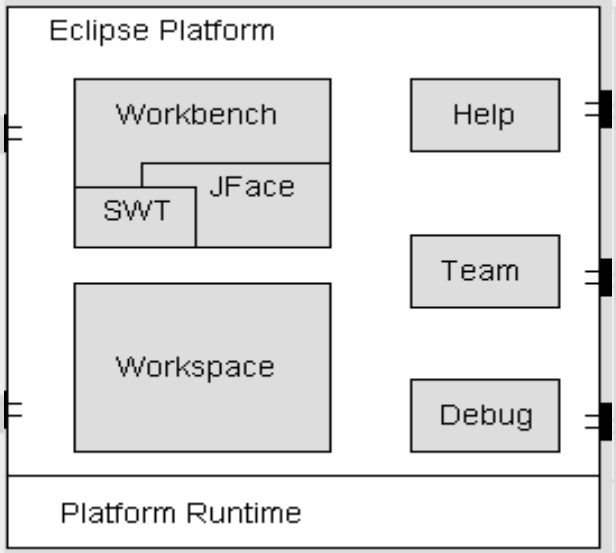


Workbench (3.0.1)

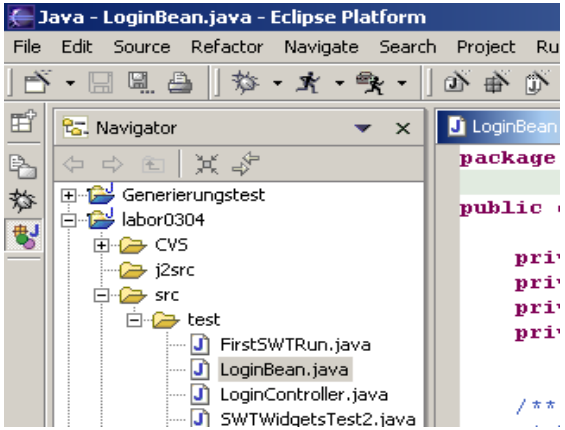


Eclipse Platform - SWT

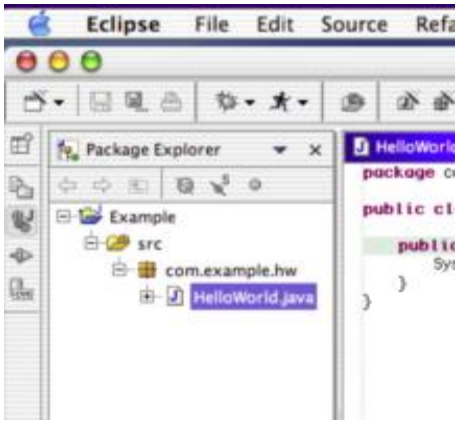
- SWT (Standard Widget Toolkit)
  - Bereitstellung GUI-Komponenten (Button, Trees, ...)
  - OS-unabhängige API
  - nutzt plattform-eigene Widgets oder emuliert diese



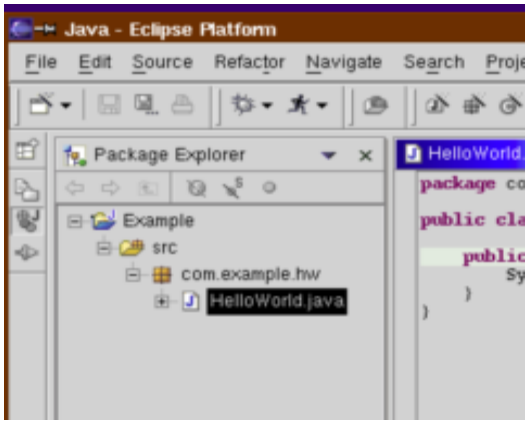
Eclipse Platform - SWT



Eclipse auf Windows XP



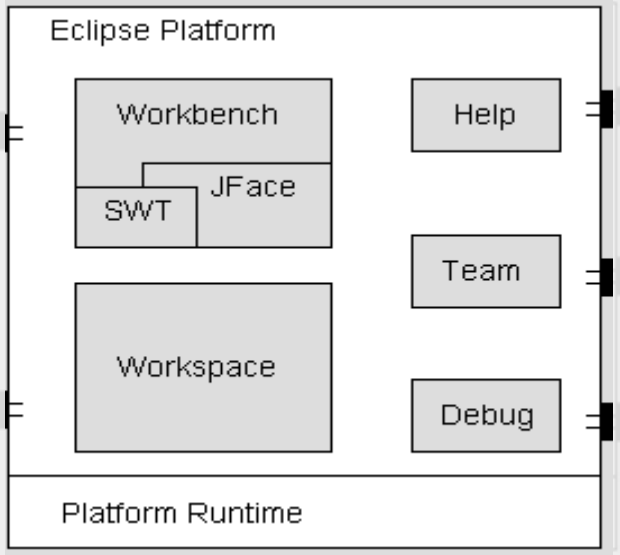
Eclipse auf Mac OS X (Carbon)



Eclipse auf Linux (Motif)

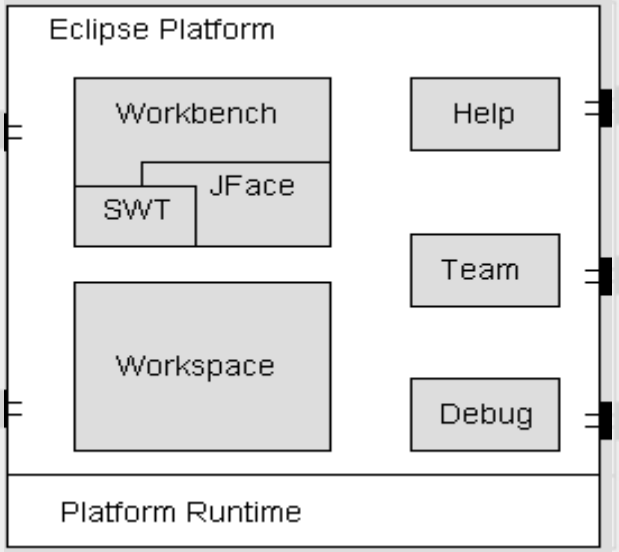
Eclipse Platform - JFace

- JFace
  - Framework zur Gestaltung grafischer Oberflächen (windowsystem unabhängig)
  - Trennung von Modell und Darstellung
  - bettet SWT in Eclipse-Framework

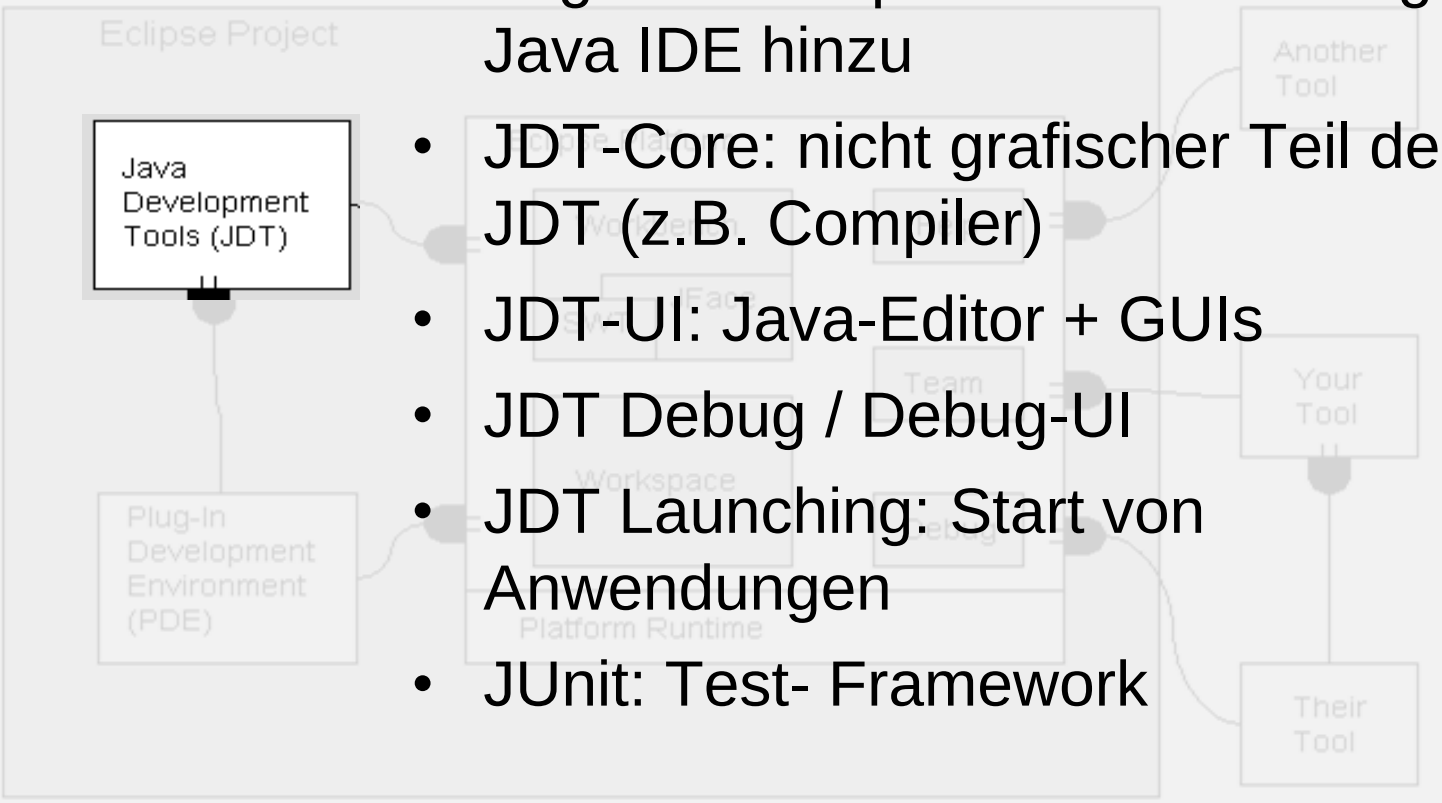


Eclipse Plattform - weitere Komponenten

- Help
  - Als Webserver implementiert, HTML-Hilfeseiten
- Team
  - Basis für Versions- und Konfigurations-management
  - gleichzeitige Nutzung von Ressourcen innerhalb Team
- Debug

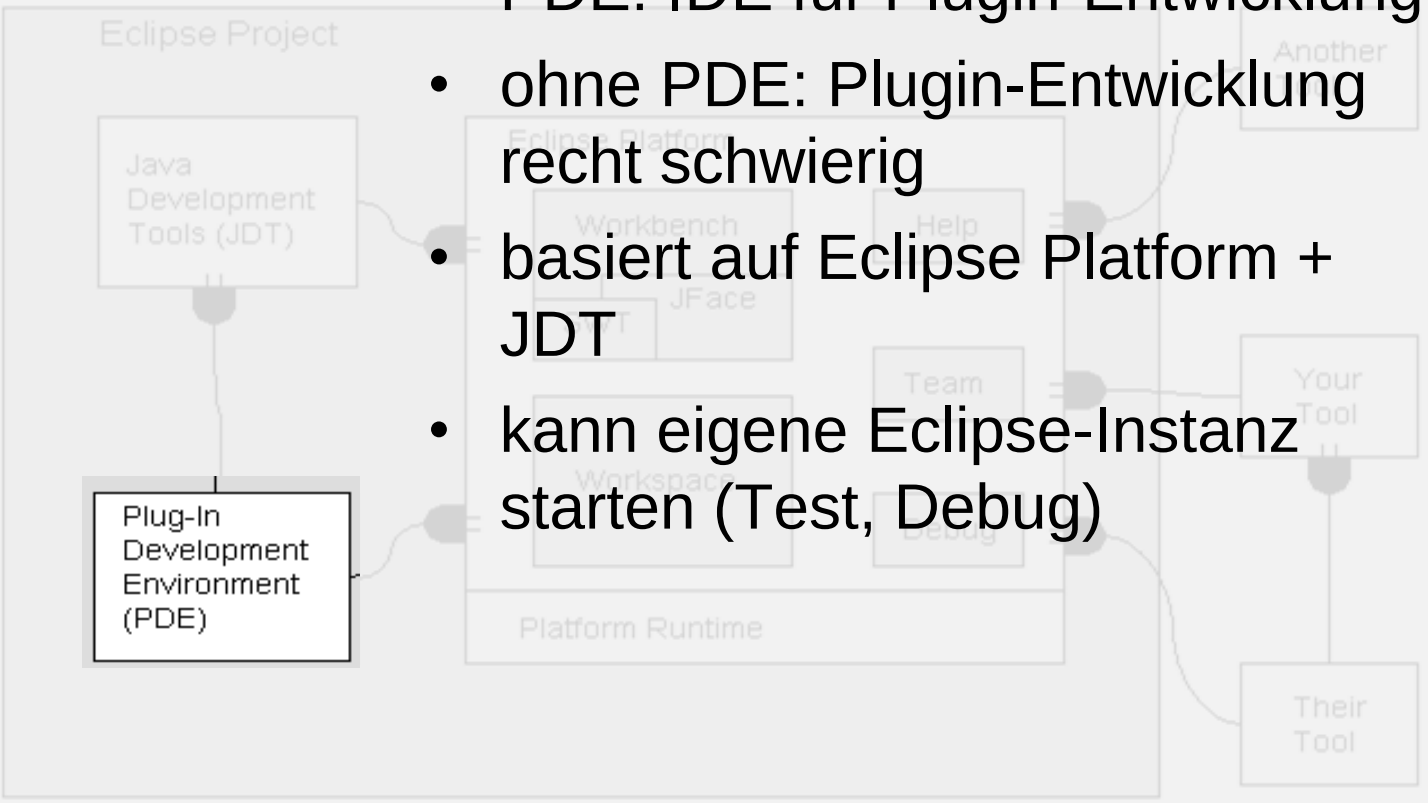


Java Development Tools (JDT)



- Fügen zu Eclipse eine vollständige Java IDE hinzu
- JDT-Core: nicht grafischer Teil des JDT (z.B. Compiler)
- JDT-UI: Java-Editor + GUIs
- JDT Debug / Debug-UI
- JDT Launching: Start von Anwendungen
- JUnit: Test- Framework

Plugin Development Environment (PDE)



- PDE: IDE für Plugin-Entwicklung
- ohne PDE: Plugin-Entwicklung recht schwierig
- basiert auf Eclipse Platform + JDT
- kann eigene Eclipse-Instanz starten (Test, Debug)

Zusammenfassung: eclipse

- open source Entwicklungsumgebung
  - Nachteil: Standardeditoren müssen z.T. als Plugins nachgerüstet werden (JSP, XML)
- Plattform für Werkzeugintegration
  - Anpassbarkeit + Erweiterbarkeit durch wieder-verwendbare Komponenten
- Werkzeug für schnelle effiziente Werkzeugimplementierung
  - Große Community z.Z. > 800 Plugins
- beinhaltet neue GUI für Java-Applikationen
- Eclipse auch als Application Framework nutzbar

Literatur zu eclipse

- Backschat, M., Edlich, J2EE-Entwicklung mit Open-Source-Tools, Spektrum Akademischer Verlag, München, 2004
- http://www.eclipse.org/eclipse/presentation/eclipse-slides_files/frame.htm
- Eclipse Homepage: www.eclipse.org
- Eclipse Plattform Technical Overview. Object Technology International, Inc., 02/2003
- Markus Weyerhäuser: Die Programmierumgebung Eclipse. JAVASpektrum, 02/2003
- Gamma, E., Beck, K., Contributing to eclipse, Addison-Wesley, Bosten, 2004
- Daum, B., Java-Entwicklung mit Eclipse 3, dpunkt.verlag, Heidelberg, 2005

Fragen ?

7.2.1 Programmierrichtlinien


```
7       if (_pinNames2PortNumber == null) {
8         _pinNames2PortNumber = new HashMap();
9       }
10            if (_pinNames2PortNumber.containsKey(thePinName)) {
11              portNumber = (String) _pinNames2PortNumber.get(thePinName);
12            } else if (thePinName.equals("get/get")) {
13        _pinNames2PortNumber.put(thePinName, "1");
14        portNumber = (String) _pinNames2PortNumber.get(thePinName);
15              } else if (thePinName.equals("set/aValue")) {
16                /*_pinNames2PortNumber.put(thePinName, "1");
17                portNumber = (String) _pinNames2PortNumber.get(thePinName); */
18      } else if (thePinName.equals("getAt/getAt")) {
19        _pinNames2PortNumber.put(thePinName, "1");
20        portNumber = (String) _pinNames2PortNumber.get(thePinName);
21    } else if (functionalElement.getAscetModelElement().toString().indexOf("ComplexModelElement"
22      {
23    com::itiv::generalStore::tools::ascettool::ascet12::blockdiagram::MBlockDiagramElementPin pi
24        if (pin.isInput()) {
25          if (_block2ListOfInputPins == null) {
26            _block2ListOfInputPins = new HashMap();
27          }
28  /*      if ("C".equals(functionalElement.getAscetModelElement().getCodeComponent().getComponent
29          com::itiv::generalStore::tools::ascettool::ascet12::database::MCodeComponent cc1 = (co
30          if ("C".equals(cc1.getComponentType().toString())) {
31            if (_block2ListOfInputPins.containsKey(theMSDElement)) {
32            java::util::List pinNameList = (java::util::List) _block2ListOfInputPins.get(theMSDE
33              if (pinNameList.indexOf(thePinName) != -1) {
34                pinPos = pinNameList.indexOf(thePinName);
35                pinPos = pinPos + 2;
36                portNumber = "" + pinPos;
37              } else {
38                pinNameList.add(thePinName);
```

Warum Programmier-Richtlinen?

- Verbessert die Lesbarkeit und deshalb die Wartbarkeit von Quellcode
- Beschleunigt Einarbeitung bei Personalwechsel und Wiedereinarbeitung,
- Erleichtert das gemeinsame Arbeiten von verschiedenen Programmierern an einem Projekt
- Wenn der Quellcode als Produkt ausgeliefert wird sollte dieser genauso verpackt und aufgeräumt sein wie jedes andere Produkt
- Zeitersparnis bei Fehlerfindung, Erweiterung und Pflege des Programms

→ Aber: Nicht übertreiben! Alle Regeln mit Augenmaß anwenden.

Was wird festgelegt?

- Formatierung (Layout)
  - z.B.

    for(i=0;i<grenze;i++)foo(i);
    for (i=0; i<grenze; i++) foo(i);
- Namens-Konventionen
  - Z.B. Methoden, die ein Attribut lesen oder schreiben, beginnen mit *hole* oder *get* bzw. *setze* oder *set* Beispiel: *getName*, *holeName*
- Implementierungs-Dokumentation
  - Vorspann
  - Klassen Kurzbeschreibung
  - Parameter Beschreibung
  - Vorbedingungen

→ wird von IDE unterstützt

Beispiel: Einrückung in Schleifen

K&R (Kernighan & Ritchie) Stil, oder „kernel style“ (Unix Kernel)

```
if (<Bedingung>) {
    <Rumpf>
}
```

„Whitesmiths Stil“

```
if (<Bedingung>)
    {
    <Rumpf>
    }
```

„Allman Stil“ nach Eric Allman, auch „BSD Stil“

```
if (<Bedingung>)
{
    <Rumpf>
}
```

„GNU Stil“, nach GNU Emacs

```
if (<Bedingung>)
  {
    <Rumpf>
  }
```

Examples of Coding Guidelines

- Java: geosoft.no/development/javastyle.html
- Java: www.horstmann.com/bigj/style.html
- Java: developer.netscape.com/docs/technote/java/codestyle.html
- C++: geosoft.no/development/cppstyle.html
- C++: oss.software.ibm.com/icu/userguide/conventions.html
- C++/Java: www.agsrhichome.bnl.gov/Controls/doc/codingGuidelines/codingGuidelines.html
- .NET: msdn.microsoft.com/library/default.asp?url=/library/enus/cpgenref/html/cpconnetframeworkdesignguidelines.asp
- PERL and shell-level: dev.panopticsearch.com/portable_coding.html
- PERL: www.lug.lk/~anu/misc/perl_coding_guidelines.txt

7.2.2 JavaDoc

- JavaDoc ist ein Programm, das aus Java-Quelltexten Dokumentationen im HTML-Format erstellt.
- Es befindet sich im Ordner /bin des sdk´s

Kommentare

Es gibt zwei Arten von Kommentaren:

- einzeilige Kommentare

```
// der Kommentar geht bis zum Ende der Zeile
```

- mehrzeilige Kommentare

```
/* Der Kommentar beginnt mit /*
und endet mit
*/
```

Beispiel


```
/**
 * Methodenbeschreibung - Methodenbeschreibung - Methodenbeschreibung -
 * Methodenbeschreibung - Methodenbeschreibung - Methodenbeschreibung -
 * Methodenbeschreibung - Methodenbeschreibung - Methodenbeschreibung
 *
 * @return boolean
 * @param text1 - String der ausgegeben wird.
 * @param text2 - String dessen Länge überprüft wird.
 * @throws Exception
 * @see TestKlasse#printPicture
 * @deprecated Bitte verwenden sie nur noch Sys.out.println(String text).
 */
public boolean printText(String text1, String text2) throws Exception {
      System.out.println(text1);
      if (text2.length()>=5) {
        throw new Exception();
     }
      return true;
}
```

Dokumentations-Kommentare


```
/**
 * beginnen immer mit /**
 * und enden mit */
```

• werden vor Klassen-, Variablen- und Methoden-Definitionen von dem Tool „javadoc“ interpretiert und zum Erstellen einer Dokumentation in Form von HTML-Dateien benutzt.

• * , Leerzeichen und Tabulatorzeichen am Zeilenanfang werden ignoriert. HTML Text kann beliebig eingesetzt werden. Spezielle tags beginnen mit „@“ und werden besonders interpretiert.

• Der erste Satz enthält die Kurzbeschreibung. Der Rest die ausführliche Beschreibung.

• Können HTML-Tags außer <H1>..<H6> und <HR> enthalten.

@deprecated

- @deprecated
  - @**deprecated** `deprecated-text`
  - vor allen Definitionen
  - Das mit dem deprecated-tag gekennzeichnete Objekt wird in einer späteren Version des Programmes möglicherweise nicht mehr enthalten sein. In dem Text kann angegeben werden, welches andere Objekt die Funktion übernimmt. Falls es keinen Ersatz gibt, sollte „No replacement“ angegeben werden.
  - Beispiel:

    @deprecated wird ersetzt durch Equation.f

    ==>

    Deprecated. wird ersetzt durch Equation.fs

@exception, @throws

- @exception, @throws
  - **@exception** `class-name description`
  - vor Methoden-Definitionen
  - In der Methode wird die Ausnahme „class-name“ nach außen weitergereicht.
  - Beispiel:

    @exception Keine Ausnahmen

    ==>

    Throws:

    Keine - Ausnahmen

@param

- @param
  - @**param** `parameter-name description`
  - vor Methoden-Definitionen
  - Beschreibt die Parameter einer Methode. Als Beschreibung kann beliebiger Text zur Erläuterung der Bedeutung und der Funktionsweise des Parameters angegeben werden.
  - Beispiel:

    @param args Enthält die Parameter, die beim Start des Programmes übergeben werden.

    ==>

    Parameters:

    - args - Enthält die Parameter, die beim Start des Programmes übergeben werden.

@return

- @return
  - @**return** `description`
  - vor Methoden-Definitionen
  - Beschreibt den return-Wert einer Methode. Als Beschreibung sollten Typ und Wertebereich des return-Wertes angegeben werden.
  - Beispiel:

    @return float - Nullstelle der Funktion f.

    ==>

    Returns:

    - float - Nullstelle der Funktion f.

@see

- @see
  - **@see** `name label`
  - vor allen Definitionen
  - Erzeugt eine Referenz auf ein anderes Objekt. Mögliche Formen sind:
    - @see package.class#member label
      - Referenz auf ein anderes Objekt
    - @see label
      - Referenz auf ein URL
    - @see "string“
      - allgemeine Referenz, z.B. auf ein Buch oder Artikel
  - Beispiel:

    @see Hello#main main

    ==>

    See Also:

    main

@link

- @link
  - {@**link** `name label`}
  - vor allen Definitionen
  - Kann genauso verwendet werden wie das see-tag, nur daß die Referenz im Text erzeugt wird und nicht als extra Absatz.
  - Beispiel:

    Zur Berechnung der Nullstelle nutze man die Methode {@link equation#newton newton} der Klasse {@link equation equation}.

    ==>

    Zur Berechnung der Nullstelle nutze man die Methode newton der Klasse equation

@since

- @since
  - @**since** `since-text`
  - vor allen Definitionen
  - Erläutert seit wann das entsprechende Objekt verfügbar ist.
  - Beispiel:

    @since Version 1.1.6

    ==>

    Since:

    Version 1.1.6

@version

- @version
  - **@version** `version-text`
  - vor Klassen-Definitionen
  - Gibt die Version der Klasse an.
  - version-tags werden übernommen, wenn
    - javadoc -version
    - benutzt wird.
  - Beispiel:

    @version 1.0

    ==>

    Version:

    – 1.0

Tags

@author *name-text*
@deprecated *deprecated-text*
@exception *class-name description*
{@link *name label***}**
@param *parameter-name description*
@return *description*
@see *reference*
@since *since-text*
@serial *field-description*
@serialField *field-name field-type field-description*
@serialData *data-description*
@throws *class-name description*
@version *version-text*

Zusammenfassung

tag	package	class interface	field	method constructor	javadoc
author		x			-author
deprecated	x	x	x	x	
exception				x	
link	x	x	x	x	
param				x	
return				x	
see	x	x	x	x	
since	x	x	x	x	
serial			x		
serialField			x		
serialData				x	
throws				x	
version		x			-version

Dokumentations- Generierung

1. Befehlszeilen-Kommando:

```
javadoc [options] [packagenames] [sourcefile] [classnames]
```

```
javadoc *.java
```

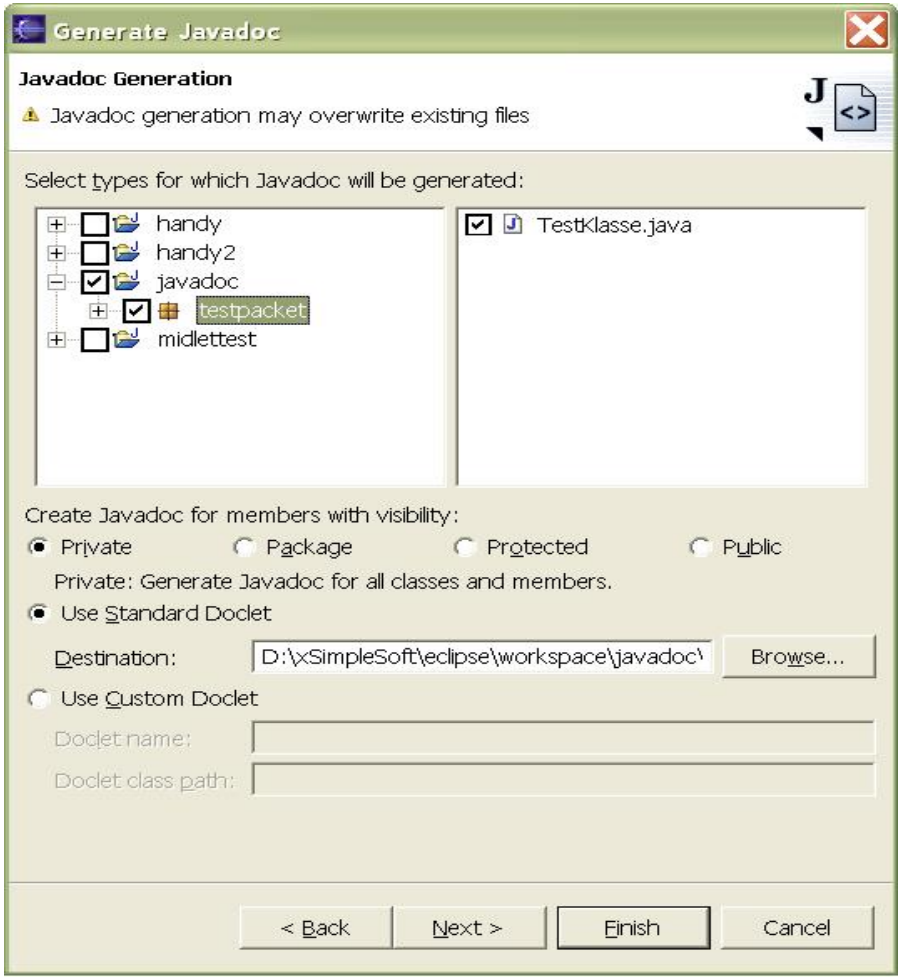
Erzeugt eine Html-Dokumentation aller java Dateien des aktuellen Verzeichnisses.

-public	Elemente des Typs public werden dokumentiert.
-protected	Elemente des Typs public und protected werden dokumentiert (das ist die Voreinstellung)
-package	Elemente des Typs package, public und protected werden dokumentiert.
-private	Alle Elemente werden dokumentiert.
-doclet	Alternatives Doclet.
-classpath	Gibt die Liste der Pfade zur Suche von Klassendateien an.

Dokumentations- Generierung

2. Eclipse:

Durch Auswahl des Menüpunktes „**Datei -> Export -> JavaDoc**“ öffnet sich folgendes Fenster:



Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification

This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.2.

See:

Description

Java 2 Platform Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional

All Classes

ClassDoc
ConstructorDoc
Doc
DocErrorReporter
Doclet
ExecutableMemberDoc
FieldDoc
MemberDoc
MethodDoc
PackageDoc
Parameter
ParamTag
ProgramElementDoc
RootDoc
SeeTag
SerialFieldTag
Tag
ThrowsTag
Type

com.sun.javadoc

Class Doclet

```
java.lang.Object
  |
  +--com.sun.javadoc.Doclet
```

public abstract class **Doclet**
extends java.lang.Object

This class documents the entry-point methods in a Doclet. It may be used as the superclass of a doclet but this is not required.

Constructor Summary

Doclet()

Method Summary

static int	**optionLength**(java.lang.String option) Check for doclet added options here.
static boolean	**start**(RootDoc root) Generate documentation here.
static boolean	**validOptions**(java.lang.String[][] options, DocErrorReporter reporter) Check that options have the correct arguments here.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Fragen ?