

## Software-Entwicklungszyklus

**Anforderungsanalyse:** Anforderungen des Auftraggebers an das zu entwickelnde System ermitteln, strukturieren und prüfen

**Systemarchitektur:** zugrunde liegenden Ressourcen der Hardware und ihr Zusammenwirken

**Konzeption:** Erstellung Fachkonzepte → technischen / fachlichen Inhalt beschreiben

**Programmierung:** Umsetzung DV-Konzeption unter Berücksichtigung Systemarchitektur & Fachkonzept

**Test:** Prüfung Einhaltung Fachkonzept & Anforderungen Auftraggeber

**Abnahme:** Abnahme & Freigabe

**Einsatz:** Entwicklungsumgebung → Produktivsystem

**Wartung:** Fehlerbehebung & Verbesserungen

## Qualitätskriterien von Programmen

**Korrektheit**

**Robustheit**

Umsetzung Anforderungen

Großteil der Entwicklungszeit für Behandlung außergewöhnlicher Situationen

**Wartbarkeit**

**Performance**

Geinger Aufwand für Anpassungen

Schonender Umgang mit Rechnerzeit, Speicher  
Vermeidung Redundanzen

# Komponenten moderner Softwareentwicklung

Sourcecodeverwaltung

Verwendung von Pattern

Teamfähigkeit

Verwendung von Standardkomponenten

Beherrschung grundlegender Sprachparadigmen

Test

## Versionierungssysteme (SVN)

Ziele:

- rekonstruierbarer Datenbestand

- Kennzeichnung Versionen

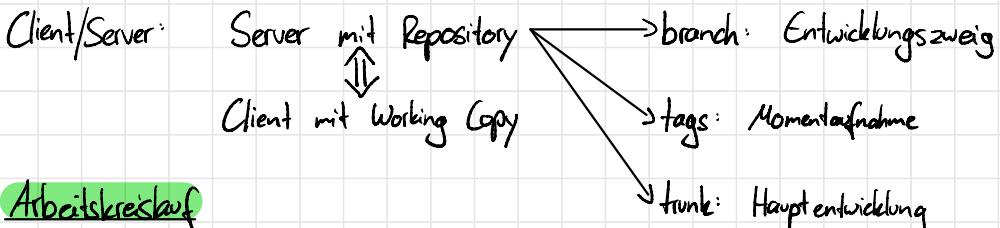
- Nachvollziehbarkeit

- Zugriffsregelung → Schutz

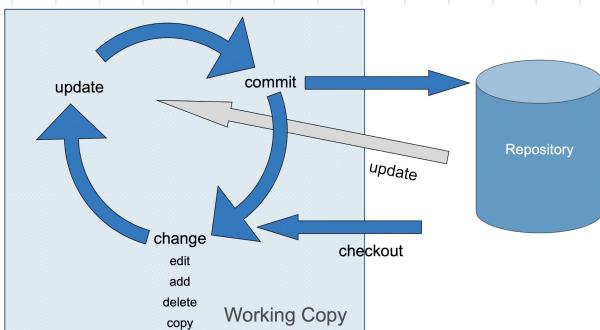
- Verarbeitung konkurrierender Änderungen

## Architektur

Repository: zentraler Speicherort der versionierten Dateien



## Arbeitskreislauf



## Objektorientierung

- Natürliche Modellierung der Umwelt
- Wiederverwendbarkeit (Vererbung)
- Strengere Typsicherheit → Vermeidung Laufzeitfehler

JavaDoc trivial

## Arrays

Kopieren mit System.arraycopy(array, von, kopie, von, längen)

## Modifier

Klassen: public, final, abstract

Methoden: public, private, protected, static, final, abstract, native, synchronized

## Abstrakte Klassen

→ Instanz der Klasse nicht sinnvoll

→ Implementierung der Methoden in Unterklasse verpflichtend

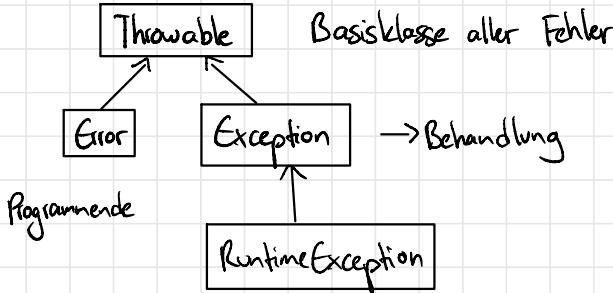
## Package

= logische Gruppierung von (zusammengehörigen) Klassen

→ Strukturierung

## Exception

= Ausnahme



Eigene Fehlerklassen für ergänzende Fehlerinformationen

## Assertions

= Bedingung, die zur Laufzeit ausgewertet wird

false → AssertionEror

## Generics

Vorteil: Methode universell nutzbar und doch typsicher

```
public <T> bool testEqual(T a, T b)
```

Bounding, etc.:

<T extends Number>

→ Nur Unterklassen von Number möglich

<T1 extends Number, T2 extends CharSequence>

<T extends Number & Comparable>

Interface!

public <? extends Number> konvertiere()

↑  
Wildcard

→ keine Festlegung auf einen Datentyp  
→ Unterklassen gehen auch

## Interfaces

Ziel: Sicherstellung, dass Klassen gewisse Eigenschaften haben

↳ Implementierung von mehreren Interfaces möglich ( $\neq$  Vererbung in Java)

```
public class Person implements Comparable<Person> {
```

Comparable: int compareTo(T o) macht Objekte sortierbar

Aufruf: Collections.sort(personenListe);

## Comparator

ermöglicht verschiedene Sortierungen/Vergleiche

```
import java.util.Comparator;  
  
public class ComperatorNummerDesc implements Comparator<Person> {  
  
    @Override  
    public int compare(Person arg0, Person arg1) {  
        return arg1.compareTo(arg0);  
    }  
}
```

```
Collections.sort(persListe, new ComperatorNummerDesc());
```

Absteigende Sortierung

## Anonyme Klassen

Erzeugung einer Implementierung/Unterklasse eines Interfaces oder Klasse,  
die nur an einer Stelle benötigt wird

## Vererbung

Es werden nur sichtbare Methoden und Variablen vererbt

## Polymorphie

Variablen mit Klassentyp können Objekte von verschiedenen Typen referieren, nämlich Typ der Variable selbst und alle Subtypen

## Reflections

Schwierig das abzufragen

## Nebentägigkeit

Sequentiell vs. parallel

Arbeiten mit Threads:

- Vererbung
- anonyme innerer Klasse
- Interfaces (Runnable)

Schlüsselwort **synchronized** zum Synchronisieren von Methoden und Variablen

## Algorithmen

= präzise endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Schritte

**terminierend**: Abbruch nach endlich vielen Schritten

**deterministisch**: eindeutige Vorgabe der Schrittfolge der auszuführenden Schritte festgelegt

**determiniert**: gleiche Eingabe → eindeutiges gleiches Ergebnis

## Sprachen

**Syntax**: gibt vor, welche Satzmuster gebildet werden dürfen

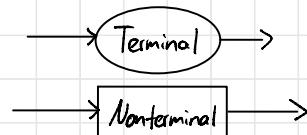
**Semantik**: Bedeutung; Inhalt der Variablen

**Grammatik**: Regelwerk zur Beschreibung der Syntax

**Produktionsregel**: Regel, um mit Grammatik Sätze zu bilden

Wert	BNF	EBNF
Nonterminal	<Bezeichner>	Bezeichner
Terminal	Wert	"Wert"
Optional (0 oder 1 mal)	[...]	oder (...)?
Optional (0 oder n mal)	{...}	oder (...)*
Optional (1 oder n mal)	<...>	oder (...)+
Sequenz	?...? z.B. ("a"..."z")	
Kommentar	(* ...*)	
Definition	::=	= oder ::=
Klammerung	(...)	

## Syntaxdiagramm



## Datentypen

**Algebra:** Wertmenge mit Operationen

**Sorten:** Wertmengen eines Datentyps

**Signatur:** Formalisierung der Schnittstellenbeschreibung eines Datentyps (Sorten + Operationen)  
→ -graphen

## Komplexität von Algorithmen

- ① Rechenzeit abhängig Eingabewerte ( $n$ )
- ② Änderung der Rechenzeit bei Vergrößerung  $n$
- ③ Speicherplatzverbrauch während Laufzeit abhängig  $n$
- ④ Rechenzeit zumutbar große  $n$
- ⑤ Berücksichtigung Eingangsdaten und Ausgangsdaten

Definitionen:

Probleumfang  $n$

Zeitkomplexität abhängig  $n$       → soll unabhängig von HW und Implementierungsweise

Speicherkomplexität abhängig  $n$

## Komplexitätsklassen

$O(1)$

$O(\log_2 n)$  nur Teil der Aufgabe muss bearbeitet werden

$O(n)$  Schleife

$O(n \cdot \log_2(n))$  eine Schleife, Teil der Aufgabe zu lösen Quicksort

$O(n^2)$  Schleife in Schleife Bubblesort

$O(2^n)$  Alle Teilmengen für  $n$  Rucksackproblem

$O(n!)$  Alle Permutationen für  $n$  Reisenderproblem

## Suche

Sequenziell  $O(n)$

Binär  $O(\log_2 n)$  sortierte Liste!

## Sortieren

Interne Verfahren: setzt voraus, alle Werte im Hauptspeicher

Externe Verfahren:  $\hookrightarrow$  Unterstützung externer Medien

Stabilität: Sortierung nach a, dann b, a bleibt erhalten

stabil: Bubble, Insertion, Merge

instabil: Heap, Quick, Selection

## Stabile Algorithmen

### Inserionsort:

Verschiebe Element solange nach links,  
bis es richtig eingesortiert ist

$$O(n^2)$$

### Bubblesort $O(n^2)$

### Mergesort:

- Teile die Menge in Einzelleile
  - Füge zwei Mengen zusammen, indem
    - immer das kleinste Element aus jeder Menge genommen wird, dieses verglichen und eine neue Menge aufgebaut wird, in der alle Elemente der beiden zusammengeführt werden
- wiederhole bis alle Mengen zusammengeführt

$$O(n \log_2 n)$$

## instabile Algorithmen

### Selectionsort:

Suche das größte Element und  
schiebe es nach rechts

$$O(n^2)$$

### Quicksort $O(n \log_2 n)$

## Datenstrukturen

Stack (LIFO)

push (T wert)

T pop()

T top()

Queue (FIFO)

enter (T wert)

T leave()

T front()

Liste

addFirst

removeFirst

addLast

removeLast

+ Getter

Iterator: Schleifendurchlauf möglich

hasNext()

T next()

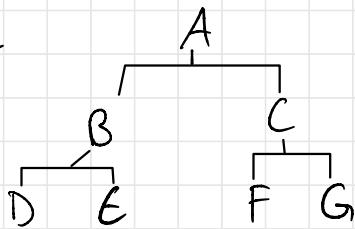
remove()

Doppelt verketzte Liste

Knoten kennt Vor- & Nachfolger

Vorteil: Zugriff auf letztes Element  
 $O(1)$  statt  $O(n)$

Tree



Pfad: Folge von unterschiedlichen Knoten durch Kanten verbunden

Traversierung: Durchlaufen aller Knoten

Entartet:  $n$  Knoten Höhe  $n$

