



JUnit (Version 4.x)

- Framework zur Unterstützung von Unit-Tests
 - Automatisierte Ausführung von Tests
- Wir verwenden nicht JUnit 3.x
 - Erfordert anderes Programmierkonzept
- Ideen dahinter
 - Testgetriebene Entwicklung: Erst testen, dann programmieren
 - Alle Testfälle häufig ausführen (nach jeder Programmänderung)
 - Jeder Testfall testet nur eine bestimmte Funktion
- Test-Methode
 - Parameterlose Methode, mit Annotation `@Test`
 - Vor und nach jedem Test werden `@Before` bzw. `@After` – Methoden aufgerufen
 - Assertionen sind statische Methoden in der Klasse `junit.framework.Assert`

JUnit



- Mögliche Ergebnisse eines Testfalles
 - Kein Fehler
 - Failure
 - Vom Tester vorausgesehener Fehlerfall
 - `junit.framework.AssertionFailedError`
 - Error
 - Unerwarteter Fehlerfall
 - Zu testendes Programm wirft eine andere Exception oder einen Error
 - Oder: Programm terminiert nicht
 - Eclipse unterscheidet Failure und Error automatisch
 - JUnit 4 nicht → Stack-Trace der Exceptions beachten
- JUnit-Testfälle für MicroJava
 - Derzeit 181 Test-Methoden
 - Für eigene Fortschrittskontrolle und für die Tutoren
 - Failures führen zu Punkteabzügen
 - **Übungen mit Errors werden nicht korrigiert (= 0 Punkte)**

JUnit-Testfälle

- Basisklasse `CompilerTestCase`
 - Initialisiert alle Compiler-Klassen
 - Zu testender Quellcode wird als String übergeben
 - Verwaltet die erwarteten Ergebnisse
 - Erwartete Fehlermeldungen: `expectError(...)`
 - Erwartete Tokens: `expectToken(...)`
 - Erwartete Symboltabelle: `expectSymTab(...)`
 - Erwarteter Bytecode: `expectCode(...)`
 - Ruft den Parser bzw. Scanner auf
 - Vergleicht die tatsächlichen Ergebnisse mit den erwarteten
 - Failure, wenn keine Übereinstimmung
 - Ausgabe auf Konsole

JUnit-Testfälle



- Gleiche Testfälle für alle Übungen
 - Testfälle definieren, ab wann welche Funktionen gefordert sind
 - Beispiel: `expectError(EX >= 4, ...)`
 - Diese Fehlermeldung wird erst ab der 4. Übung erwartet
 - Vor der 4. Übung kann der Fehler noch nicht überprüft werden
- Konfiguration
 - Klasse `ssw.mj.test.Configuration`
 - Feld `CURRENT_EXERCISE`
 - Aktuelle Übungsnummer
 - **Bei jeder neuen Übung anpassen**
 - Feld `PRINT_ALL_OUTPUT`
 - Debug-Ausgabe aller Ergebnisse

Tests Ausführen

- **Compilieren des Compilers und der Testfälle:**
 - > `javac -classpath .;junit.jar
 ssw/mj/*.java
 ssw/mj/symtab/*.java
 ssw/mj/codegen/*.java
 ssw/mj/test/*.java`
- **Ausführen aller Testfälle auf der Kommandozeile**
 - > `java -cp .;junit.jar org.junit.runner.JUnitCore
 ssw.mj.test.ScannerTest
 ssw.mj.test.ParserTest
 ssw.mj.test.ParserErrorTest
 ssw.mj.test.UEBei spi el eTest
 ssw.mj.test.ConcurrentTest`
- **Immer alle Tests ausführen!**

UE 2: Lexikalische Analyse (*Scanner*)



- MJ-Angabe.zip:
 - Compilerklassen (Token.java, Errors.java, Gerüst von Scanner.java, ...)
- MJ-Tests.zip
 - JUnit-Testfälle
- Abgabe
 - siehe Abgabeanleitung auf Homepage!
 - elektronisch bis Mi, 24.10.2007, 20:15
 - alle zum Ausführen benötigten Dateien
 - auf Papier
 - nur Scanner.java
 - Testabgabe durchführen
 - Beliebige, kleine zip-Datei abgeben

Versionsverwaltungssystem

- Verwaltung von Quelltexten und Dokumenten
 - Protokollierung der Änderungen
 - Wiederherstellung eines alten Standes
 - Archivierung von Releases
 - Koordination mehrerer Entwickler
 - Verwaltung paralleler Entwicklungszweige
- Pessimistic Revision Control
 - Auschecken und Sperren, Verändern, Einchecken
 - z.B. SourceSafe
- Optimistic Revision Control
 - Auschecken, Verändern, Zusammenführen
 - z.B. CVS, Subversion

Repository

- Zentraler Speicher
 - Projektdateien
 - Zeitstempel und Benutzer
 - Log-Meldungen
- Benutzer arbeitet auf lokaler Arbeitskopie
 - Auschecken kopiert Dateien aus dem Repository
 - Einchecken schreibt veränderte Dateien zurück
 - Update aktualisiert die Arbeitskopie
- Kompakte Speicherung
 - Änderungen zur letzten Revision
 - Delta / Reverse Delta
- Branching, Tagging



Subversion

- Open-Source-Versionsverwaltungssystem
 - Nachfolger von CVS
 - Versioniert Verzeichnisse und Umbenennungen
 - Atomares Commit
 - Client/Server-Architektur
 - Billiges Branching und Tagging
 - Effiziente Behandlung binärer Dateien
- Repository
 - FSFS
 - Berkeley DB
- <http://subversion.tigris.org/>

Wichtige Befehle

- Repository anlegen

```
> svnadmin create --fs-type fsfs /path/to/repos
```

- Importieren von Dateien

```
> svn import /tmp file:///path/to/repos -m "initial import"
```

- Durchsuchen

```
> svn ls file:///path/to/repos
```

- Auschecken (Check out)

```
> svn co file:///path/to/repos project
```

- Einchecken (Commit)

```
> svn ci -m "changed some files"
```

- Aktualisieren auf letzte Revision (Update)

```
> svn up
```

Tools

- TortoiseSVN
 - Erweiterung für den Windows Explorer
 - <http://tortoisesvn.tigris.org/>
- Subclipse
 - Plugin für Eclipse
 - <http://subclipse.tigris.org/>
- RapidSVN
 - Graphisches Front-End
 - <http://rapidsvn.tigris.org/>
- cvs2svn
 - Konvertiert Repository von CVS nach Subversion
 - <http://cvs2svn.tigris.org/>