Software-Entwicklungs-Praktikum

Martin Lange, Uli Schöpp

Martin Burger, Bernhard Frauendienst, Bastian Gebhardt,
Annabelle Klarl, Andy Lehmann, Julien Oster, Michael Weber

Institut für Informatik, LMU München

October 17, 2008

Organisatorisches

Veranstaltung SEP

Publikum

- Informatik Bachelor 3. Semester
- Medien-Informatik Bachelor 3. Semester
- Kunst und Multimedia Bachelor
- andere?

Ziel: Durchführung eines großen Softwareprojekts in Java (Gruppenarbeit)

hier: Ameisenfuttersammelspiel als Server-Client-Anwendung

siehe auch Vorlesung Softwaretechnik (Hennicker)

Charakter der Veranstaltung

keine Vorlesung, sondern Praktikum

- Selbständigkeit verlangt: Planung, Aufteilung, Erarbeitung der Grundlagen
- niedrige und abstrakte Mindestanforderungen an Software
- bei Ausführung “nach oben hin” keine Grenzen gesetzt
- Gruppenarbeit wichtig; keine Durchführung alleine möglich, auch nicht versteckt

Konkurrenz zwischen den Gruppen durch Wettkampf zwischen den Programmen am Ende

Gruppenarbeit

Teilnehmer hat Pflicht, an Gruppenarbeit sich in gleichem Maße zu beteiligen

Gruppe hat die Pflicht, niemanden auszuschließen

nicht alle Teilnehmer gleich bzgl. Können und Wissen; daher kann Eigenanteil am Projekt durchaus verschieden aussehen (z.B. Koordination vs. Programmierung)

verschiedene Rollen (Koordinator, Entwickler, Programmierer, Kunde, etc.) im Team möglich (aber nicht ausschließlich und nicht permanent verteilt!)

Lernziele dieser Veranstaltung

zu erreichende Qualifikationen:

- größere Programmieraufgaben in der Sprache Java zu erledigen
- Software im Team entwickeln
- Kenntnis grundlegender Probleme und Abläufe in der Software-Entwicklung erlangen
- vertiefte Kenntnisse in objektorientierter Software-Entwicklung
- Präsentation von Ergebnissen

Inhalte

- Software-Entwicklung
- Gruppenarbeit / -dynamik
- Versionskontrolle (Subversion / SVN)
- Nebenläufigkeit (Sockets oder RMI)
- Dokumentation (JavaDoc)
- Spezifikation (UML)
- GUI
- Compiler-Technologie (Lexer, Parser)
- Programmiersprachen (Interpreter für AntBrain)
- Algorithmik
- (künstliche Intelligenz)
- ...

Termine

1. Plenum, Do 14-16, E004 HGB, Teilnahme verpflichtend
   - Vermittlung von Lernstoff, Theorie
   - Ansage von Aufgaben
   - Diskussion / Feedback
2. Tutortreffen, wöchentlich fester Termin (und Ort), 60min
   - Bewältigung von Schwierigkeiten
   - Kontrolle der Gruppenarbeit
3. 2 Abnahmen im Semester, 90min, evtl. statt (2) in der Woche

Plenum

Vermittlung von Grundlagen

Aufgaben stellen

- sind als Zeitplan für das Gesamtprojekt zu verstehen
- keine Abgaben

Diskussion von Problemen

Tutortreffen

Vorstellung dessen, was die Gruppe und die einzelnen Teilnehmer in jeweils letzter Woche erarbeitet haben

Planung der Arbeit für die jeweils kommende Woche

Bewältigung von Problemen

Abnahmen

1. Abnahme im Dezember ohne Bewertung (Test-Abnahme)
2. Abnahme im Februar mit Bewertung

Ablauf:

- Vorstellung der Software
  - Vorführen des Programms
  - Inspektion des Source-Codes
- Präsentation des Eigenanteils (Gesamtüberblick und Beispiel)

Ablauf der Abnahme mit Tutor planen

Prüfung

notwendige Voraussetzungen zum Bestehen:

- Gruppe legt am Ende funktionstüchtige Software vor, die den Spezifikationen genügt
- aktive Teilnahme an Tutortreffen
- Mitarbeit in der Gruppe (ca. 20% der Gruppenarbeit durchgeführt)
- Darlegung der individuellen Leistung (bei Tutortreffen und Abnahme)

Prüfung

Note für Teilnehmer (Regelfall):

- Bewertung der Gruppenarbeit, Faktor 0.6
  - Erfüllung der Vorgaben
  - Benutzbarkeit, Wiederverwendbarkeit, Weiterentwickelbarkeit der Software
- Bewertung der individuellen Leistung, Faktor 0.4
  - ca. 5-10min Präsentation über Eigenanteil in der 2. Abnahme
  - (evtl. Eindruck aus wöchentlichen Tutortreffen)
- Gesamtnote durch Aufrunden ermittelt

Individualprüfung (Spezialfall)

Ansetzen einer Individualprüfung durch Dozenten jederzeit möglich

Vorbereitungszeit mindestens 3 Tage

wird gemacht bei erkennbarer Verfehlung der Ziele, z.B. ungenügende Teilnahme an der Gruppenarbeit, kein erkennbarer Lernerfolg bzgl. vermittelter Grundlagen

Ablauf ähnlich einer mündlichen Prüfung, Prüfungsstoff ist Projekt der Gruppe sowie Stoff der Plena

Ergebnis:

- Erfolg: Weiterführen des Praktikums, Note nach Regelfall ermittelt
- Misserfolg: Ende des Praktikums, Note 5.0

Sonderpreise

gibt es am Ende in Form von Urkunden für

- bestes Programm im Wettkampf
- bestes Programm bzgl. Benutzbarkeit
- bestes Programm bzgl. Wiederverwendbarkeit

Hilfe

- Gruppe
- Tutor
- Plenum
- Forum `http://www.die-informatiker.net`
- Internet
- ...

Gruppenaufteilung

Kriterien:

- lose Gruppenzugehörigkeit bei Anmeldung im Juni nicht verpflichtend
- 5 Teilnehmer pro Gruppe
- höchstens ein Kunst-und-Multimedia-Student pro Gruppe
- falls Teilnehmerzahl $\not\equiv 0 \mod 5$, dann auch 6-er Gruppe, aber mit einem K&M-Teilnehmer
- mindestens 3 bestandene Prüfungen "Einführung in die Programmierung" pro Gruppe

keine Schikane, sondern Schutz vor unnötigen Komplikationen

Gruppen finden sich jetzt und registrieren sich bei uns

Gruppenregistrierung

bei der Registrierung anzugeben:

- eindeutiger Gruppenname
- Auswahl aus Liste unten von mindestens 3 möglichen Terminen (60min) für Tutortreffen, an mindestens 2 verschiedenen Tagen

mögliche Termine:

- Montag 8–19
- Dienstag 11–19
- Mittwoch 8–10, 12–14, 16–19
- Donnerstag 8–10, 12–14, 16–19
- Freitag 8–19

Subversion

Sourcecodeverwaltung

zentrale Verwaltung des Sourcecodes unablässig bei Gruppenarbeit

offensichtlich schlecht:

- Code per Email herumschicken: Aktualität, Vollständigkeit
- Code in Verzeichnis mit Zugriff für alle: Überschreiben von Änderungen anderer

stattdessen professionelle Verwaltung des Sourcecodes mit Versionskontrolle: SVN

SVN – Server/Client-Modell

SVN-Server hält Daten: *Repository*

SVN-Clients können sich lokale Kopie davon machen, editieren und ins Repository einspielen

Transfer erfolgt über HTTP

hier: Repositories werden auf Server des Lehrstuhls TCS bereitgestellt; Zugriff über

`http://svn.tcs.ifi.lmu.de/`⟨*Name des Repositories*⟩

Repository

ist mehr als nur Ansammlung von Dateien, sondern auch Meta-Information

- Geschichte der Daten wird protokolliert: Revisionen 1,2,...
- Wer hat welche Änderungen gemacht?
- Nachrichten zu Änderungen

Revision = aktueller Zustand der Dateien zu gewissem Zeitpunkt

Operationen auf/mit Repository:

- `checkout`, lokale Kopie mit letzter Revision erzeugen
- `update`, lokale Kopie auf aktuellen Stand bringen
- `commit`, Änderungen an lokaler Kopie ins Repository einspielen

Checkout

lokale Kopie wird so erzeugt

```
mlange@pozzuoli:~/tmp> svn checkout --username lange http://svn.tcs.ifi.lmu.de/
Authentication realm: <http://svn.tcs.ifi.lmu.de:80> Repository SEP Project
Password for 'lange':
A    SEP/MeineKlasse.java
...
Checked out revision 117.
```

erzeugt lokale Kopie, die editiert werden kann

Usernamen und Passwörter werden von uns vergeben und per Email verschickt

Ausgabe von `svn checkout` ist Liste von Dateien mit Flag für Aktionen

Update

lokale Kopie auf aktuellen Stand bringen

```
mlange@pozzuoli:~/tmp/SEP> svn update
A    MeineKlasse.java
...
Updated to revision 118.
```

lokale Änderungen gehen dabei nicht verloren

Statusmeldungen:

`A`	`Datei.java`	Datei wurde neu angelegt
`D`	`Datei.java`	Datei wurde gelöscht
`U`	`Datei.java`	Datei wurde auf aktuellen Stand gebracht
`M`	`Datei.java`	...und dabei wurden Änderungen im Repository mit lokalen Änderungen vereint
`C`	`Datei.java`	...und dies führte zu einem Konflikt

Commit

lokale Änderungen ins Repository einspielen, um sie den anderen zugänglich zu machen

```
mlange@pozzuoli:~/tmp/SEP> svn commit
```

öffnet Editor, um Nachricht einzutragen; diese sollte Änderungen für die Teammitglieder beschreiben

Commit nicht immer möglich, z.B. falls sich Repository mittlerweile ebenfalls geändert hat; dann zuerst nochmal Update

Merging und Konflikte

betrachte folgenden Ablauf, `Datei.java` ist im Repository

1. User *A* ändert `Datei.java` bei sich
2. User *B* ändert `Datei.java` bei sich
3. User *A* führt `svn commit` aus

User *B* kann dann kein Commit durchführen, aber Update von User *B* kann zu Problemen führen

SVN versucht, Änderungen im Repository und lokale zu vereinen; mehr oder weniger zeilenweise

Vereinigung nicht möglich: Konflikt; wird in Datei markiert

```
<<<<<<< .mine
lokale Aenderung von User B
=======
Aenderung im Repository
>>>>>>> .r118
```

Lösung: Änderungen per Hand vereinen, dann

```
mlange@pozzuoli:~/tmp/SEP> svn resolved Datei.java
```

Sonstiges

Update kann Datei aus bestimmter Revision wiederherstellen (Option `-r`); keine Backups in lokaler Kopie nötig

`svn move` verschiebt Dateien lokal und bei nächstem Commit auch im Repository

`svn delete` genauso für Löschen

Verhaltensregeln

was SVN nicht erzwingt, wir aber trotzdem verlangen

- aussagekräftige Nachrichten bei Commit angeben
- keine Binärdateien (z.B. `.class`) im Repository (Ausnahme: unveränderliche und nicht erzeugbare wie z.B. Bilder, Sounds)
- keine Backup-Dateien im Repository
- nur syntaktisch korrekte Java-Dateien einchecken
- regelmäßige Updates durchführen

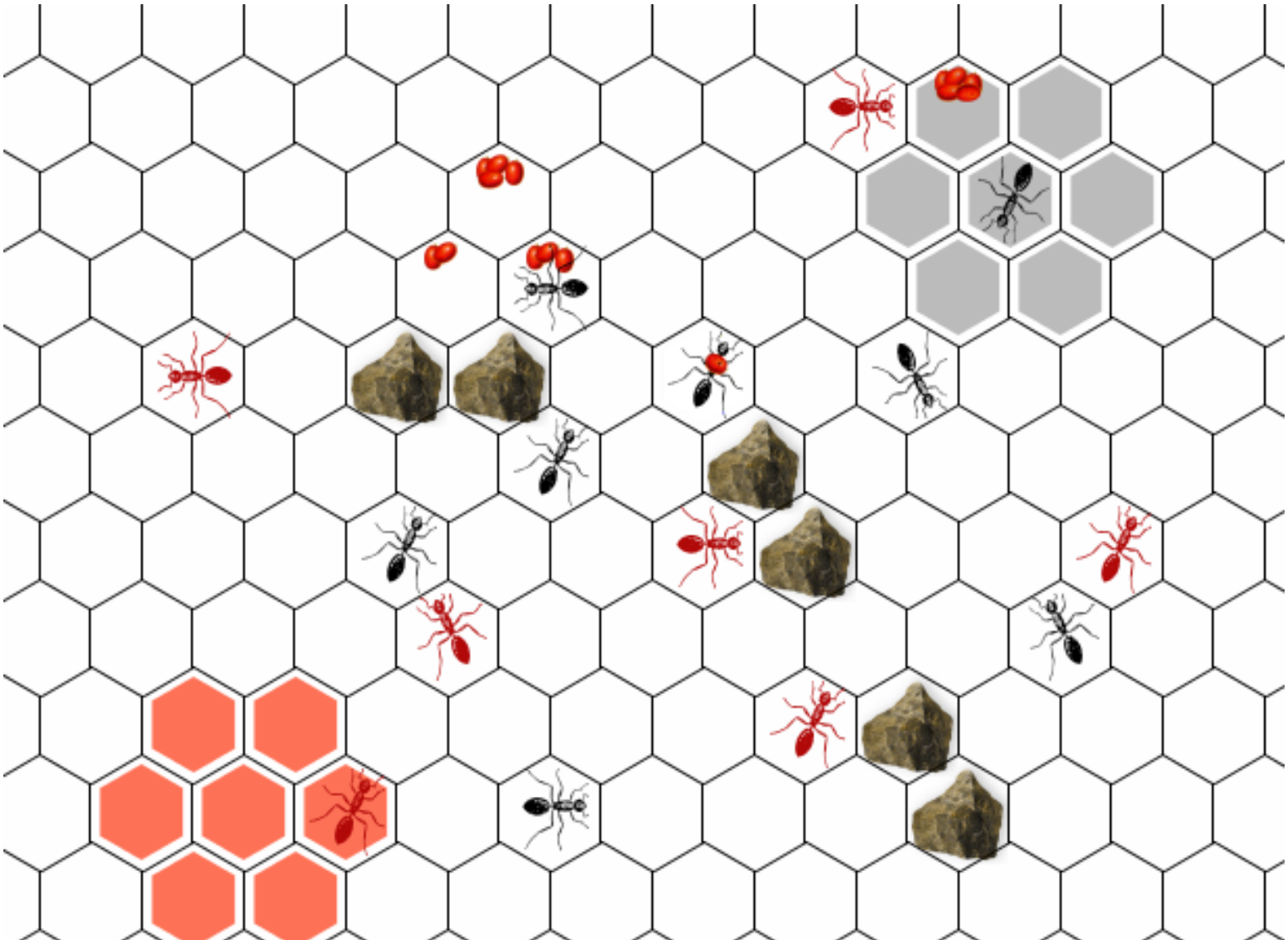
Hilfe

```
mlange@pozzuoli:~/tmp/SEP> svn help

mlange@pozzuoli:~/tmp/SEP> svn help ⟨Kommando⟩

http://svnbook.red-bean.com/
```

Das Projekt

Die Ameisenart formica artificiosa

Ameisen dieser Art in ihrer natürlichen Umgebung



Hexagonale Zellen

Jede Zelle

- kann ein Hindernis enthalten (vielleicht einen Stein oder Wasser, aber das ist nicht näher spezifiziert).
- kann *eine* Ameise enthalten, wenn sie nicht schon ein Hindernis enthält.
- kann eine beliebige Anzahl von Futterstücken enthalten.
- kann ein Teil des Ameisenhügels eines bestimmten Ameisenstammes sein.
- kann Duftspuren enthalten. Jeder Ameisenstamm hat sechs verschiedene Duftmarker, von dem jeder entweder in der Zelle vorhanden ist oder nicht.

Ameisen

Ameisen

- verhalten sich völlig autonom.
- kommunizieren und orientieren sich mit Duftmarkern.
- sind in Ameisenstämmen organisiert.

Ameisenstämme

- stehen im Wettbewerb um Futter.
- werden mit (hoffentlich) intelligentem Design entworfen.

Praktikumsinhalt

Wir wollen die Ameisenart *formica artificiosa* in `Java` simulieren und Programme schreiben

- zur Entwicklung und Erprobung von Ameisenstämmen.
- zum Ausrichten von Wettbewerben zwischen Ameisendesignern.

Ein Wettbewerb für Ameisendesigner

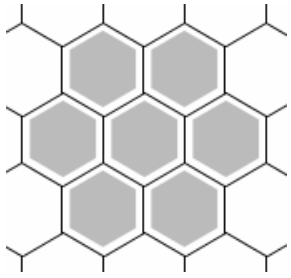
Das Ziel ist, einen Ameisenstamm zu entwickeln, der in allen Situationen möglichst viele Futterstücke findet und zurück zu seinem Ameisenhügel bringt.

- Spieler entwickeln Ameisenstämme.
- Ameisenstämme treten gegeneinander auf einem Spielfeld an.
- Gesamtgewinner wird in einem Turnier ermittelt.

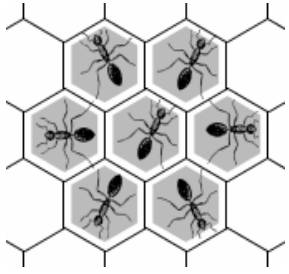
Spielablauf I

Die Spieler entwickeln ihre Ameisenstämme und reichen sie dann auf einem Spielserver ein.

- Bekannt ist nur, wie groß der Ameisenhaufen ist, aber nicht, wie das Spielfeld aussieht.



- Spieler legt fest, wie seine Ameisen auf den Haufen zu setzen sind.



- Spieler bestimmt das Verhalten der Ameisen, indem er das Gehirn jeder Ameise „programmiert“.

Spielablauf II

Wenn alle Spieler ihre Ameisenstämme eingereicht haben, dann kann man verschieden Stämme gegeneinander antreten lassen.

1. Setze Ameisenstämme auf verschiedene Ameisenhaufen desselben Spielbretts.
   - Üblicherweise spielen jeweils zwei Stämme gegeneinander, aber auf größeren Spielfeldern sind auch mehr möglich.
   - Alle Ameisenhaufen sind gleich groß.
     $\Rightarrow$ Rückspiele mit vertauschten Ameisenhügeln möglich
2. Lasse die Ameisen eine bestimmte Zeit lang fleißig arbeiten.
3. Gewonnen hat, wer danach die meisten Futterstücke auf seinem Hügel liegen hat.

Praktikumsinhalt

Wir wollen die Ameisenart *formica artificiosa* in `Java` simulieren und Programme schreiben

- zur Entwicklung und Erprobung von Ameisenstämmen.
- zum Ausrichten von Wettbewerben zwischen Ameisendesignern.

Entwicklung von Ameisenstämmen

Gehirne werden in der Programmiersprache AntBrain programmiert.

```
lookingForFood = 1;
while true do {
  if sense(here, food) then {pickup; lookingForFood = 0} else skip;
  if sense(here, home) then {drop; lookingForFood = 1} else skip;
  t = rand%2 + rand%2 - 1;  turn(t);
  walk
}
```

Die Ameisen eines Ameisenstammes sind gegeben durch:

- ihre Position auf dem Ameisenhaufen
- ein AntBrain-Programm, welches ihr Verhalten bestimmt

Entwicklung von Ameisenstämmen

Programmiersprache AntBrain

- Minimalanforderung
- Serverkompatibilität
- einfach zu implementieren, dafür nicht sehr bequem

Ameisenentwicklung

- Werkzeuge zum Experimentieren mit Ameisen
- Erweiterte Sprachen, die in AntBrain zurückübersetzt werden
- ...

Simulation der Ameisen im Spielserver

Wenn Ameisenstämme gegeneinander antreten passiert Folgendes:

1. Die Ameisenstämme werden auf Ameisenhaufen gesetzt und die Gehirne initialisiert.
2. Die Ameisen bewegen sich dann selbständig, indem sie die Programme in ihren Gehirnen abarbeiten.
   - Zeit ist durch die Anzahl der Spielrunden beschränkt (Größenordnung: 1 Million).
   - In jede Spielrunde kommen die Ameisen nacheinander dran und machen je genau einen Schritt in ihrem AntBrain-Programm.
   - Durch Instruktionen wie `walk` oder `turn(i)` können sich die Ameisen auf diese Art bewegen.
3. Am Ende wird ermittelt, wie viele Futterstücke jeder Ameisenstamm auf seinem Ameisenhaufen liegen hat.

Detailinformationen

Siehe Spezifikation.

Aufgabe

Entwicklung zweier getrennter Komponenten, die über eine Netzwerkverbindung miteinander kommunizieren können.

1. Der Server
   - speichert Spielfelder.
   - erlaubt den Spielern Ameisenstämme einzureichen.
   - berechnet, was passiert wenn verschiedene Ameisenstämme auf einem Spielfeld gegeneinander antreten.
   - speichert Spiele zur späteren Ansicht.
2. Der Client
   - erlaubt das Editieren und Ausprobieren von Ameisenstämmen.
   - kann Ameisenstämme bei einem Server einreichen.
   - zeigt die auf einem Server liegenden Spiele grafisch an.

Mehrere Spieler können gegeneinander antreten, indem sie sich mit ihre Clients mit demselben Server verbinden.

Remote Method Invocation

Netzwerkprogrammierung in Java

Package `java.net` implementiert Protokolle zur Kommunikation über das Internet Protocol (IP).

User Datagram Protocol (UDP)

- verbindungslos, es werden einfach Datenpakete verschickt und empfangen
- ohne Garantien, dass Pakete ankommen und in welcher Reihenfolge

Transfer Control Protocol (TCP)

- verbindungsorientiert
- mit Fehlerkorrektur

Applikationsprotokolle (HTTP, FTP, ...)

Möglichkeiten der Netzwerkkommunikation

Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert

Beispiel: SMTP

```
...
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Hallo!
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

Möglichkeiten der Netzwerkkommunikation

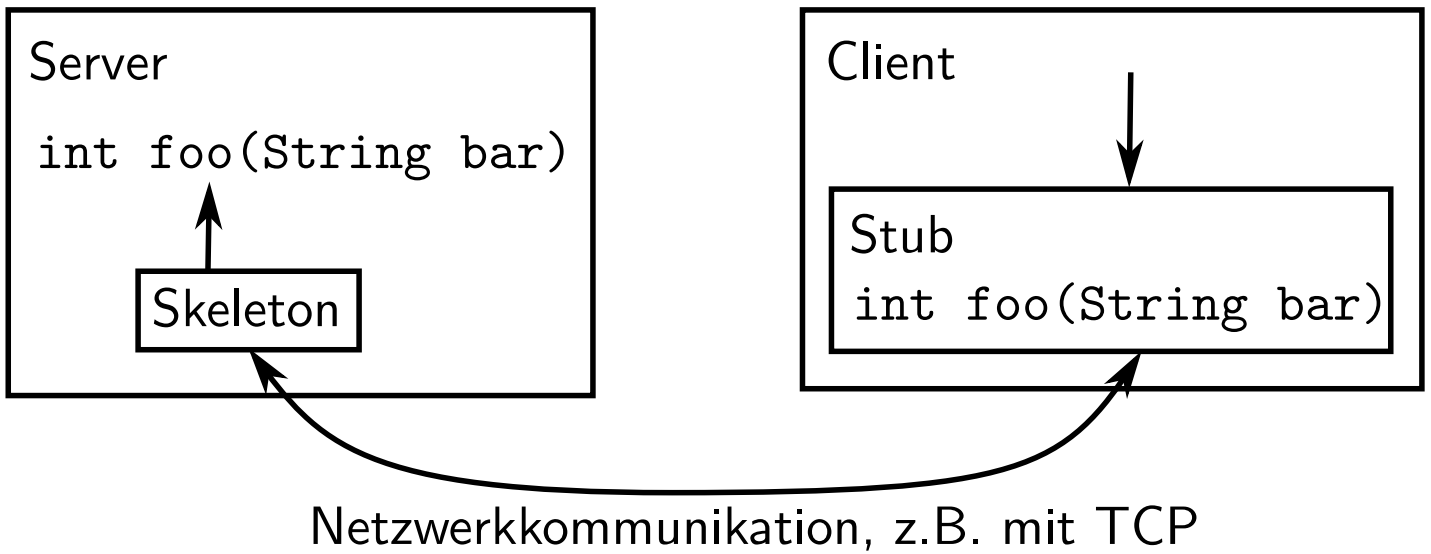
Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert
- aufwendig
  - Protokollentwurf
  - Kodierung und Dekodierung
  - Synchronisierung
- flexibel
  - heterogene Umgebungen
  - Fehlerbehandlung
  - effiziente Kodierungen möglich

Verteilte Java-Anwendungen

Typischer Fall:

- Server stellt Funktion bereit: `int foo(String bar)`
- Client möchte diese Funktion aufrufen
- Implementierung mittels Stubs/Skeletons.
  1. Stub kodiert Argument und schickt es an Skeleton
  2. Skeleton dekodiert es, ruft `foo` auf, kodiert das Ergebnis und schickt es zum Stub
  3. Stub dekodiert das Ergebnis und gibt es zurück



Remote Method Invocation (RMI)

Java-spezifische Methode zum entfernten Methodenaufruf

- automatische Generierung von Stub- und Skeletonklassen
- Client kann Methoden auf dem Server wie normale Java-Funktionen aufrufen
- transparentes Verschicken von Objekten, inklusive ihrer Funktionen
- dynamisches Nachladen von Objektklassen

RMI — Überblick

Entfernt aufrufbare Objekte haben folgenden Eigenschaften:

- Sie implementieren das (leere) Interface `java.rmi.Remote`
- Jede ihrer Methoden deklariert `throws java.rmi.RemoteException`

Server erzeugt ein solches Objekt und registriert es unter einem bestimmten Namen in der *RMI Registry* (Namensdienst).

Client holt sich eine Referenz auf das Objekt von der *RMI Registry*.

Alle Methodenaufrufe des Clients werden dann über das Netzwerk an das Objekt im Server weitergegeben.

Beispiel: Zeitserver

Entfernt aufrufbares Interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Time extends Remote {
   public long getTime() throws RemoteException;
}
```

- Jedes entfernt aufrufbare Interface leitet von `Remote` ab.
- Jede Methode muss die Exception `RemoteException` werfen können.

Implementierung

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public class TimeImpl implements Time {
   public long getTime() throws RemoteException {
      return System.currentTimeMillis();
   }
}
```

Server

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class Server {
public static void main(String args[]) {
    try {
      TimeImpl ts = new TimeImpl();
      Time stub = (Time)UnicastRemoteObject.exportObject(ts, 0);

      // Den Stub registrieren
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("Time", stub);

      System.err.println("TimeServer bereit");
    } catch (Exception e) {...}
  }
}
```

Server

```
TimeImpl ts = new TimeImpl();
Time stub =
 (Time)UnicastRemoteObject.exportObject(ts,0);
```

- `UnicastRemoteObject.exportObject(Remote obj, int port)` erzeugt ein Stub-Objekt für `obj` und exportiert `obj` über einen TCP-Port (`port=0` für einen anonymen Port).
- Das Stub-Objekt werden sich später die Clients holen und es dann benutzen, um mit dem Server zu kommunizieren.
- `Unicast` bedeutet, dass das Stub-Objekt mit einem einzigen Server kommuniziert (hier mit Objekt `ts`).

Server

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind("Time", stub);
```

- Registriere das Stub-Objekt unter dem Namen „Time“ in der RMI Registry.
- Clients können es sich jetzt unter diesem Namen erfragen.

Server starten

1. `Time.java`, `TimeImpl.java` und `Server.java` normal kompilieren, z.B. im Verzeichnis `/home/schoepp/rmi/`
2. Registry starten:

   `rmiregistry &` (Unix)

   `start rmiregistry` (Windows)
3. Server starten:

   ```
   java -Djava.rmi.server.codebase=url Server
   ```

   Dabei ist `url` eine URL, die auf die Klassen des Servers zeigt, z.B. `file:/home/schoepp/rmi/` (letztes '/' ist wichtig!)

   Die URL muss man angeben, damit die RMI-Implementierung die Stubs aus den Klassen des Servers erzeugen kann.

Client Implementierung

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
  public static void main(String[] args) {
    String host = (args.length < 1) ? null : args[0];
    try {
      Registry reg = LocateRegistry.getRegistry(host);
      Time server = (Time)reg.lookup("Time");
      System.out.println("Zeit auf dem Server: "
                         + server.getTime());
    } catch (Exception e) {...}
  }
}
```

Client starten

1. `Time.java` und `Client.java` normal kompilieren
2. Client starten:

   ```
   java -Djava.rmi.server.codebase=url Client
   ```

   Dabei ist nun `url` eine URL, die auf die Klassen des Clients zeigt.

   Die URL wäre nötig, wenn der Server sich noch Klassen vom Client holen muss, die dieser an den Server schicken will.

   In diesem Beispiel ist das aber nicht nötig und man könnte die `codebase`-URL auch weglassen.

Weitere Themen

- Übergabe von komplexen Argumenten
- Callbacks
- Nebenläufigkeit
- dynamisches Nachladen von Klassen

Argumentübergabe

Welche Klassen `A` und `B` können in einem `Remote`-Interface benutzt werden?

```
public interface I extends Remote {
  public A funktion(B x) throws RemoteException;
}
```

1. Klassen, die `java.rmi.Remote` implementieren.
   Objekte, die zu solchen Klassen gehören, werden per *Referenz* übergeben (wie sonst auch).
2. Klassen, die `java.io.Serializable` implementieren.
   Objekte, die zu solchen Klassen gehören, werden *kopiert*!

   Beispiel: `List<List<Integer>>` (`int[][]` ist effizienter)

Wenn `A` oder `B` keines der beiden Interfaces implementiert, dann wird der Aufruf (zur Laufzeit!) fehlschlagen.

Beispiel: Callbacks

- Server bietet Funktion `int getNumber()` an.
- Client möchte wissen, wenn sich die Zahl ändert.
- Möglichkeiten
  1. Polling:
     Client fragt alle 100ms beim Server nach.
  2. Callbacks:
     Server sagt dem Client, wenn sich die Anzahl geändert hat.

Beispiel: Callbacks

```
public interface Server extends java.rmi.Remote {
  public int getNumber();
  public void registerChangeNotifier(ChangeNotifier n);
}

public interface ChangeNotifier extends java.rmi.Remote {
  public void changed();
}
```

- Client erzeugt Objekt, das `ChangeNotifier` implementiert und gibt eine Referenz darauf mit `registerChangeNotifier` an den Server.
- Server kann jetzt `changed()` aufrufen, um dem Client zu sagen, dass etwas passiert ist.

Übergabe der Referenzen

1. Server und Client erzeugen Objekte `ServerImpl` und `NotifierImpl`, die `Server` und `ChangeNotifier` implementieren.
2. Server gibt eine Referenz auf `ServerImpl` an die RMI Registry.
3. Client holt sich Referenz auf `ServerImpl` von der Registry
4. Client gibt eine Referenz auf `NotifierImpl` an den Server, indem er `registerChangeNotifier` aufruft.
5. Server kann nun `NotifierImpl.changed()` aufrufen.

Würde `NotifierImpl` statt `Remote` das Interface `Serializable` implementieren, dann würde `NotifierImpl.changed()` auf dem Server ausgeführt.

Nebenläufigkeit

Spezifikation:

> *A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads.*

- Suns RMI-Implementierung benutzt Thread-Pools.
- Jede Methode in einem `Remote`-Objekt kann gleichzeitig von mehreren Clients aufgerufen werden.

$\Rightarrow$ Man muss selbst für Synchronisierung sorgen.

Dynamisches Laden von Klassen

```
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t)
      throws RemoteException;
}
public interface Task<T> extends java.io.Serializable {
    T execute();
}
```

Implementierung im Server:

```
T executeTask(Task<T> t) {
  return t.execute();
}
```

Client kann beliebige Funktionen an den Server schicken, die dann dort ausgerechnet werden.

Quelle: `http://java.sun.com/docs/books/tutorial/rmi`

Dynamisches Laden von Klassen

Zweites Beispiel:

- Ein Serverobjekt implementiert zwei voneinander unabhängige `Remote`-Interfaces `A` und `B`.
- Der Client kennt nur das Interface `A`.

Zur Übertragung des Stubs für das Serverobjekt, muss RMI beide Interfaces `A` und `B` auf der Client-Seite haben.

Dynamisches Laden von Klassen

RMI lädt fehlende Klassen automatisch nach, *wenn das vom Security Manager erlaubt ist*.

- Übertragung sowohl vom Server zum Client als auch vom Client zum Server.
- RMI findet die Klassen, indem es unter den URLs in `java.rmi.server.codebase` nachschaut.

Standardmäßig ist kein Security Manager installiert und es werden deshalb keine Klassen nachgeladen.

Security Manager

RMI Security Manager installieren

```
public static void main(String[] args) {
  if(System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
    }
  ...
```

Sicherheitsregeln beim Start von Server/Client übergeben:

```
java -Djava.rmi.server.codebase=url
     -Djava.security.policy=security.policy
     Client
```

Datei `security.policy` enthält die Sicherheitsregeln.

Sicherheitsregeln

Beispiele für die Datei `security.policy`:

- Alles ist erlaubt (inkl. Festplatte löschen):

```
grant {
  permission java.security.AllPermission;
};
```

- Kommunikation über Sockets (TCP) und lesender Dateizugriff

```
grant {
  permission java.net.SocketPermission
    "*:1024-65535", "connect,accept";
  permission java.io.FilePermission
    "<<ALL FILES>>", "read";
}
```

Aufgabe

Jede Gruppe schreibt bis nächste Woche mit RMI ein Client/Server-Chatprogramm.

Interfacevorschlag:

```
public interface ChatServer extends java.rmi.Remote {
  ServerInterface register(String nick, ClientInterface client)
    throws java.rmi.RemoteException;
}

public interface ClientInterface extends java.rmi.Remote {
  void hear(String nick, String message)
    throws java.rmi.RemoteException;
}

public interface ServerInterface extends java.rmi.Remote {
  void say(String message) throws java.rmi.RemoteException;
  void signoff() throws java.rmi.RemoteException;
}
```

Remote Method Invocation

Netzwerkprogrammierung in Java

Package `java.net` implementiert Protokolle zur Kommunikation über das Internet Protocol (IP).

User Datagram Protocol (UDP)

- verbindungslos, es werden einfach Datenpakete verschickt und empfangen
- ohne Garantien, dass Pakete ankommen und in welcher Reihenfolge

Transfer Control Protocol (TCP)

- verbindungsorientiert
- mit Fehlerkorrektur

Applikationsprotokolle (HTTP, FTP, ...)

Möglichkeiten der Netzwerkkommunikation

Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert

Beispiel: SMTP

```
...
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Hallo!
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

Möglichkeiten der Netzwerkkommunikation

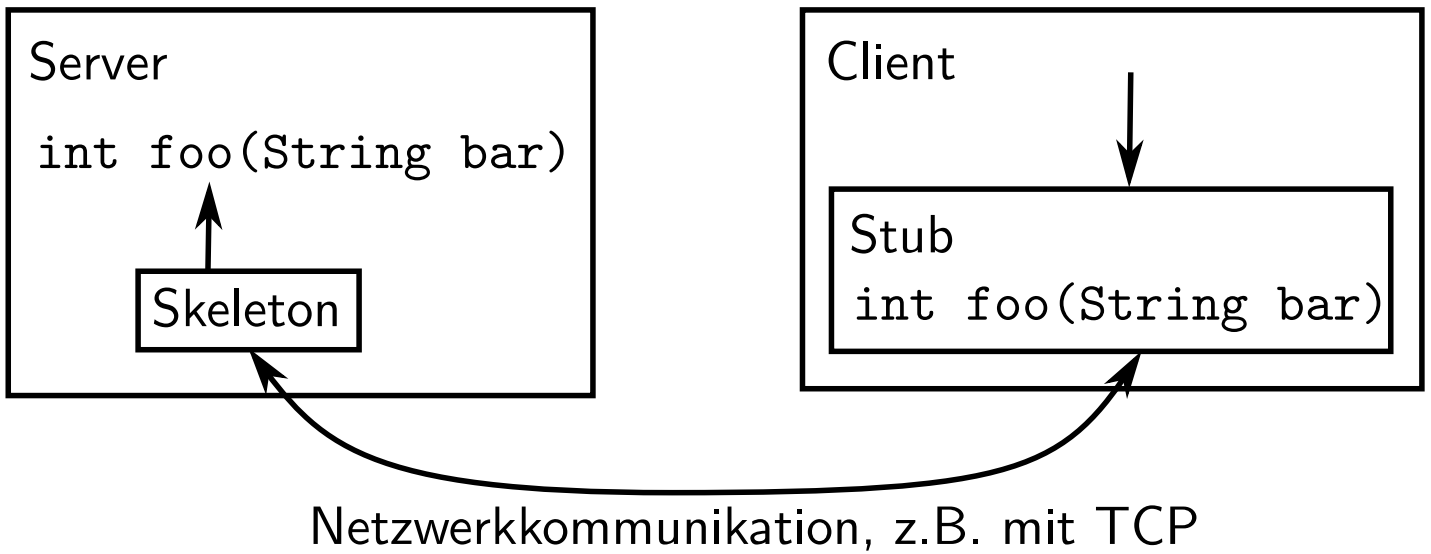
Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert
- aufwendig
  - Protokollentwurf
  - Kodierung und Dekodierung
  - Synchronisierung
- flexibel
  - heterogene Umgebungen
  - Fehlerbehandlung
  - effiziente Kodierungen möglich

Verteilte Java-Anwendungen

Typischer Fall:

- Server stellt Funktion bereit: `int foo(String bar)`
- Client möchte diese Funktion aufrufen
- Implementierung mittels Stubs/Skeletons.
  1. Stub kodiert Argument und schickt es an Skeleton
  2. Skeleton dekodiert es, ruft `foo` auf, kodiert das Ergebnis und schickt es zum Stub
  3. Stub dekodiert das Ergebnis und gibt es zurück



Remote Method Invocation (RMI)

Java-spezifische Methode zum entfernten Methodenaufruf

- automatische Generierung von Stub- und Skeletonklassen
- Client kann Methoden auf dem Server wie normale Java-Funktionen aufrufen
- transparentes Verschicken von Objekten, inklusive ihrer Funktionen
- dynamisches Nachladen von Objektklassen

RMI — Überblick

Entfernt aufrufbare Objekte haben folgenden Eigenschaften:

- Sie implementieren das (leere) Interface `java.rmi.Remote`
- Jede ihrer Methoden deklariert `throws java.rmi.RemoteException`

Server erzeugt ein solches Objekt und registriert es unter einem bestimmten Namen in der *RMI Registry* (Namensdienst).

Client holt sich eine Referenz auf das Objekt von der *RMI Registry*.

Alle Methodenaufrufe des Clients werden dann über das Netzwerk an das Objekt im Server weitergegeben.

Beispiel: Zeitserver

Entfernt aufrufbares Interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Time extends Remote {
   public long getTime() throws RemoteException;
}
```

- Jedes entfernt aufrufbare Interface leitet von `Remote` ab.
- Jede Methode muss die Exception `RemoteException` werfen können.

Implementierung

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public class TimeImpl implements Time {
   public long getTime() throws RemoteException {
      return System.currentTimeMillis();
   }
}
```

Server

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class Server {
public static void main(String args[]) {
    try {
      TimeImpl ts = new TimeImpl();
      Time stub = (Time)UnicastRemoteObject.exportObject(ts, 0);

      // Den Stub registrieren
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("Time", stub);

      System.err.println("TimeServer bereit");
    } catch (Exception e) {...}
  }
}
```

Server

```
TimeImpl ts = new TimeImpl();
Time stub =
 (Time)UnicastRemoteObject.exportObject(ts,0);
```

- `UnicastRemoteObject.exportObject(Remote obj, int port)` erzeugt ein Stub-Objekt für `obj` und exportiert `obj` über einen TCP-Port (`port=0` für einen anonymen Port).
- Das Stub-Objekt werden sich später die Clients holen und es dann benutzen, um mit dem Server zu kommunizieren.
- `Unicast` bedeutet, dass das Stub-Objekt mit einem einzigen Server kommuniziert (hier mit Objekt `ts`).

Server

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind("Time", stub);
```

- Registriere das Stub-Objekt unter dem Namen „Time“ in der RMI Registry.
- Clients können es sich jetzt unter diesem Namen erfragen.

Server starten

1. `Time.java`, `TimeImpl.java` und `Server.java` normal kompilieren, z.B. im Verzeichnis `/home/schoepp/rmi/`
2. Registry starten:

   `rmiregistry &` (Unix)

   `start rmiregistry` (Windows)
3. Server starten:

   ```
   java -Djava.rmi.server.codebase=url Server
   ```

   Dabei ist `url` eine URL, die auf die Klassen des Servers zeigt, z.B. `file:/home/schoepp/rmi/` (letztes '/' ist wichtig!)

   Die URL muss man angeben, damit die RMI-Implementierung die Stubs aus den Klassen des Servers erzeugen kann.

Client Implementierung

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
  public static void main(String[] args) {
    String host = (args.length < 1) ? null : args[0];
    try {
      Registry reg = LocateRegistry.getRegistry(host);
      Time server = (Time)reg.lookup("Time");
      System.out.println("Zeit auf dem Server: "
                         + server.getTime());
    } catch (Exception e) {...}
  }
}
```

Client starten

1. `Time.java` und `Client.java` normal kompilieren
2. Client starten:

   ```
   java -Djava.rmi.server.codebase=url Client
   ```

   Dabei ist nun `url` eine URL, die auf die Klassen des Clients zeigt.

   Die URL wäre nötig, wenn der Server sich noch Klassen vom Client holen muss, die dieser an den Server schicken will.

   In diesem Beispiel ist das aber nicht nötig und man könnte die `codebase`-URL auch weglassen.

Weitere Themen

- Übergabe von komplexen Argumenten
- Callbacks
- Nebenläufigkeit
- dynamisches Nachladen von Klassen

Argumentübergabe

Welche Klassen `A` und `B` können in einem `Remote`-Interface benutzt werden?

```
public interface I extends Remote {
  public A funktion(B x) throws RemoteException;
}
```

1. Klassen, die `java.rmi.Remote` implementieren.
   Objekte, die zu solchen Klassen gehören, werden per *Referenz* übergeben (wie sonst auch).
2. Klassen, die `java.io.Serializable` implementieren.
   Objekte, die zu solchen Klassen gehören, werden *kopiert*!

   Beispiel: `List<List<Integer>>` (`int[][]` ist effizienter)

Wenn `A` oder `B` keines der beiden Interfaces implementiert, dann wird der Aufruf (zur Laufzeit!) fehlschlagen.

Beispiel: Callbacks

- Server bietet Funktion `int getNumber()` an.
- Client möchte wissen, wenn sich die Zahl ändert.
- Möglichkeiten
  1. Polling:
     Client fragt alle 100ms beim Server nach.
  2. Callbacks:
     Server sagt dem Client, wenn sich die Anzahl geändert hat.

Beispiel: Callbacks

```
public interface Server extends java.rmi.Remote {
  public int getNumber();
  public void registerChangeNotifier(ChangeNotifier n);
}

public interface ChangeNotifier extends java.rmi.Remote {
  public void changed();
}
```

- Client erzeugt Objekt, das `ChangeNotifier` implementiert und gibt eine Referenz darauf mit `registerChangeNotifier` an den Server.
- Server kann jetzt `changed()` aufrufen, um dem Client zu sagen, dass etwas passiert ist.

Übergabe der Referenzen

1. Server und Client erzeugen Objekte `ServerImpl` und `NotifierImpl`, die `Server` und `ChangeNotifier` implementieren.
2. Server gibt eine Referenz auf `ServerImpl` an die RMI Registry.
3. Client holt sich Referenz auf `ServerImpl` von der Registry
4. Client gibt eine Referenz auf `NotifierImpl` an den Server, indem er `registerChangeNotifier` aufruft.
5. Server kann nun `NotifierImpl.changed()` aufrufen.

Würde `NotifierImpl` statt `Remote` das Interface `Serializable` implementieren, dann würde `NotifierImpl.changed()` auf dem Server ausgeführt.

Nebenläufigkeit

Spezifikation:

> *A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads.*

- Suns RMI-Implementierung benutzt Thread-Pools.
- Jede Methode in einem `Remote`-Objekt kann gleichzeitig von mehreren Clients aufgerufen werden.

$\Rightarrow$ Man muss selbst für Synchronisierung sorgen.

Dynamisches Laden von Klassen

```
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t)
      throws RemoteException;
}
public interface Task<T> extends java.io.Serializable {
    T execute();
}
```

Implementierung im Server:

```
T executeTask(Task<T> t) {
  return t.execute();
}
```

Client kann beliebige Funktionen an den Server schicken, die dann dort ausgerechnet werden.

Quelle: `http://java.sun.com/docs/books/tutorial/rmi`

Dynamisches Laden von Klassen

Zweites Beispiel:

- Ein Serverobjekt implementiert zwei voneinander unabhängige `Remote`-Interfaces `A` und `B`.
- Der Client kennt nur das Interface `A`.

Zur Übertragung des Stubs für das Serverobjekt, muss RMI beide Interfaces `A` und `B` auf der Client-Seite haben.

Dynamisches Laden von Klassen

RMI lädt fehlende Klassen automatisch nach, *wenn das vom Security Manager erlaubt ist*.

- Übertragung sowohl vom Server zum Client als auch vom Client zum Server.
- RMI findet die Klassen, indem es unter den URLs in `java.rmi.server.codebase` nachschaut.

Standardmäßig ist kein Security Manager installiert und es werden deshalb keine Klassen nachgeladen.

Security Manager

RMI Security Manager installieren

```
public static void main(String[] args) {
  if(System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
    }
  ...
```

Sicherheitsregeln beim Start von Server/Client übergeben:

```
java -Djava.rmi.server.codebase=url
     -Djava.security.policy=security.policy
     Client
```

Datei `security.policy` enthält die Sicherheitsregeln.

Sicherheitsregeln

Beispiele für die Datei `security.policy`:

- Alles ist erlaubt (inkl. Festplatte löschen):

```
grant {
  permission java.security.AllPermission;
};
```

- Kommunikation über Sockets (TCP) und lesender Dateizugriff

```
grant {
  permission java.net.SocketPermission
    "*:1024-65535", "connect,accept";
  permission java.io.FilePermission
    "<<ALL FILES>>", "read";
}
```

Aufgabe

Jede Gruppe schreibt bis nächste Woche mit RMI ein Client/Server-Chatprogramm.

Interfacevorschlag:

```
public interface ChatServer extends java.rmi.Remote {
  ServerInterface register(String nick, ClientInterface client)
    throws java.rmi.RemoteException;
}

public interface ClientInterface extends java.rmi.Remote {
  void hear(String nick, String message)
    throws java.rmi.RemoteException;
}

public interface ServerInterface extends java.rmi.Remote {
  void say(String message) throws java.rmi.RemoteException;
  void signoff() throws java.rmi.RemoteException;
}
```

Unified Modelling Language

Software-Entwicklung

Software-Entwicklung ist Prozess von Anforderung über Modellierungen zu fertigen Programmen

Anforderungen oft informell gegeben

fertige Programme sollen am Ende Anforderungen genügen

UML ist Werkzeug zur Modellierung verschiedener Aspekte der Software, z.B.

- Welche Komponenten gibt es?
- Was sollen diese im Einzelnen leisten können?
- Wie verhalten sich diese im Einzelnen?
- Wie interagieren sie miteinander?
- ...

UML-Diagramme

UML ist grafische Modellierungssprache

Sachverhalte werden in Form von Diagrammen dargestellt

es gibt verschiedene Arten von Diagrammen mit jeweils verschiedenen Aufgaben

- Anwendungsfalldiagramme
- Klassendiagramme
- (Objektdiagramme)
- Sequenzdiagramme
- Zustandsdiagramme
- ...

Verwendung von Diagrammen

primärer Einsatz als Strukturierungs- und Modellierungsmittel auf dem Weg von informeller Beschreibung zum Programm

nicht: Beschreibung von existierender Software

Modellierung beinhaltet auch Abstraktion

z.B. unpassende Frage: *"Muss im Klassendiagramm auch der Konstruktor in den Methoden aufgelistet werden?"*

Diagramme so zu verstehen:

- Konstruktor nicht aufgelistet: Verhalten wohl Standard, Klasse muss natürlich dennoch einen haben
- Konstruktor aufgelistet: evtl. Besonderheit zu beachten, z.B. bei übergebenen Argumenten, verschiedene Konstruktoren sollen vorhanden sein, etc.

Anwendungsfalldiagramme

definieren Funktionalität (von Teilen) des Systems nach außen hin

Bsp.:

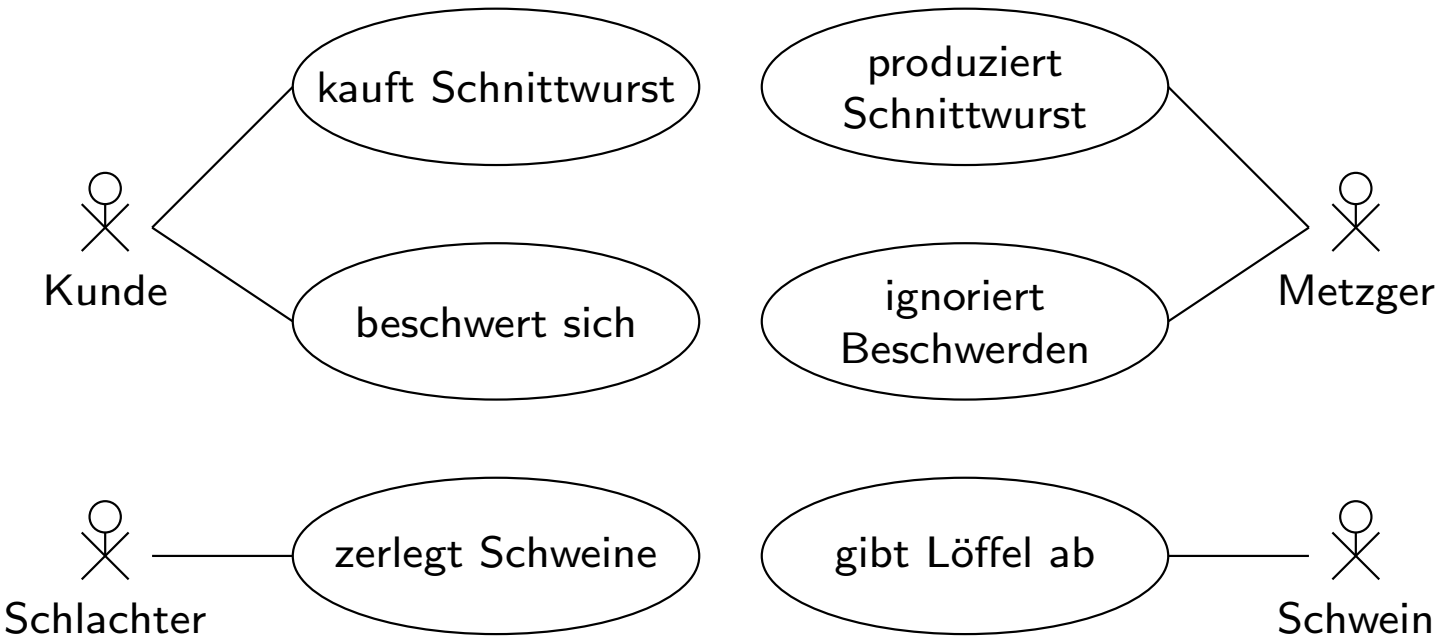
- Was macht Benutzer mit Programm?
- Was macht ein Teil des Systems mit anderem?
- ...

Vorsicht: hohe Variabilität bzgl. Detailliertheit

Modellierung identifiziert *Aktoren* und *Anwendungsfälle*

Anwendungsfalldiagramme – Beispiel

bayerischer Schlachthof mit integrierter Metzgerei



Klassendiagramme

Klasse = gemeinsame Ausprägung verschiedener Objekte

hier nur objekt-orientierte Software-Entwicklung in Java: Klasse = Java-Klasse

Klassendiagramme stellen Beziehungen zwischen einzelnen Klassen dar

Beziehungen spiegeln Zusammenhänge in den Anforderungen wider

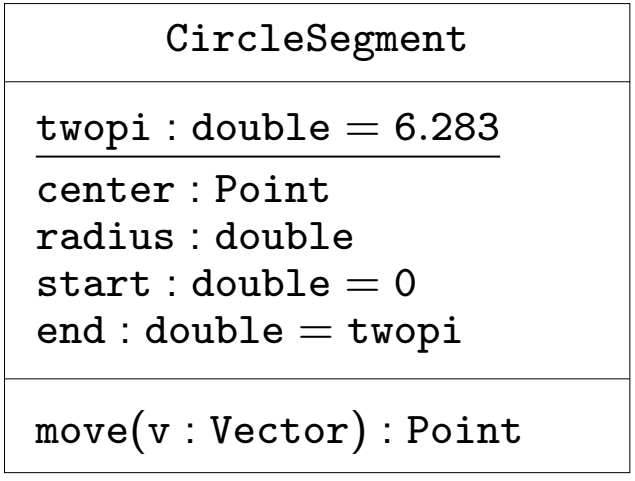
Beziehungen lassen sich oft direkt in Code umsetzen

Klassen

Klassen werden dargestellt als dreigeteiltes Rechteck

- oberer Teil enthält Klassenname
- mittlerer Teil enthält Attribute
- unterer Teil enthält Methoden

Attribute werden mit Typ spezifiert, evtl. auch mit Default-Wert



Methoden werden mit Typ spezifiziert

unterstrichene Attribute/Methoden = Klassenattribute / -methoden

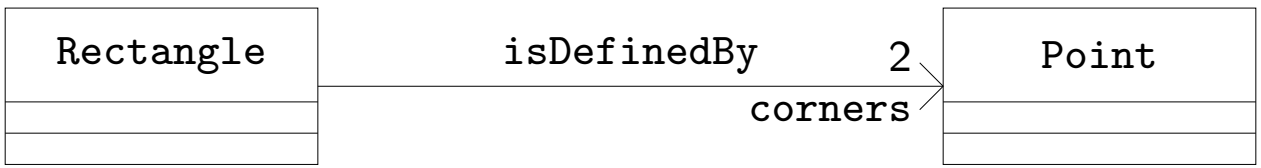
Assoziation

Objekte einer Klasse stehen in Verbindung zu Objekten der anderen Klasse

Multiplizitäten möglich, z.B. `1` oder `1..2` oder `*`

unidirektionale, bidirektionale Assoziation möglich, sogar mehrstellig

optionale Rollennamen verdeutlichen Bedeutung der Verbindung



Auswirkung auf Implementierung: `Rectangle` hat zwei Instanzvariablen vom Typ `Point`

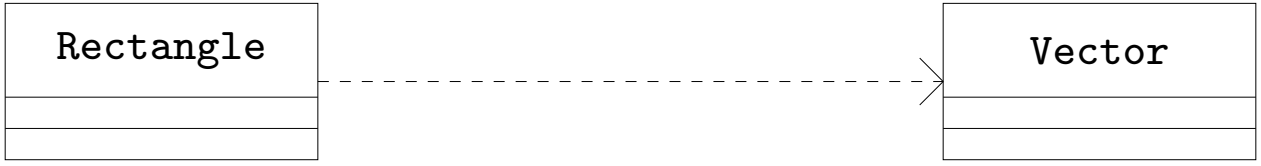
Aggregation

eine Klasse kann Teile einer Entität modellieren, die durch eine andere Klasse modelliert wird

Implementierung z.B. durch Instanzvariablen

Abhängigkeit

sollte sich `Vector` ändern, so muss evtl. auch `Rectangle` geändert werden



liegt z.B. vor, wenn Objekte vom Typ `Vector` in Methoden von `Rectangle` benutzt werden

Vererbung

eine Klasse kann spezieller als eine andere sein, (umgekehrt genereller)

in Java realisiert durch `extends`

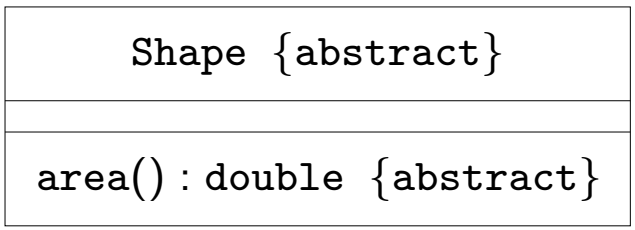
beachte Konventionen zum Vererben und Überschreiben von Instanzvariablen und Methoden

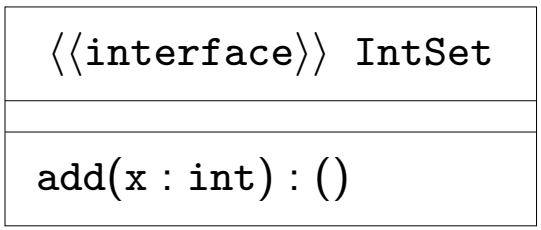
Interfaces und abstrakte Klassen

Interfaces geben Typ aber nicht Implementierung vor

abstrakte Klassen können abstrakte Methoden enthalten

abstrakte Methode: Implementierung abhängig von Untertyp





Methoden müssen in anderen Klassen implementiert werden

Zustandsdiagramme

nicht zu verwechslen mit Objektdiagrammen (= Zustand des Heaps)

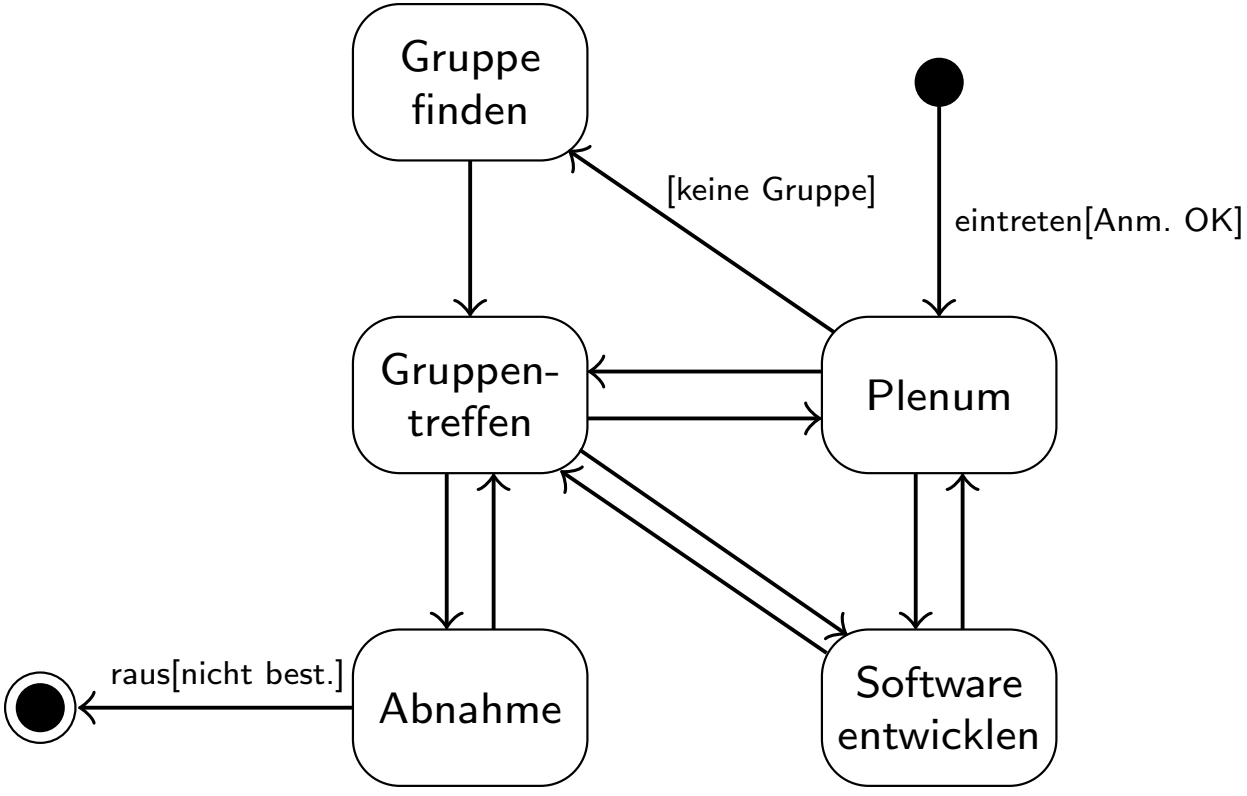
Modellierung des zeitlichen Verhaltens eines Objekts als endlicher Automat

endliche Anzahl von Zuständen kann z.B. durch Datenabstraktion erreicht werden (z.B. Inhalt einer `int`-Variable ignorieren)

zwei ausgezeichnete Zustände für Anfang (Konstruktion) und Ende (Destruktion)

Zustandsdiagramme – Beispiel

Teilnehmer im SEP



Zustandsübergänge z.B. durch Methodenaufrufe realisiert

Sequenzdiagramme

Beschreibung des dynamischen Verhaltens eines Systems

stellen Interaktion von Objekten in zeitlicher Reihenfolge dar

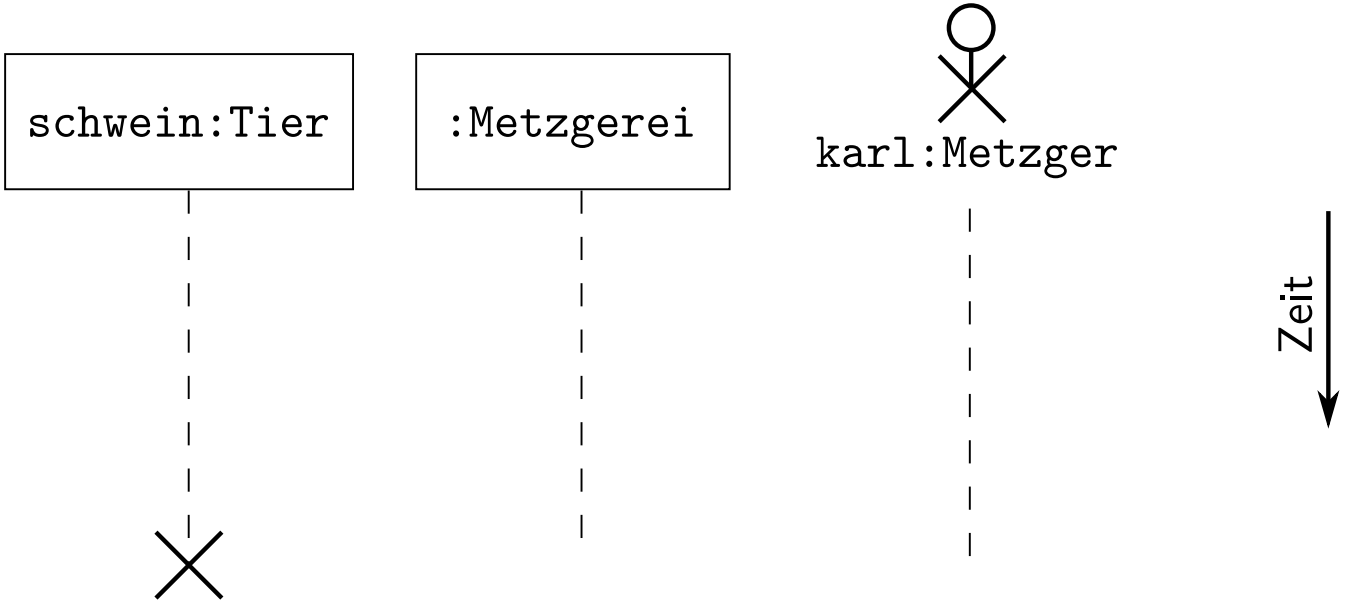
Systemverhalten wird durch Austausch von *Nachrichten* beschrieben

in Java: Nachrichtenaustausch z.B. Methodenaufruf

Zeitlinien

Sequenzdiagramme bestehen aus mehreren Zeitlinien, welche die Lebenszeit von Objekten darstellen

Zeitlinien sind mit Klassennamen und evtl. Objektnamen annotiert



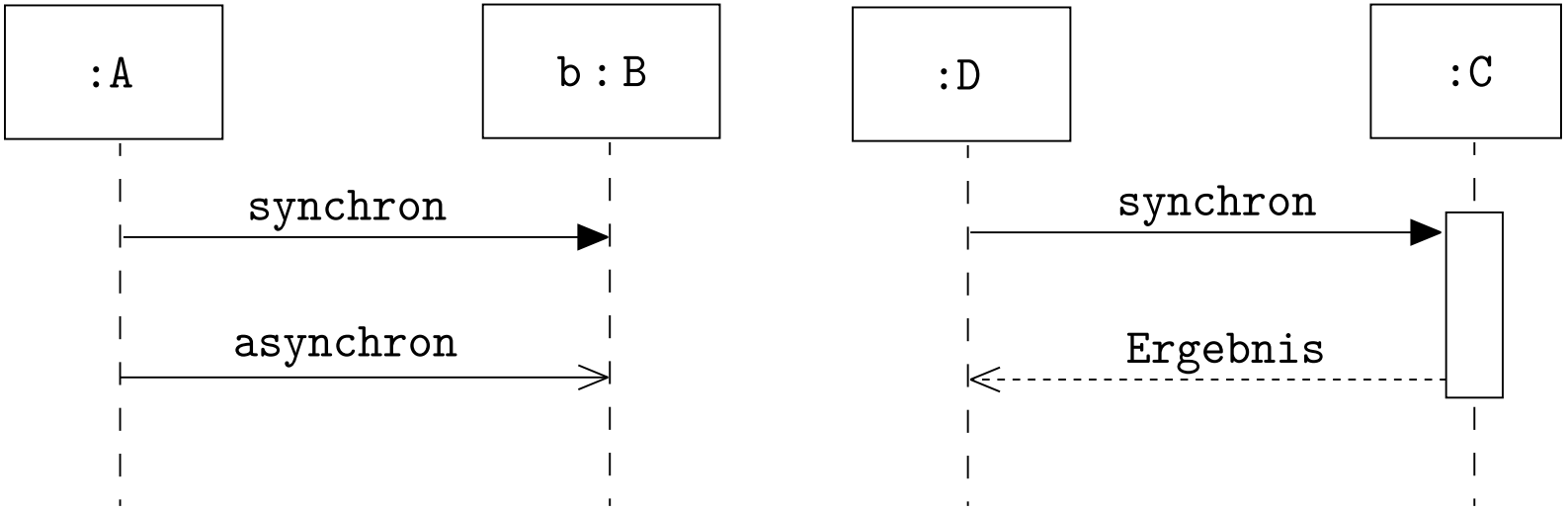
Objektdestruktion mit Kreuz gekennzeichnet

Nachrichten

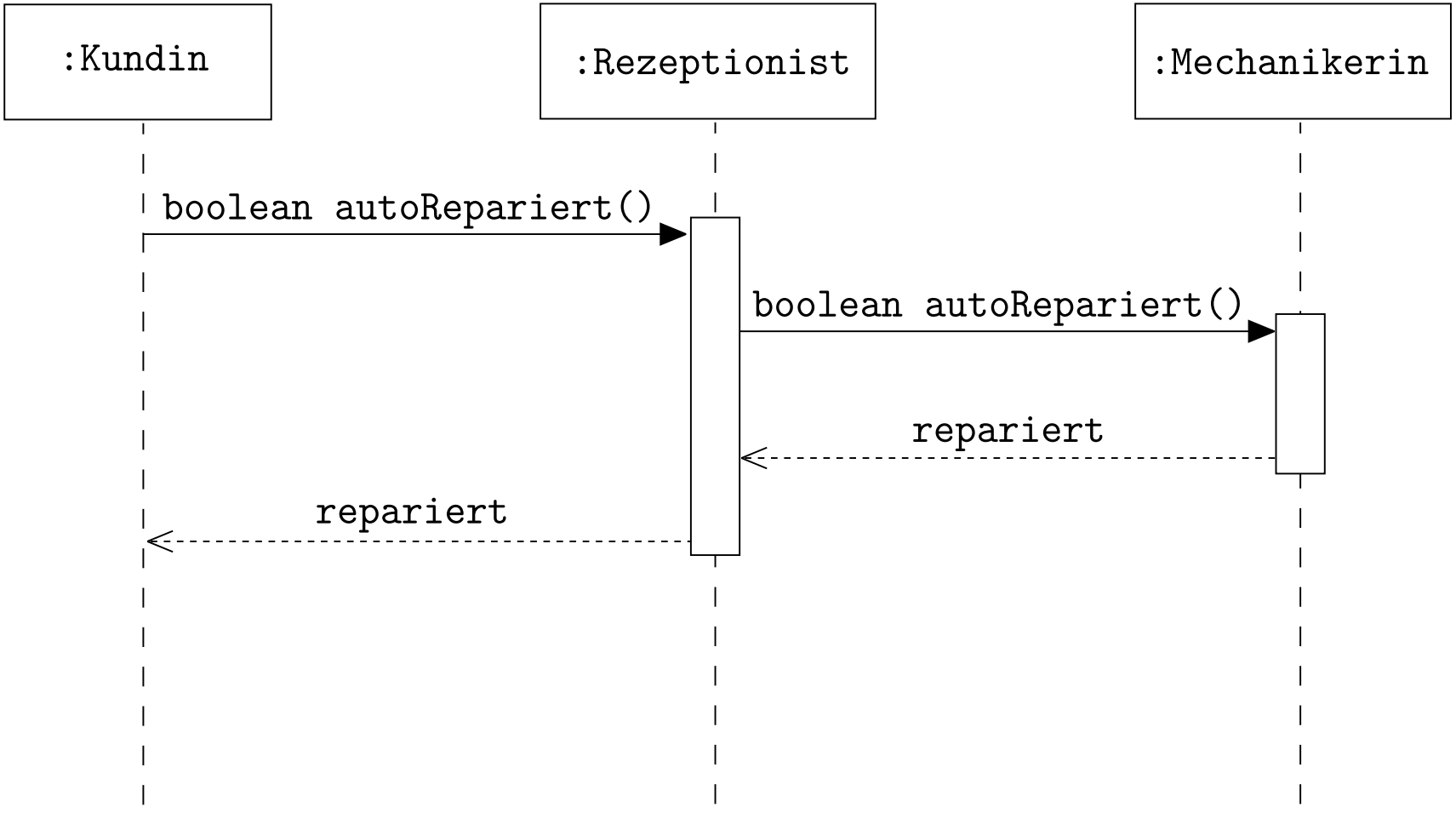
synchrone und asynchrone Nachrichten

bei synchronen Nachrichten optional Rückantwort, die mit einem Rückgabewert beschriftet sein können

Rechtecke auf den Zeitlinien symbolisieren die durch Nachrichten ausgelöste Aktivität



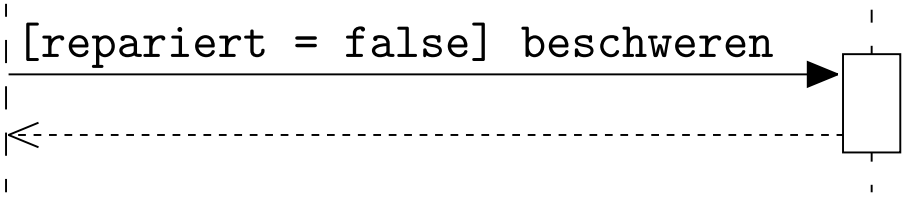
Synchrone Nachrichten – Beispiel



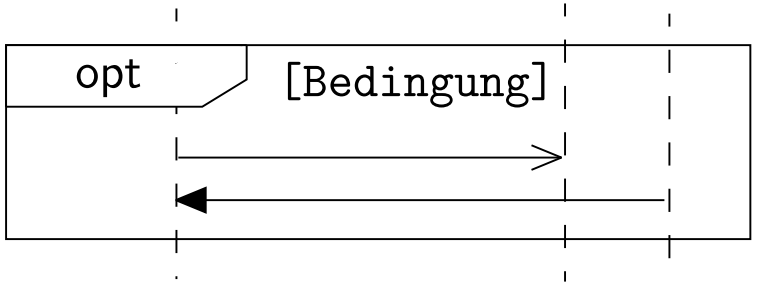
wie in Java übersetzen?

Kontrollstrukturen

Guards Nachricht wird nur geschickt, wenn eine bestimmte Bedingung erfüllt ist.



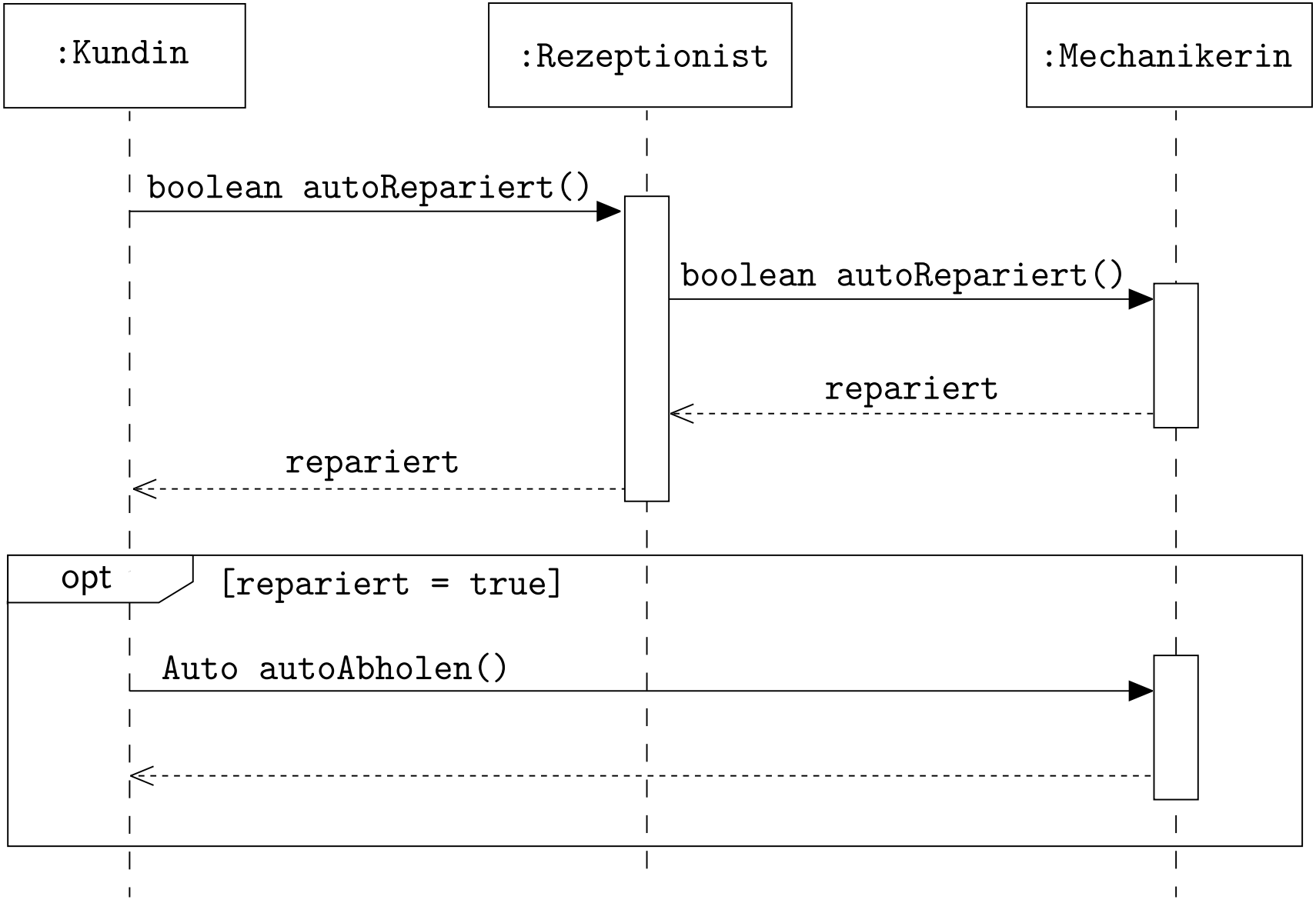
Optionen eine ganze Reihe von Nachrichten wird nur ausgeführt wenn eine Bedingung wahr ist



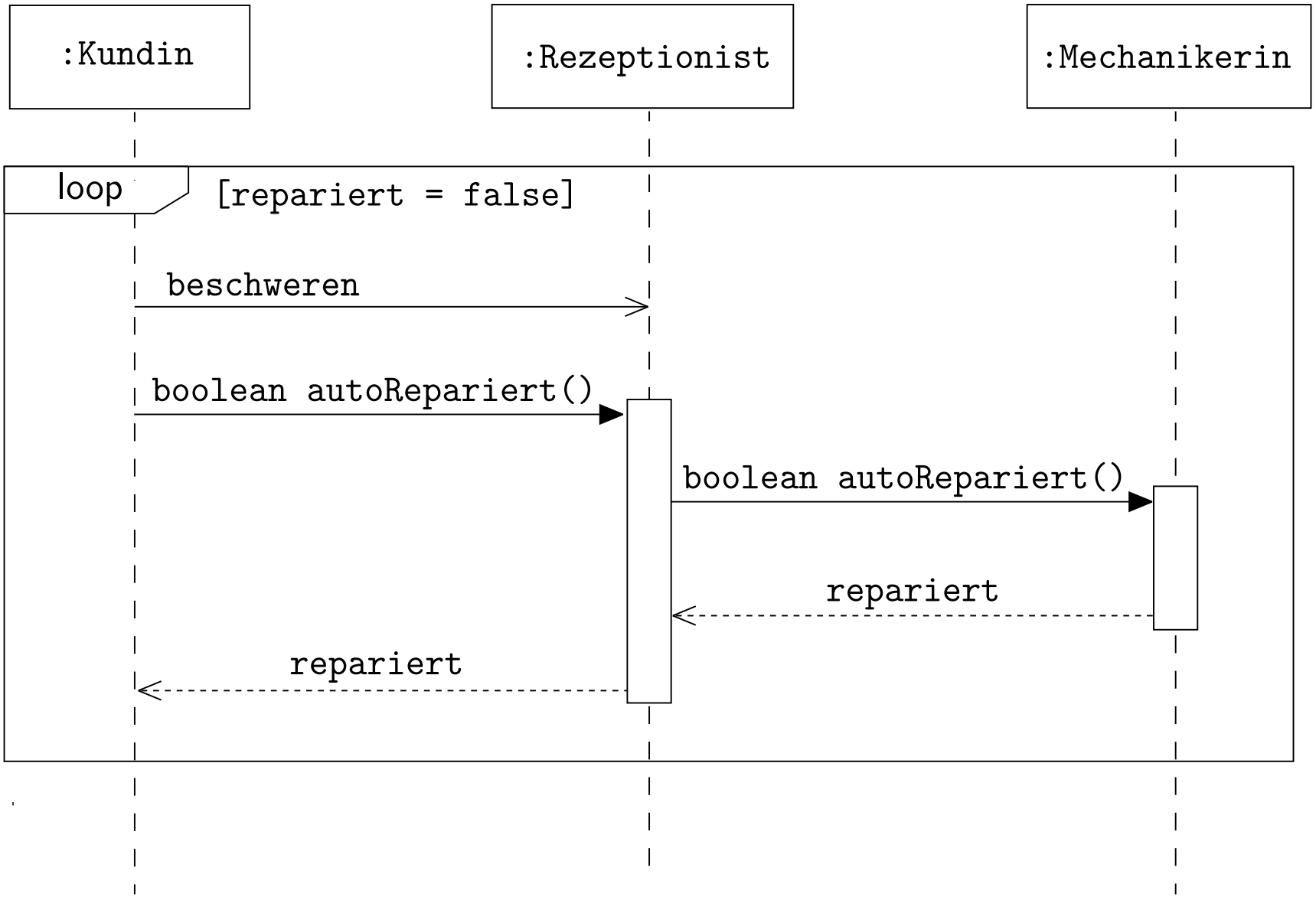
Schleifen wiederhole Nachrichten solange Bedingung wahr ist: „loop“ statt „opt“

viele weitere ...

Optionen – Beispiel



Schleifen – Beispiel



UML-Diagramme im Praktikum

- Teile der Spezifikation in UML gegeben
- Entwicklungsprozess in den Gruppen soll ebenfalls mithilfe von UML erledigt werden
  - Entwurf von detaillierter Architektur mithilfe von Klassendiagrammen, evtl. noch Anwendungsfalldiagrammen
  - Entwurf von Klassen mithilfe von Zustandsdiagrammen
  - Entwurf von Abläufen (z.B. im Server) mithife von Sequenzdiagrammen
  - (Entwurf und Debugging mit Objektdiagrammen)

Links

- Folien der Vorlesung “Objektorientierte Software-Entwicklung”

  `http://www.pst.ifi.lmu.de/Lehre/wise-07-08/oose/material`

- `http://www.uml.org/` – siehe Tutorials

- Bücher

- Tools zur Erstellung von Diagrammen, siehe z.B. “UML-Werkzeug” auf Wikipedia

Software-Entwicklung

Geschichte der Programmierung

Aufgaben von, Anforderungen an Programme mit der Zeit verändert

1. Programmierung über Lochkarten
   z.B. für Rechenaufgaben
2. maschinennahe Programmierung
   auch wichtig wegen Geschwindigkeitsverlusts durch Compilation
3. höhere Programmiersprachen
   geeignet für größere Programme
4. ???
   komplexe Software-Systeme

Die Software-Krise

in den 1960er Jahren begannen Kosten für Software die Kosten für Hardware zu übersteigen

davor: Computer nicht sehr leistungsfähig, keine großen Programme nötig

ab dann: Computer leistungsfähig, Software muss mithalten

Begriff *Software-Engineering* / *Softwaretechnik* wurde geprägt

generelle Ziele der Software-Entwicklung:

- Korrektheit
- Effizienz
- Wartbarkeit
- Wiederverwertbarkeit

Softwaretechnik

beschreibt Prozess des Herstellens von Software

mehr als nur Erstellen eines Programms, sondern u.a.

- Planung
- Analyse
- Entwurf / Modellierung
- Programmierung
- Validierung
- Dokumentation

Planung

Planung eines Softwareprojekts beinhaltet normalerweise u.a.

- Aufwandsschätzung
  - Zeit
  - Personal
  - Budget
  - ...
- Planung der Vorgehensweise

hier keine Modelle der Aufwandsschätzung, aber zeitliche Planung wird von Gruppen erwartet

Zeitliche Planung

beachte zeitlichen Rahmen

- Endabnahme in der Woche vom 9.02.09–13.02.09
  voll funktionsfähiges System, sinnvoll dokumentierter Quellcode, etc.
- Wettbewerb am 5.02.09
- Testabnahme in der Woche 15.12.08–19.12.08
  funktionsfähiges aber nicht unbedingt vollständiges System, sinnvoll dokumentierter Quellcode, etc.

definiert zwei Phasen der Software-Entwicklung

jede Gruppe legt Tutor sobald wie möglich Planung für verbleibenden Zeitraum vor (was wird wann gemacht?)

Vorgehensweisen

werden in Vorgehensmodellen festgelegt

Vorgehensmodelle machen in verschiedener Detailliertheit
Vorgaben zum Ablauf der einzelnen Prozesse (wie, wann)

verschiedene Typen von Vorgehensmodellen, z.B.

- eher Philosophie
  - V-Modell
  - Modellgetriebene Software-Entwicklung
  - Agile Software-Entwicklung
  - Design by Contract
- Prozessmodelle, z.B.
  - Wasserfallmodell
  - Spiralmodell

Vorgehensweisen im Praktikum

nicht jedes Vorgehensmodell in jeder Situation anwendbar

z.B. hier: Kunde nicht wirklich vorhanden (Aufgabensteller, Tutoren?)

Teilnehmer sollen über Existenz von Vorgehensmodellen informiert sein

Vorgehensmodelle im Detail mit Diskussion $\rightsquigarrow$ Vorlesung Softwaretechnik

Gruppen sollen hier vermitteltes Wissen über Vorgehensmodelle einsetzen (z.B. Zerlegung des Entwicklungsprozesses in Teilprozesse)

Agile Software-Entwicklung

kein einzelnes Prozessmodell, sondern Entwicklungsphilosophie

hier nicht unbedingt Empfehlung, schon gar nicht Vorschrift

Philosophie definiert drei aufeinander aufbauende Ebenen

- *agile Werte*
  was bei der Software-Entwicklung wichtig sein sollte
- *agile Prinzipien*
  generelle Vorschriften für alle Teilprozesse um die Werte zu respektieren
- *agile Methoden*
  Beschreibung der Ausführung von Teilprozessen nach den Prinzipien

Agile Werte

- Wichtiger als vordefinierte Prozesse und vorgefertigte Tools sind die beteiligten Individuen und deren Miteinander.
- Wichtiger als ausführliche Dokumentation ist eine funktionierende Software.
- Wichtiger als nach Vertrag zu arbeiten ist die Zusammenarbeit mit dem Kunden.
- Wichtiger als einem Plan zu folgen ist es, flexibel auf Änderungen reagieren können.

allgemein: Verzicht auf starre Regeln

Agile Prinzipien

1. Kunde durch frühes und häufiges Liefern zufrieden stellen
2. sich ändernde Anforderungen sind willkommen
3. funktionierende Software häufig abliefern
4. tägliche Zusammenarbeit zwischen Entwicklern und Managern
5. Projekte von motivierten und zufriedenen Mitarbeitern durchführen lassen
6. Information durch Konversation übermitteln
7. Erfolg wird an funktionierender Software bemessen
8. nachhaltige Entwicklung
9. ständige Aufmerksamkeit auf technische Exzellenz und gutes Design
10. Einfachheit ist essentiell
11. selbst-organisierende Teams erbringen beste Designs, etc.
12. Selbstreflektion im Team in regelmäßigen Abständen mit anschließender Verbesserung

Agile Methoden

Methoden sollen Prinzipien in Entwicklungsprozesse einbringen

Aversion gegen starre Pläne, besser Fokussierung auf das, was gebraucht wird, statt dem, was geplant ist

Beispiele für agile Methoden:

- Programmieren zu zweit
- testgetriebene Entwicklung

Abschluss: agile Software-Entwicklung erfreut sich wachsender Beliebtheit und ist in vielen Fällen erfolgversprechend

Validierung

Erinnerung: Korrektheit von Software ist eines der Ziele ihrer Entwicklung

imminente Frage: wann ist ein Programm korrekt?

offensichtliche Antwort: wenn es den Anforderungen genügt

Antwort nicht sehr befriedigend ("wann genügt ein Programm seinen Anforderungen?")

Ansatz: formalisiere Anforderungen als *Eigenschaften* von Programmen

Der Satz von Rice

Typ einer Eigenschaft ist lediglich *Menge von Programmen*

(Eigenschaft E kodiert als Menge aller Programme, die E haben)

Eigenschaft trivial: $E = \emptyset$ oder $E =$ Menge aller Programme

Satz: Sei E eine nicht-triviale Eigenschaft von Programmen. Dann gibt es keinen Algorithmus, der zu einem vorgegebenen Programm P entscheidet, ob $P \in E$ gilt oder nicht.

Konsequenz: Validierung von Programmen nicht automatisierbar

Validierung

Validierung dennoch wichtig, denn z.B.

- Software soll fehlerfrei sein
- Kunde kauft nicht gerne Katze im Sack

verschiedene Methoden zur Validierung von Software

- Validierung per Hand, z.B. Theorembeweisen
- proof carrying code
- automatische Validierung eingeschränkter Programme, z.B. durch Abstraktion
- Testen
- ...

Testen

Testen = exemplarische (also nicht vollständige) Validierung, nicht nur bzgl. Fehlerfreiheit, sondern auch bzgl. genereller Spezifikation

Vorteil von Testen: nicht eingeschränkt durch Programmgröße

Nachteil: kann nur Fehler aufdecken, aber keine Fehlerfreiheit garantieren

in vielen Vorgehensmodellen: Validierung (z.B. durch Testen) nach Programmierung oder zeitlich unabhängig davon

in testgetriebener Entwicklung: erst Tests definieren, dann Programmcode erzeugen

vorliegende Tests können auch als Dokumentation gesehen werden

Testarten

man unterscheidet drei Arten von Tests bzgl. Kenntnis des Codes:

- white-box test: Code ist bekannt (z.B. wegen Testen nach Programmieren)
- black-box test: Code ist unbekannt, stattdessen nur Spezifikation
- grey-box test: Code ist teilweise bekannt, z.B. weil *noch nicht* geschrieben

man unterscheidet zwei Arten von Tests bzgl. Granularität:

- Systemtests
- unit tests: einzelne Komponenten werden getestet
  Annahme: Gesamtsystem korrekt, wenn alle Komponenten korrekt sind

unit tests haben größere Chance, Fehler aufzudecken, erfordern aber detaillierte Spezifikation

Testgetriebene Entwicklung

benutzt grey-box unit tests

Codegenerierung erfolgt in Iteration von Phasen

1. identifiziere nächste zu integrierende Funktionalität
2. schreibe Tests dafür
3. solange diese Tests fehlschlagen: schreibe/repariere Code dafür
4. vereinfache Gesamtcode durch Abstraktion, Aufräumen, etc.
5. wiederhole Tests

Design by Contract

Design by Contract

Teile das zu entwickelnde Programm in kleine Einheiten (Klassen, Methoden), die unabhängig voneinander entwickelt und überprüft werden können.

Einheiten mit klar definierten Aufgaben und wohldefiniertem Verhalten, z.B.

- Klasse `List`
- Methode `List.add`

Setze größere Komponenten (letztendlich das Gesamtprogramm) aus kleineren zusammen.

Wenn sich alle Teilkomponenten korrekt verhalten, dann auch die zusammengesetzte Komponente.

Entwicklung von Komponenten

Jede Komponente

- hat eine klar definierte Aufgabe.
- stellt ihre Dienste durch eine öffentliche Schnittstelle bereit.
- kann selbst andere Komponenten benutzen.

Komponenten sollen voneinander *unabhängig* sein:

- unabhängige Entwicklung und Überprüfung (z.B. von verschiedenen Teammitgliedern)
- Austauschbarkeit
- interne Änderungen in einer Komponente beeinflussen andere Komponenten nicht negativ

Wie kann man solche Komponenten entwickeln und sicherstellen, dass sich zusammengesetzte Komponenten wunschgemäß verhalten?

Entwicklung von Komponenten

In einer objektorientierten Sprache spielen Klassen die Rolle von Komponenten.

Beim Implementieren einer Klasse sollte man sich genau überlegen

- welche *Leistungen* die einzelnen Funktionen der Klasse erbringen sollen.
- welche *Annahmen* die Funktionen über ihre Argumente und den Zustand der Klasse machen.

Dokumentiere Leistungen und Annahmen der Klasse.

Benutze nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.

Java-Typsystem oft zu schwach, um Annahmen zu garantieren (z.B. Wert von x immer positiv).

Beispiel

```
public class Konto {

   public double getKontostand()

   /**
    *  Wenn vorher getKontoStand() = x
    *  und betrag >= 0,
    *  dann danach getKontoStand() = x + betrag
    */
   public void einzahlen(double betrag)

   /**
    *  Wenn vorher getKontoStand() = x
    *  und x > betrag >= 0,
    *  dann danach getKontoStand() = x - betrag */
   public void abheben(double betrag)
}
```

Entwicklungsprinzip

Benutze immer nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.

Austauschbarkeit Jede Klasse kann durch eine andere mit der gleichen öffentlichen Schnittstelle ersetzt werden, solange diese mindestens die gleichen Leistungen erbringt und nicht mehr Annahmen macht und

unabhängige Entwicklung Teammitglieder einigen sich über Leistungen und Annahmen und können diese dann unabhängig voneinander implementieren.

unabhängige Überprüfung systematisches Testen der dokumentierten Leistungen

Design by Contract

Dieses Entwicklungsprinzip ist unter dem Namen Design by Contract (etwa: Entwurf gemäß Vertrag) bekannt.

Design by Contract enthält eine Reihe von methodologischen Prinzipien zur komponentenbasierten Softwareentwicklung.

Analogie mit Verträgen im Geschäftsleben:

- Die Komponenten schließen untereinander Verträge ab.
- Ein Vertrag beinhaltet das Versprechen einer Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind.
- Setze Programm so aus Komponenten zusammen, dass es richtig funktioniert, wenn nur alle ihre Verträge einhalten.

Verträge

Verträge werden zwischen Klassen abgeschlossen und haben folgende Form:

Wenn vor dem Aufruf einer Methode in einer Klasse eine bestimmte Voraussetzung erfüllt ist, dann wird die Klasse durch die Ausführung der Methode in einen bestimmten Zustand versetzt.

Beispiele

Für jedes Objekt `List l` gilt:

- Nach der Ausführung von `l.add(x)` gilt die Eigenschaft `l.isEmpty() = false`.
- Ist die Voraussetzung `l.size() > 0` erfüllt, so wird die Funktion `l.remove(0)` keine Exception werfen.

Verträge

Ein Vertrag für eine Methode besteht aus:

Vorbedingung: Welche Annahmen werden an die Argumente der Methode und den Zustand der Klasse gemacht.

Nachbedingung: Welche Leistung erbringt die Methode, d.h. welche Eigenschaft gilt nach ihrer Ausführung.

Effektbeschreibung: Welche Nebeneffekte hat die Ausführung der Methode (Exceptions, Ein- und Ausgabe, usw.)

Für uns genügt es, Verträge informell zu beschreiben.

Beispiel – Klasseninvarianten

Klasseninvariante Zu jedem Zeitpunkt erfüllt der Zustand der Klasse eine bestimmte Eigenschaft (die *Invariante*).

(Achtung: "zu jedem Zeitpunkt" fragwürdig, warum?)

Beispiel

- Klasse `Universitaet` mit privatem Feld

  ```
  private Map<Student, Set<Seminar>> seminareProStudent;
  ```

- Um herauszufinden, welche Studenten ein bestimmtes Seminar besuchen wird, muss man alle Studenten durchgehen.
- effizienter: Daten auch noch nach Seminaren indizieren.

  ```
  private Map<Seminar, Set<Student>> studentenInSeminar;
  ```

- Invariante: Die beiden Felder `seminareProStudent` und `studentenInSeminar` enthalten die gleichen Daten.

Beispiel – Klasseninvarianten

Implementierung von Klasseninvarianten

- Konstruktoren haben Invariante als Nachbedingung.

  Im Beispiel: `seminareProStudent` und `studentenInSeminar` beide anfangs leer

- Alle Methoden haben Invariante sowohl als Vor- als auch als Nachbedingung: Wenn die Invariante vor der Ausführung gilt, dann auch danach.

  Im Beispiel: Alle Methoden können annehmen, dass `seminareProStudent` und `studentenInSeminar` die gleichen Daten enthalten, müssen aber auch sicherstellen, dass das nach ihrer Ausführung immer noch gilt.

Überprüfung von Bedingungen mit `assert`

Bedingungen und Invarianten können mit der Java-Instruktion `assert` getestet werden.

Die Instruktion

```
assert test : errorValue;
```

prüft, ob der boolesche Ausdruck `test` zu `true` auswertet, und wirft eine `AssertException`, wenn das nicht der Fall ist.

äquivalent:

```
if (!test) {
  throw new AssertionError(errorValue)
}
```

Warum benutzt man nicht die `if`-Abfrage?

Überprüfung von Bedingungen mit `assert`

`assert`-Instruktionen werden nur zum Testen benutzt.

- Standardmäßig werden Assertions ignoriert, d.h. sie haben keinen Einfluss auf die Progammgeschwindingkeit.
- Assertions müssen beim Programmstart ausdrücklich eingeschaltet werden.

  ```
  java -ea Main
  ```

  (`-ea` für „enable assertions“)

⇒ Das Programm sollte sich mit und ohne Assertions gleich verhalten.

`assert`-Instruktionen dienen auch der Dokumentation von Verträgen.

Assertions – Beispiele

Nachbedingung prüfen

```
class NichtUeberziehbaresKonto {
  private double kontoStand;
  ...
  public void abheben(double betrag) {
     ...
     assert (kontoStand >= 0) : "Interner Fehler " +
                                "Argument nicht richtig geprüft," +
                                "Konto überzogen";
  }
}
```

Unerreichbare Programmteile

```
  ...
  for (...) {
    if (...) return;
  }
  assert false; // Diese Instuktion sollte unerreichbar sein
```

Assertions – Beispiele

Argumente von öffentlichen Methoden nicht durch Assertions überprüfen.

öffentliche Methoden müssen Argumente immer selbst validieren

```
public void setX(int x) {
  if ((0 > x) || (x >= xmax)) {
    throw new IndexOutOfRangeException();
  }
  ...
}
```

Programm soll sich gleich verhalten, egal ob Assertions eingeschaltet sind oder nicht.

Assertions – Beispiele

Argumente von privaten Methoden

Private Methoden müssen falsche Argumente nicht immer mit Exceptions behandeln.

Da sie nur von der Klasse selbst benutzt werden können, kann man intern sicherstellen, dass sie nur mit guten Argumenten aufgerufen werden.

Dies kann man mit `assert` testen.

```
private void setX(int x) {
  assert (0 <= x) && (x < xmax);
  ...
}
```

Systematisches Testen

Unit Testing

Objektorientierte Entwicklung

Entwicklung von vielen unabhängigen Einheiten (Klassen, Methoden), aus denen das Gesamtprogramm zusammengesetzt wird.

Ziel: Wenn sie nur alle Einheiten korrekt verhalten, dann verhält sich das Programm wie gewünscht.

Unit Testing

- systematisches Testen der einzelnen Einheiten
- zwingt Entwickler zu planen und über das gewünschte Verhalten der Programmeinheiten nachzudenken
- unterstützt Design by Contract: teste stichprobenartig, dass alle Klassen ihre Verträge einhalten

Unit Testing mit JUnit

Was wird getestet

- public/protected-Methoden
- private-Methoden können mit `assert` getestet werden.

Wie wird getestet

- Tests sind vom Programm separate Klassen (man kann Tests als Teil der Dokumentation auffassen).
- Darin wird eine beliebige Anzahl von Testfunktionen implementiert.
- Testfunktionen mit Java-Annotationen gekennzeichnet.
- Die Tests werden dann durch ein bestimmtes Programm ausgeführt und die Ergebnisse werden ausgewertet (üblicherweise durch Tools wie Eclipse oder NetBeans).

Java-Annotationen

Mit Annotationen kann man Java-Programme mit nützlichen Zusatzinformationen versehen werden.

In Java sind sieben Arten von Annotationen eingebaut, z.B. `@Deprecated`, `@Override`, `@SuppressWarnings`

Man kann eigene Annotationen definieren.

In JUnit 4 sind zum Beispiel `@Test`, `@Before`, `@After`, `@BeforeClass`, `@AfterClass` und `@Ignore` definiert.

Annotationen – Beispiel

Kennzeichnung von Klassen/Methoden, die nicht mehr benutzt werden sollen und nur aus Kompatibilitätsgründen noch vorhanden sind.

```
@Deprecated
public class A {
   ...
}
```

Compiler erzeugt Warnung, wenn die Klasse `A` benutzt wird.

Annotationen – Beispiel

Kennzeichne Funktionen, die eine bereits existierende Funktion in einer Oberklasse überschreiben mit `@Override`.

```
public class A {
  public void f(List l) {
    ...
  }
}

public class B extends A {
  @Override
  public void f(List l) {
    ...
  }
}
```

Annotationen – Beispiel

`@Override` hilft häufige Fehler zu finden:

```
public class A {
  public void f(List l) {
    ...
  }
  private void g() {
    ...
  }
}
```

```
public class B extends A {
  @Override
  public void f(LinkedList l) {
      ...
  }
  @Override
  public void g() {
      ...
  }
}
```

Compilerfehler: Überladen statt Überschreiben.

JUnit 4

Tests sind in separat vom eigentlichen Programm in eigenen Klassen implementiert (z.B. eine Testklasse für jede zu testende Klasse)

Testklasse kann eine beliebige Anzahl von Tests implementieren.

Einzelne Tests werden durch Methoden in der Testklasse implementiert:

- durch Annotation `@Test` als Test gekennzeichnet
- beliebiger Methodenname
- `public` und ohne Argumente

Die Tests in einer Testklasse werden automatisiert ausgeführt und ein Gesamtergebnis angezeigt.

Testfunktionen

Allgemeine Funktion von Testfunktionen

1. Konstruktion der zu testenden Situation
   - Objekte der zu testenden Klassen erzeugen
   - Objekte in den gewünschten Zustand versetzen
2. Überprüfung des erwarteten Ergebnis
   - Funktionen `assertTrue(test)`, `assertEquals(x, y)`, `...` in `org.junit.Assert`.
   - Ist das erwartete Ergebnis eine Exception, kann man das in der Annotation angeben:
     `@Test(expected = ExceptionName.class)`

JUnit 4 – Beispiel

```
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class BoardTest {
  @Test
  public void testFood1() {
    BoardServerToClient b = new BoardServerToClient(100, 100);
    int x = 94, y = 23, a = 34;
    b.setFood(x, y, a);
    assertEquals(b.getFood(x,y), a);
  }

  @Test(expected = java.lang.IndexOutOfBoundsException.class)
  public void testFood2() {
    BoardServerToClient b = new BoardServerToClient(100, 100);
    int x = 100, y = 23, a = 34;
    b.setFood(x, y, a);
  }
}
```

JUnit 4 – Ausführen der Tests

Eclipse, NetBeans, ... bieten bequeme Möglichkeiten, Tests ablaufen zu lassen und Ergebnisse anzuzeigen.

Tests direkt ausführen:

- `junit.jar` von `www.junit.org` beziehen und zum Classpath hinzufügen.
- Programm und Testklassen kompilieren.
- Ausführung der Tests mit

```
java org.junit.runner.JUnitCore BoardTest
```

```
JUnit version 4.3.1
..E
Time: 0.028
There was 1 failure:
1) testFood2(BoardTest)
java.lang.AssertionError:  (Stack backtrace)

FAILURES!!!
Tests run: 2,  Failures: 1
```

Weitere Annotationen

Eine Testklasse enthält oft viele ähnliche Testfälle.

Programmtext, der vor und nach jedem Test ausgeführt werden soll, kann in Methoden mit den Annotationen `@Before` und `@After` geschrieben werden.

- Alle mit `@Before` markierten Methoden werden vor *jedem* Test aufgerufen (in unbestimmter Reihenfolge).
- Alle mit `@After` markierten Methoden werden nach *jedem* Test aufgerufen (in unbestimmter Reihenfolge).

Beispiel – @Before

```
public class BoardTest {
  private BoardServerToClient b;

  @Before public void initBoard() {
    b = new BoardServerToClient(100, 100);
  }

  @Test public void testFood1() {
    int x = 94, y = 23, a = 34;
    b.setFood(x, y, a);
    assertEquals(b.getFood(x, y), a);
  }

  @Test(expected = java.lang.IndexOutOfBoundsException.class)
  public void testFood2() {
    b.setFood(100, 23, 34);
  }
}
```

Einmaliges Einrichten der Testumgebung

In manchen Fällen (z.B. teure `@Before`-Funktionen) ist es günstig, eine Testsituation einmal einzurichten und dann für alle Testfälle zu benutzen.

Statische Methoden mit Annotationen `@BeforeClass` und `@AfterClass`.

- Alle mit `@BeforeClass` markierten Methoden werden *einmal* vor den Tests ausgeführt.
- Alle mit `@AfterClass` markierten Methoden werden *einmal* nach den Tests ausgeführt.

Entwurf von Tests

Was sollte man mit Unit-Tests prüfen?

- Methoden sowohl mit erwarteten als auch unerwarteten Eingaben testen
- Rand- und Ausnahmefälle prüfen
- Werden Fehler richtig behandelt?
- Schreibe Tests, um zu verhindern, dass einmal aufgetretene Fehler nochmal auftreten (z.B. durch unvorsichtiges Revert).

Tests sollten einen (gedachten) Vertrag möglichst gut stichprobenartig überprüfen.

Exceptions

Fehler abfangen

zur Laufzeit auftretene Fehler sollen Programm nicht terminieren

- ungültige Eingabe kann wiederholt werden lassen
- Fehler in einem Teil muss andere Teile nicht berühren
- plötzliche Termination evtl. fatal
- ...

gebraucht: Mechanismus, um Fehler zur Laufzeit abzufangen und zu behandeln

muss damit natürlich Teil der Sprache werden

Was ist eigentlich *Fehler*? Allgemeiner: *Ausnahme* / *Exception*

Mechanismus

Ausnahmen werden

- geworfen: signalisiert das Auftreten, Ausführung des Programms wird unterbrochen
- gefangen: anderer Teil des Programms übernimmt Ausführung, normalerweise Fehlerbehandlung
- deklariert: Compiler soll Nicht-Abbruch garantieren können

verschiedene Ausnahmen erfordern i.A. verschiedene Behandlungen

Mechanismus für Verschiedenheit in Java vorhanden:
Klassenhierarchie mit Vererbung

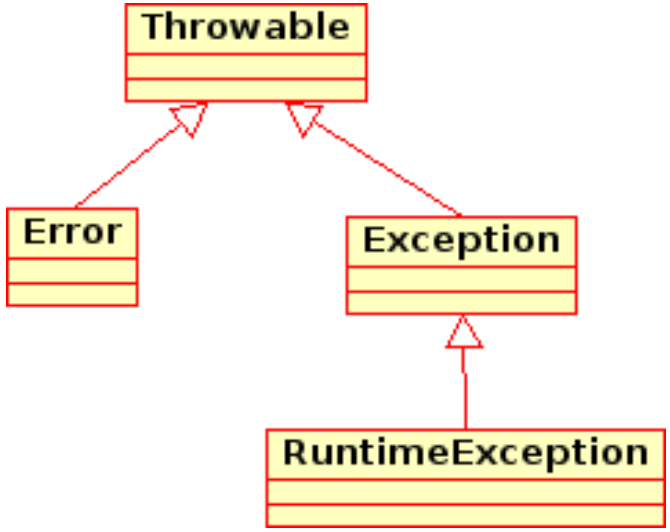
Fehler- und Ausnahmenmodellierung

- `Throwable`: formalisiert allgemein Ausnahmen
- `Error`: Abnormalitäten, die normalerweise zu Programmabbruch führen

  sollten nicht gefangen werden
- `Exception`: unerwartete / unerwünschte Situationen

  sollten gefangen werden
- `RuntimeException`: Fehler in der Programmlogik

  sollten erst gar nicht auftreten, also auch nicht gefangen werden



RuntimeExceptions

typische RuntimeExceptions: `IndexOutOfBoundsException`, `NullPointerException`, etc.

Auftreten kann eigentlich vorher ausgeschlossen werden

sollten nicht gefangen werden, denn was würde Fehlerbehandlung schon besser machen?

RuntimeExceptions sind *unchecked*: müssen im Gegensatz zu Unterklassen von `Exception` nicht deklariert werden

Idee dahinter: Laufzeitfehler sind unvorhergesehen, können praktisch in jeder Methode auftreten

Ausnahmen in Java

Ausnahmen werfen mit `throw` e, wobei e vom Typ `Throwable` (besser: `Exception`)

Ausnahmen fangen:

$$\texttt{try}\ c\ \texttt{catch}\ (E_1\ e_1)\ c_1\ \ldots\texttt{catch}\ (E_1\ e_1)\ c_n$$

Block c wird ausgeführt

beim Auftreten einer Ausnahme e wird kleinstes i gesucht, sodass e Typ E_i hat

Block c_i wird ausgeführt

Finally

Programmfluss kann durch Ausnahmen unterbrochen werden, *bevor* essentieller Code ausgeführt wurde; Bsp.: `EOFException` beim Lesen aus einer Datei

essentieller Code (z.B. Datei schließen) kann dennoch automatisch ausgeführt werden (auch ohne Exception!)

```
FileReader reader;
try {
   reader = new FileReader("meineLieblingsDatei.txt");
   while (true) {
     reader.read();
   }
} catch (FileNotFoundException e) {
   System.out.println("Gibt's nicht. Schade.");
} catch (IOException e) {
   System.out.println("Heute genug getan.");
} finally {
   reader.close();
}
```

Ausnahmen deklarieren

kann eine Exception, die nicht RuntimeException ist, in einer Methode auftreten, so muss dies deklariert werden

```
class NichtsFunktioniert extends Exception { ... }

class A {
  ...
  public void probierMal() throws NichtsFunktioniert {
    ...
    throw new NichtsFunktioniert();
    ...
  }
  ...
}
```

nicht nötig, falls Ausnahme innerhalb der Methode gefangen wird

Zu beachten

- Ausnahmen selbst definieren falls nötig
- `try`-Blöcke verlangsamen Code nicht, auftretende Ausnahmen aber schon
- Ausnahmen nicht zur Steuerung des Kontrollflusses (wie z.B. `break`) einsetzen
- Ausnahmen sind wirklich Teil der Sprache: können z.B. auch in `catch`-Teilen auftreten
- Ausnahmen so speziell wie möglich abfangen und wirklich behandeln, nicht
  `catch (Exception e) { e.printStackTrace(System.err); }`
- Ausnahmen in JavaDoc-Dokumentation nicht vergessen

Generics

Typen und Klassen

Erinnerung: jedes Objekt hat einen *Typ* = Klasse

Typsystem wird benutzt, um Fehler zur Compilezeit anzufangen, z.B. Zugriff auf nicht-definierte Instanzvariablen

Nachteil von Typ = Klasse: allgemeinere Funktionalitäten können nicht mit Typsystem modelliert werden

seit Java 1.5: Erweiterung des Typsystems um *Generics* = parametrisierte Typen

Beispiel: Datenstrukturen

Aufgabe: verwalte zu einem Spielbrett die Menge aller darauf angemeldeten Ameisenstämme

benötigt Datentyp "Menge von Ameisenstämmen"

kann z.B. durch `Ant[][]` modelliert werden, hat aber Nachteile:

- schlechte Zugriffszeit beim Einfügen und Suchen
- keine symbolischen Namen
- ...

bessere Ansätze siehe Vorlesung Algorithmen und Datenstrukturen

hier: besserer Ansatz erfordert eigene Klasse mit Methoden zum Einfügen, Suchen, etc.

was macht man, wenn man auch noch Mengen von anderen Objekten verwaltet möchte?

Casting

bis Java 1.4: einzig Array-Typ `[]` war generisch (aber nicht typsicher!), z.B. in

```
String[] namen = new String[10];
```

selbstverständlich:

- nur Strings können eingetragen werden
- Zugriffe liefern Objekte vom Typ `String` (oder `null`)

nicht so bei anderen Datenstrukturen, z.B.

```
List namen = new LinkedList();
namen.add("Hans Wurst");
String name = (String) namen.get(0);
```

Nachteile:

- Datenstrukturen können Elemente verschiedener Typen halten
- Laufzeitfehler, falls Cast nicht möglich
- fehleranfälliger Code (Cast vergessen)

Generics

ab Java 1.5: Typen können parametrisiert werden (sind generisch)

Bsp: nicht Typ *Liste*, sondern *Liste von Strings*

Notation in Java: `List<String>`, z.B.

```
List<String> namen = new LinkedList<Namen>();
namen.add("Hans Wurst");
String name = namen.get(0);
```

offensichtliche Vorteile, insbesondere Fehlerabfang zu Compilezeit

Variablen für Parameter

Verwendung von Objekten generischen Typs normalerweise mit konkretem Parametertyp

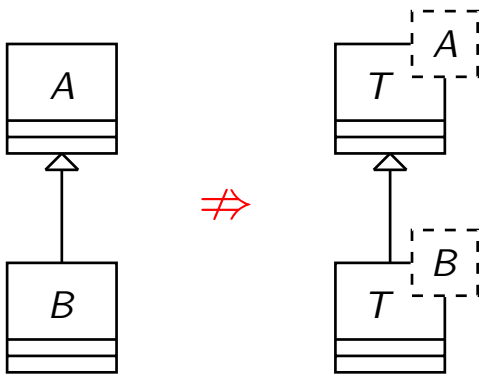
aber Typvariablen nötig für Definition; soll ja mit allen (oder bestimmten) Typen benutzt werden können, z.B.

- *Listen von Objekten* (beliebigen aber gleichen Typs)
- *Listen von Zahlen*, also sowohl `Integer`, als auch `Double`, etc.

```
public interface List<E> extends Collection<E> {
  ...
  boolean add(E o) {
    ...
  }
}
```

Generics und Vererbung

Achtung! Typkontexte erhalten Vererbungsbeziehung nicht




```
class B extends A { ... }
class C extends A { ... }

  ...
  Vector<B> bVector = new Vector<B>();
  bVector.add(0, new B());
  Vector<A> aVector = bVector;
  aVector.add(1, new C());
```

Grund: Zuweisungskompatibilitäten

Wildcards und Kovarianz

nicht weiter verwendete Typparameter können auch mit der Wildcard ? gekennzeichnet werden

```
Vector<B> bVector = new Vector<B>();
Vector<?> aVector = bVector;
```

Kovarianz = Einschränkung auf Typ und seine Untertypen

```
class T<E extends Comparable<E>> { ... }
class U<? extends A & B> { ... }
```

Zu beachten

- generische Typen sind (fast) *first-class citizens*, können also auch z.B. in Methodendefinitionen verwendet werden
- Arrays über generischen Typen können nicht angelegt werden, verwende stattdessen z.B. `ArrayList`
- schreibende Zugriffe bei kovarianten Wildcardtypen sind unzulässig
- in Kovarianz `extends` sowohl bei Vererbung wie auch Implementation von Interfaces
- Konstruktoren von Typargumenten sind nicht sichtbar

```
class T<E> {
  T() {
    ...
    new E();   // verboten
    ...
  }
}
```

Objektserialisierung

Serialisierung von Objekten

Umwandlung des Objektzustandes in einen Strom von Bytes, aus dem eine Kopie des Objekts zurückgelesen werden kann.

Serialisierung in Java

- einfacher Mechanismus zur Serialisierung von Objekten
- eigenes Datenformat

Anwendungen

- Abspeicherung von internen Programmzuständen
- Übertragung von Objekten zwischen verschiedenen JVMs, z.B. über ein Netzwerk

Aber: Zum Austausch von Daten mit anderen Programmen sind Standardformate (z.B. XML) oder speziell definierte Datenformate besser geeignet.

Serialisierung in Java

Java stellt einen Standardmechanismus zur Serialisierung von Objekten zu Verfügung.

Ablauf der Serialsierung eines Java-Objekts:

1. Metadaten, wie Klassenname und Versionsnummer, in den Byte-Strom schreiben
2. alle nichtstatischen Attribute (`private`, `protected`, `public`) serialisieren
3. die entstehenden Byte-Ströme in einem bestimmten Format zu einem zusammenfassen

Bei der Deserialisierung wird der Datenstrom eingelesen und ein Java Objekt mit dem gespeicherten Zustand erzeugt.

Serialisierung in Java

Kennzeichnung von serialisierbaren Objekten: Klasse implementiert das (leere) Interface `java.io.Serializable`.

Attribute einer serialisierbaren Klasse sollten Basistypen oder serialisierbare Objekte sein (sonst Laufzeitfehler bei Serialisierung).

Gründe für Kennzeichnungspflicht:

- Sicherheit, z.B. Zugriffsbeschränkung auf `private`-Attribute
- Serialisierbarkeit soll aktiv vom Programmierer erlaubt werden

Serialisierung von Objekten

Objekte schreiben

```
FileOutputStream f = new FileOutputStream("datei");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject(new Integer(3));
s.writeObject("Text");
s.flush();
```

Objekte lesen

```
FileInputStream in = new FileInputStream("datei");
ObjectInputStream s = new ObjectInputStream(in);
Integer int = (Integer)s.readObject();
String str = (String)s.readObject();
```

Serialisierung von Objekten

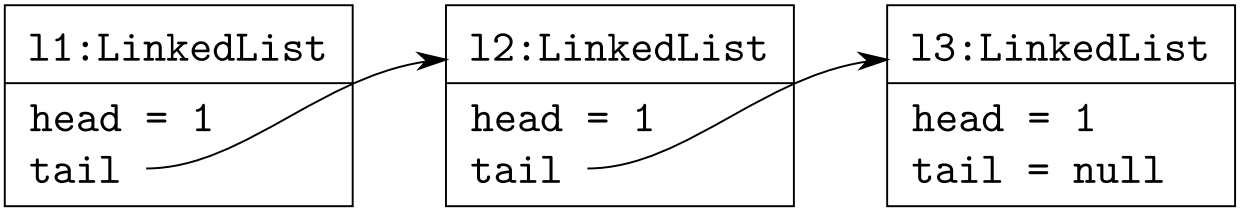
Binär kodierte Objekte in „`datei`“

```
aced0005737200116a6176612e6c616e  |....sr..java.lan|
672e496e74656765721 2e2a0a4f78187  |g.Integer.......|
380200014900057616c756578720010  |8...I..valuexr..|
6a6176612e6c616e672e4e756d626572  |java.lang.Number|
86ac951d0b94e08b0200007870000000  |..........xp....|
037400045465874                   |.t..Text|
```

Serialisierung – Beispiel

```
class LinkedList implements Serializable {
  private int head;
  private LinkedList tail;
  //...
}
```

Attribute (`head` und `tail`) sind serialisierbar.



Durch `s.writeObject(l1)` wird die gesamte Liste seralisiert, also auch `l2` und `l3`.

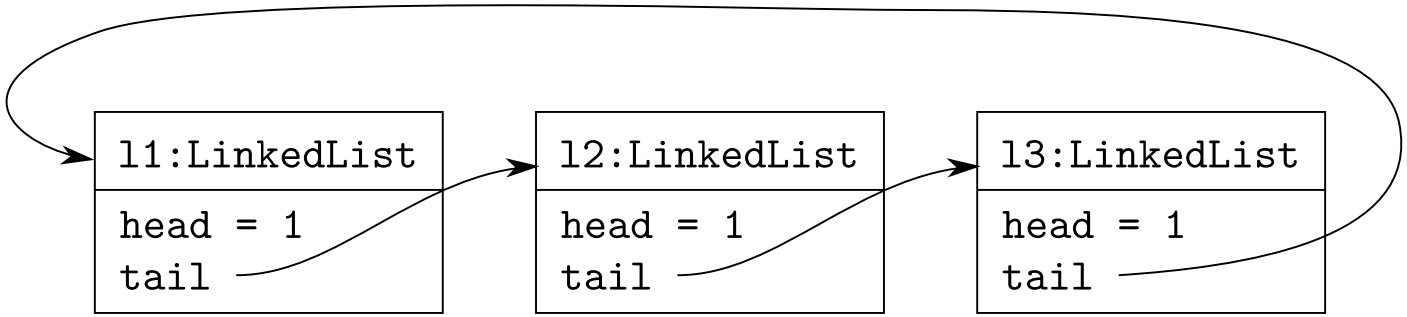
Serialisierung – Beispiel

Serialisierung speichert den Objektgraphen.

Kommt ein Objekt bei der Serialisierung eines anderen Objekts mehrmals vor, so wird nur eine Kopie gespeichert.

Durch Serialisierung und Deserialisierung erhält man eine Kopie der Datenstruktur mit dem gleichen Objektgraphen. Identitätsgleichheit (==) von Objekten im Objektgraph bleibt erhalten.

Beispiel Rückwärtszeiger in einer verketteten Liste werden richtig wiederhergestellt.



Transient

Attribute, die nicht serialisiert werden sollen, können als `transient` markiert werden.

Beipiele

- rekonstruierbare Daten, z.B. Caches oder andere aus Effizienzgründen gespeicherte Daten
- nichtserialisierbare Felder, z.B. Threads
- sensitive Daten

```
public class Account {
  private String username;
  private transient String password;
  //...
}
```

N.B.: sensitive Daten wie Passwörter sollten ohnehin möglichst nicht im Hauptspeicher gehalten werden

Anpassen der Serialisierungsprozedur

Serialisierungsmethoden können angepasst werden.

- Schreiben von zusätzlichen Daten, z.B. Datum
- wiederherstellen von transienten und nichtserialisierbaren Feldern, z.B. Wiederherstellung von Caches, Neustart von Threads

Dazu müssen in der serialisierbaren Klasse zwei Methoden mit folgender Signatur geschrieben werden:

```
private void writeObject(ObjectOutputStream oos)
  throws IOException

private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException
```

Anpassen der Serialisierungsprozedur – Beispiel

```
private void writeObject(ObjectOutputStream oos)
  throws IOException {
  // zuerst die Standardserialisierung aufrufen:
  oos.defaultWriteObject();
  // zusätzliche Daten schreiben
  oos.writeObject(new java.util.Date());
}

private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {
  // zuerst die Standarddeserialisierung aufrufen:
  ois.defaultReadObject();
  // zusätzliche Daten lesen:
  date = (Date)ois.readObject();
  // mit transient markierte Felder wiederherstellen
  ...
}
```

Versionsnummern

Serialisierte Objekte haben eine Versionsnummer. Objekte mit falscher Versionsnummer können nicht deserialisiert werden.

Versionsnummer kann als statisches Attribut definiert werden:

```
public static long serialVerUID = 1L
```

Ist keine Nummer angegeben, so benutzt Java einen Hashwert, der sich u.A. aus den *Namen* der Klassenattribute errechnet

- $\rightsquigarrow$ Änderungen an der Klasse (z.B. Hinzufügen einer Methode) machen serialisierte Daten inkompatibel.
- verhindert vesehentliches Einlesen inkompatibler Daten

Durch Angabe einer `serialVerUID` kann man Kompatibilität steuern, evtl. mit eigener `readObject`-Funktion.

Serialisierbarkeit und Vererbung

Alle Unterklassen einer serialisierbaren Klasse sind serialisierbar.

Kann Serialisierung verhindern mit:

```
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException {
        throw new NotSerializableException();
}
private void writeObject(ObjectOutputStream stream)
    throws IOException {
        throw new NotSerializableException();
}
```

Serialisierbarkeit und Vererbung

Ist eine Oberklasse einer serialisierbaren Klasse *nicht* serialisierbar, so werden ihre privaten Felder *nicht* serialisert.

Beim Deserialisieren wird der Konstruktor ohne Argumente der ersten nichtserialisierbaren Oberklasse aufgerufen (ein Konstruktor ohne Argumente muss existieren!).

`readObject` und `writeObject` müssen geeignet implementiert sein, damit der Zustand der Oberklasse beim Deserialisieren wiederhergestellt wird.

Graphical User Interfaces

Toolkits

es gibt verschiedene Toolkits (= Sammlungen von Klassen, Anbindungen an Betriebssystem, etc.) für grafische Oberflächen

- Abstract Windows Toolkit (AWT)
- Swing
- Standard Widget Toolkit (SWT)
- Gimp's Widget Toolkit (GTK)
- ...

nicht nur Unterschiede im look-and-feel, sondern insbesondere in der Verwendung, Realisierung (und damit auch der Portabilität, Performanz, etc.)

im Folgenden: Swing

GUIs bauen

im Prinzip einfach: alle Komponenten sind Objekte bzw. werden darüber verwaltet

Komponenten erzeugen durch Anlegen von Objekten, evtl. noch explizit sichtbar machen

```
JFrame f = new JFrame("JFrame");
f.setSize(200,100);
f.setVisible(true);
```

Anordnung der einzelnen Komponenten durch *LayoutManager* gesteuert

Verknüpfungen (Button *im* Frame, bzw. als Kind von Frame) durch bereitgestellte Methoden, z.B. `add`

Komponenten einer GUI bilden baumartige Struktur von außen nach innen

Zeichnen

Methode `paint` in `JComponent` legt fest, was wie gezeichnet wird

Aufruf aber mit `repaint()`

interner Mechanismus zum effizienteren Zeichnen (unnötiges Zeichnen vermeiden, etc.)

gezeichnet wird nicht in Komponente direkt, sondern über assoziiertes `Graphics`-Objekt

Methode `getGraphics()` von `JComponent` liefert `Graphics`-Objekt

Zeichnen

Methode `paint` von `JComponent` ruft normalerweise `paintComponent` auf (Rahmen und Kinder in entsprechenden Methoden gezeichnet)

zum Zeichnen (z.B. vom Spielbrett) sollte also

```
protected void paintComponent(Graphics g)
```

überschrieben werden

GUIs benutzen können

GUIs erfordern im Prinzip komplizierten Kontrollfluss (Benutzer kann jederzeit überall hinklicken, etc.)

zentrales Element zur Steuerung des Kontrollflusses: *Events*, z.B. Mausklick

Programm muss nicht selbst ständig alle möglichen Interaktionen bereitstellen, sondern registriert *Listener*, die bei Bedarf aufgerufen werden

Swing benutzt eigenen Thread für Grafik

Events

Events sind Objekte vom Typ `EventObject` oder Unterklasse, z.B. `MouseEvent`

Methode `getSource()` liefert Object vom Typ `Object`, Auslöser des Events, z.B. Button

zwei Arten von Events:

- low-level Events, z.B. `MouseEvent`
- semantische Events, z.B. `ActionEvent`

oft wird semantisches Event von low-level Event ausgelöst
Bsp.: Mausklick löst MouseEvent aus, welches auf Button ActionEvent auslöst

Listener

Komponenten (z.B. Buttons) können Listener registrieren

verschiedene Arten von Listenern vorhanden, typischerweise als Interfaces

```
public interface ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

Mechanismus:

- implementiere den Listener
- erzeuge Listener-Objekt `myListener`
- registriere dies an der GUI-Komponente, z.B. mit `komponente.addActionListener(myListener)`
- GUI-Komponente wird bei Bedarf Methode `actionPerformed` bei `myListener` aufrufen

Listener installieren

Standardmethode:

```
public class MyListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
     ...
  }
}

public class MyFrame extends JFrame {
  ...
  JButton button = new JButton("Klick mich an!");
  MyListener mylistener = new MyListener();
  button.addActionListener(mylistener);
  ...
}
```

Nachteile

- ganze Klassendefinition evtl. nur für ein einziges Objekt
- `MyListener` hat keinen Zugriff auf `MyFrame`, methode `actionPerformed` soll aber intuitiv darin arbeiten

Innere Klassen

zweiter Nachteil durch innere Klassen zu beheben

```
public class MyFrame extends JFrame {

  class MyListener implements ActionListener {
    public actionPerformed(ActionEvent e) {
       ...
    }
  }

  ...
  JButton button = new JButton("Klick mich an!");
  MyListener mylistener = new MyListener();
  button.addActionListener(mylistener);
  ...
}
```

z.B. Instanzvariablen von `MyFrame` jetzt sichtbar in `MyListener`

Anonyme Klassen

spezielle Form der inneren Klasse, die keinen Namen hat und deswegen auch nirgendwo anders mehr benutzt werden kann

```
public class MyFrame extends JFrame {

  ...
  JButton button = new JButton("Klick mich an!");
  MyListener mylistener = new MyListener();
  button.addActionListener(new ActionListener() {
    public actionPerformed(ActionEvent e) {
      ...
    }
  });
  ...
}
```

Hinweis: Zugriff auf lokale Variablen nur, wenn diese `final` sind

Anonyme Klassen

beachte: Definition = Verwendung

Syntax:

`new` *Klassenname*`(`*Parameter*`) { `*Klassendefinition*` }`

beachte:

- *Klassenname* ist **nicht** Name der Klasse (sonst wäre sie wohl kaum anonym), sondern Name einer Klasse, von der geerbt wird, oder eines Interfaces, welches implementiert wird
- anonyme Klassen können keine Konstruktoren haben
- die Parameter (auch im Falle von `()`) werden an den Konstruktor der Oberklasse übergeben

Eigenimplementierung

Nachteile obiger Ansätze: neue Klassen werden definiert

beachte: Listener sind typischerweise Interfaces

eine Komponente braucht nicht unbedingt einen separaten Listener, kann es ja auch selbst machen

```
public class MyFrame extends JFrame implements ActionListener {

  public actionPerformed(ActionEvent e) {
    ...
  }

    ...
    JButton button = new JButton("Klick mich an!");
    MyListener mylistener = new MyListener();
    button.addActionListener(this);
    ...
}
```

Swing und Threads

Swing benutzt einen Thread, in dem Events ausgelöst und gezeichnet wird: *event-dispatching thread* (EDT)

Komponenten können nicht gleichzeitig von mehreren Threads benutzt werden, normalerweise EDT

einige Komponenten bzw. Methoden sind thread-safe, insbesondere `repaint()` und (De-)Registrierung von Listenern

es gibt Methoden, um im EDT etwas ausführen zu lassen

neue Threads können angelegt und mit Aufgaben gefüttert werden

Code im EDT ausführen

`invokeLater` und `invokeAndWait` lassen Code im EDT ausführen

- `invokeLater` terminiert sofort, führt Code später aus
- `invokeAndWait` terminiert erst, wenn Code vollständig abgearbeitet ist

```
import javax.swing;

...
Runnable o = new Runnable() {
  public void run() {
    // auszufuehrender Code
  }
};

SwingUtilities.invokeLater(o);
```

auch hier evtl. innere Klassen sinnvoll

Berechnungsintensive Aufgaben

Code im EDT (z.B. beim Ausführen von Listenern) sollte nicht zeitaufwändig sein, da sonst weitere Events blockiert werden

Threads eigentlich am besten vermeiden (Race conditions, Performanz)

manchmal aber unumgänglich, neue Threads aus EDT heraus zu starten

- größere Berechnung
- wiederholte Ausführung
- Berechnung mit Abwarten
- ...

$\rightsquigarrow$ `SwingWorker` und `Timer`, erst ab Java 1.6 in Swing enthalten!

SwingWorker

abstrakte Klasse, parametrisiert mit

1. Typ `T` des Resultats der Berechnung
2. Typ `V` eventueller Zwischenresultate

überschreibe `T doInBackground()` mit Code zur Durchführung der Berechnung

Start der Berechnung in eigenem Thread mittels `void execute()`, terminiert sofort

auf Ergebnis warten mittels `T get()`

Zwischenergebnisse können mit `publish` und `process` verarbeitet werden

Zu beachten

- Listener sollten sehr schnell auszuführen sein
- Listener für semantische statt low-level Events installieren
- Performanz des gesamten Programms hängt ab von Anzahl der zu ladenden Klassen

Dokumentation mit Javadoc

Dokumentation

Wenn man nicht sagt, was ein Programm machen soll, ist die Wahrscheinlichkeit gering, dass es das macht.

Dokumentation soll helfen,

- Schnittstellen zu verstehen
- Entwurfsideen zu erklären
- implizite Annahmen (z.B. Klasseninvarianten) auszudrücken

Nicht sinnvoll:

```
x++; // erhöhe x um eins
```

Was soll man dokumentieren?

- welche Werte zulässige Eingaben für eine Methode sind
- wie Methoden mit unzulässigen Eingaben umgehen
- wie Fehler behandelt und an den Aufrufer der Methode zurückgegeben werden
- Verträge (Vorbedingungen, Nachbedingungen, Klasseninvarianten)
- Seiteneffekte von Methoden (Zustandsänderungen, Ein- und Ausgabe, usw.)
- wie die Klasse mit Nebenläufigkeit umgeht
- (wie Algorithmen funktionieren, wenn das nicht leicht vom Programmtext ablesbar ist)

Javadoc-Kommentare

- Javadoc-Kommentare
  `/** HTML-formatierter Kommentar */`
- stehen vor Packages, Klassen, Methoden, Variablen
- spezielle Tags wie `@see`, `@link`
- HTML-Dokumentation wird erzeugt mit

```
javadoc Package
javadoc Klasse1.java Klasse2.java ...
```

Javadoc-Tags

Allgemein

- `@author Name`
- `@version text`

Vor Methoden

- `@param Name Beschreibung` – beschreibe einen Parameter einer Methode
- `@return Beschreibung` – beschreibe den Rückgabewert einer Methode
- `@throws Exception Beschreibung` – beschreibe eine Exception, die von einer Methode ausgelöst werden kann

Javadoc-Tags

Querverweise

- `{@link Klasse}`
  `{@link Klasse#Methode}`
  ...
  Verweis auf andere Klassen und Methoden im laufenden Text (in HTML-Version werden daraus Links, Warnung wenn nicht existent)
- `@see Klasse`
  `@see Klasse#Methode`
  ...
  Fügt der Dokumentation einen „See Also“-Abschnitt mit Querverweisen hinzu.

  `@see`-Tags können nicht im laufenden Text stehen

Javadoc-Beispiel

```
/**
 * Allgemeine Kontenklasse
 * @author banker
 * @see NichtUeberziehbaresKonto
 */
public class Konto {

   /**
    *  Geld auf Konto einzahlen.
    *  <p>
    *  Wenn vorher <code> getKontoStand() = x </code>
    *  und <code> betrag >=0 </code>,
    *  dann danach <code> getKontoStand() = x + betrag </code>
    *  @param betrag positive Zahl, der einzuzahlende Betrag
    *  @throws ArgumentNegativ wenn betrag negativ
    */
   public void einzahlen(double betrag)
}
```

Enumeration Types

Aufzählungstypen

bestehen aus einer festen (und normalerweise kleinen) Anzahl benannter Konstanten

Bsp.:

- Spielkartenfarben: Karo, Kreuz, Herz, Pik
- Wochentage: Montag, . . . , Sonntag
- Noten: Sehr gut,. . . , Ungenügend
- . . .

vor Version 1.5 keine direkt Unterstützung in Java, Modellierung z.B. durch

- `final`-Konstanten vom Typ `int` o.ä., nicht typ-sicher!
- Klasse `Wochentag` mit sieben Unterklassen, sehr aufwändig

Aufzählungstypen in Java

Bsp.: Wochentage

```
public enum Wochentag {
  MO, DI, MI, DO, FR, SA, SO;
}
```

definiert Typ mit 7 Werten

Werte zu verwenden z.B. als `Wochentag.DO`

Aufzählungstypen als Klassen

Deklaration der Form `enum A { ... }` wird vom Compiler in normale Klasse übersetzt

Enum-Typen können auch Methoden haben

Konstanten können assoziierte Werte haben

Beispiel

```
public enum Wochentag {
  MO(0), DI(1), MI(2), DO(3), FR(4), SA(5), SO(6);

  private tag;
  private static final MAX = 7;

  Wochentag(int t) {
    tag = t;
  }

  public index() {
    return tag;
  }

  public static int abstand(Wochentag x, Wochentag y) {
    return (y.index() - x.index() + MAX) % MAX;
  }
}
```

Aufzählungstypen im Typsystem

Enum-Typen erweitern immer `java.lang.Enum` implizit

`public enum A extends B { ... }` nicht zulässig

cleverer Mechanismus (in der Konstruktion von `Enum`, nicht im Compiler) sorgt dafür, dass folgendes bereits syntaktisch nicht korrekt ist

```
enum A { A1, A2 }
enum B { B1, B2 }


  ...
  if (B1.compareTo(A2)) { ... }
```

Zu beachten

- Konstruktoren in Enum-Typen nicht `public` machen
- Enum-Typen können nicht mithilfe von `extends` etwas erweitern
- Werte eines Enum-Typs sind automatisch geordnet, wie üblich mit `compareTo` erfragen
- Parameter an Konstanten können sogar Methoden sein
- statische Methode `values()` liefert `Collection` der einzelnen Werte, kann z.B. mit `Iterator` durchlaufen werden

Syntax von Programmiersprachen

Programmiersprachen

Sprache = Menge von Wörtern, typischerweise unendlich

Programmiersprache: Wörter repräsentieren *Programme*

Programm kann auf einem Computer (evtl. nur abstrakte Maschine) ausgeführt werden und induziert somit eine Berechnung

Bsp.:

- Javaprogramme werden in JVM ausgeführt
- arithmetische Ausdrücke im Taschenrechner evaluieren
- Tastenkombinationen zur Steuerung des Handys
- ...

Begriffsbildung

typisches Problem: gegeben Programm P, potentiell aus der Sprache L, führe dies aus

- Syntax: beschreibt die Programmiersprache; welche Wörter sind korrekte Programme, welche nicht
- Semantik: beschreibt die Bedeutung eines Programms; wie ist dieses auszuführen
- Interpreter: Programm, welches einen Computer steuert, um andere Programme auszuführen
- Compiler: Programm, welches Programme einer Sprache L in äquivalente Programme einer anderen Sprache L' (typischerweise Maschinensprache) übersetzt
- Lexer/Parser: Programme, die aus einem Wort, gegeben als String, eine baumartige Struktur bauen; werden benutzt, um die Struktur in Programmen zu erkennen

Syntax

Syntax von Programmiersprachen häufig gegeben durch kontext-freie Grammatik (evtl. mit Nebenbedingungen), z.B.

$$\begin{array}{rcl} \langle \mathit{Prg} \rangle & ::= & \ldots \mid \texttt{while} \texttt{ (} \langle \mathit{BExp} \rangle \texttt{) \{} \langle \mathit{Prg} \rangle \texttt{ \}} \mid \ldots \\ \langle \mathit{BExp} \rangle & ::= & \ldots \mid \langle \mathit{BExp} \rangle < \langle \mathit{BExp} \rangle \mid \ldots \end{array}$$

in diesem Beispiel wichtig:

- Teil, der aus $\langle \mathit{Prg} \rangle$ abgeleitet wird, ist ein *Programm*
- Teil, der aus $\langle \mathit{BExp} \rangle$ abgeleitet wird, ist ein *boolescher Ausdruck*

warum ist solch eine Unterscheidung wichtig?

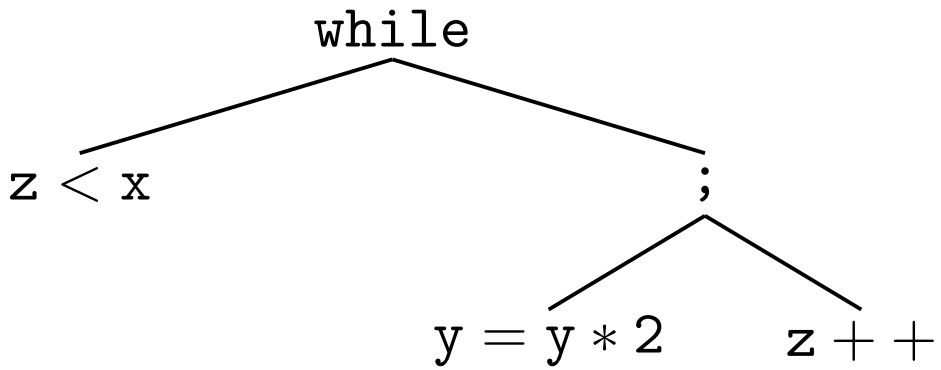
Semantik besagt sicherlich, dass boolesche Ausdrücke zu **true** oder **false** ausgewertet werden; Auswertung eines Programm ist jedoch anders

Notwendigkeit des Parsens

zum Verstehen und Ausführen des Programms

```
while (z < x) {
    y = y*2; z++
}
```

ist es notwendig, darin Struktur zu erkennen, z.B.



beachte:

- Links-Mitte-Rechts-Durchlauf des Baums liefert nicht originale Stringrepräsentation des Programms
- Ausführung des Programms abhängig von Art des Teilbaums (z.B. Programm vs. bool. Ausdruck)

Konkrete Syntax

konkrete Syntax einer Programmiersprache beschreibt genau die Zeichenketten, die gültige Programme beschreiben, z.B.

$$\begin{aligned} \langle Prg \rangle &::= \ldots \mid \langle Id \rangle = \langle Exp \rangle \\ \langle Id \rangle &::= \langle Alpha \rangle (\langle Alpha \rangle \cup \langle Num \rangle)^* \end{aligned}$$

beachte: in diesem Fall gar nicht so konkret, gewisser Grad an Abstraktion bereits vorhanden: Whitespaces evtl. implizit zugelassen

```
x3=x1        x3 = x1        x3 =
                                 x1
```

konkrete Syntax wird dazu verwendet, in Programmen Struktur zu erkennen, also für den *Parser*

Abstraktionsgrad bzgl. Whitespaces o.ä. wird vom *Lexer* gehandhabt

Abstrakte Syntax

konkrete Syntax stellt oft zuviel Information bereit, welche nur zum Erkennen der Programmstruktur notwendig ist, aber zu Redundanz in Ableitungsbäumen führt (Beispiel folgt)

abstrakte Syntax beschreibt ebenfalls Prgrammstruktur, aber auf abstrakterer Ebene, z.B.

$$\langle Prg \rangle \quad ::= \quad \ldots \mid Id{=}\langle Exp \rangle$$

hier also z.B. Bezeichner als Terminale aufgefasst, die zusätzlich einen Namen haben, z.B. $Id(x3)$

Grund: für Auswertung von Programmen oder Ausdrücken ist lexikalische Struktur von Bezeichnern uninteressant, wichtig ist nur zu wissen, ob es ein Bezeichner oder Ausdruck oder Konstante etc. ist

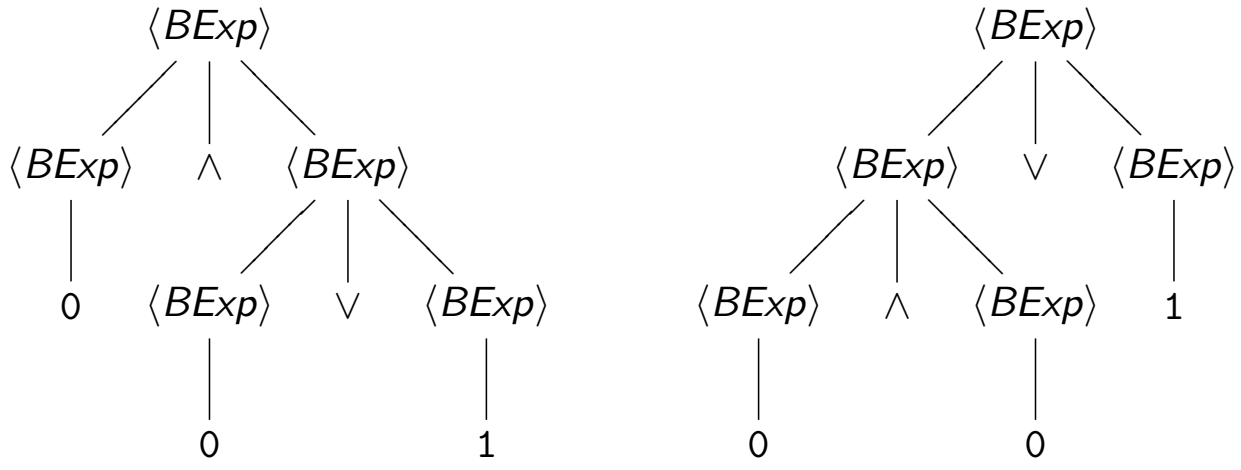
Konkrete vs. abstrakte Syntax

Bsp.: einfache Sprache boolescher Ausdrücke über 0,1 mit $\vee$, $\wedge$ mit üblicher Auswertung

$$\langle BExp \rangle \quad ::= 0 \mid 1 \mid \langle BExp \rangle \vee \langle BExp \rangle \mid \langle BExp \rangle \wedge \langle BExp \rangle$$

Ausdruck $0 \wedge 0 \vee 1$ is gültig bzgl. dieser Grammatik, aber was ist sein Wert?

Grammatik ist *mehrdeutig*



Mehrdeutigkeiten

Grammatik ist mehrdeutig, wenn es ableitbare Wörter mit mehr als einem Ableitungsbaum gibt

$\rightsquigarrow$ keine eindeutige Struktur; Auswerten nicht eindeutig

Mehrdeutigkeiten können oft durch *Präzedenzregeln* aufgelöst werden; hier: "$\wedge$ bindet stärker als $\vee$"

Umgang mit Präzedenzen:

- Parser so programmieren, dass er sie berücksichtigt, oder
- in Grammatik einbauen

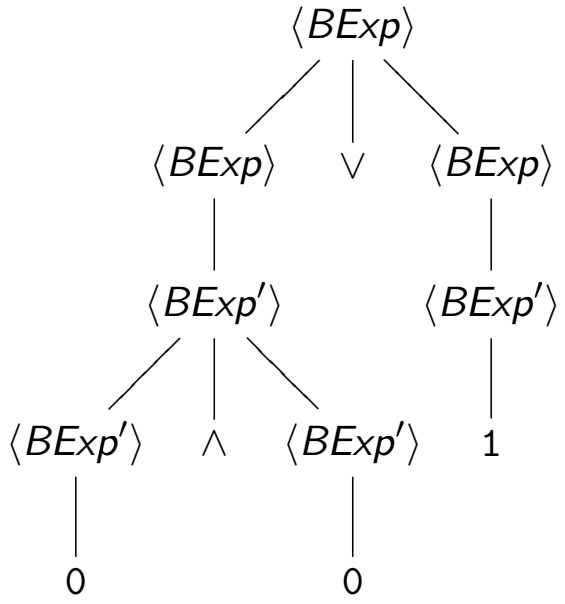
Mehrdeutigkeiten eliminieren

hier z.B.

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

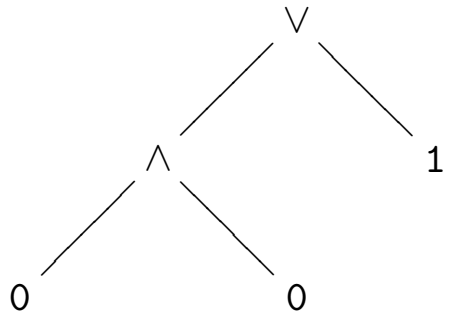
nur noch ein Ableitungsbaum für besagtes Wort mit richtiger Präzedenz



Vorteile abstrakter Syntax

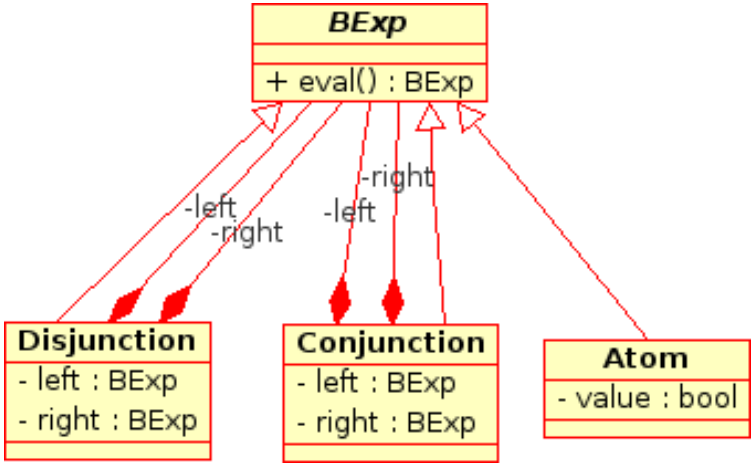
obiger Ableitungsbaum ist unnötig gross und Auswertung von $\langle BExp \rangle$ und $\langle BExp' \rangle$ unterscheidet sich nicht wesentlich

Struktur und nötige Information kann auch so gespeichert werden



hier benötigt: innere Knoten sind Operatoren, Blätter sind Konstanten

leicht abzubilden in Klassendiagramm



Vorgehensweise

hier: Programmiersprache AntBrain gegeben

- konkrete Syntax durch kontext-freie Grammatik
- abstrakte Syntax durch UML-Klassendiagramm

was ist zu tun?

1. aus Grammatik extrahieren: was sind die wichtigen Teile einer Eingabe? Nicht einzelne Zeichen, sondern Schlüsselwörter, Bezeichner, Konstanten, Befehle, . . . , genannt *Tokens*
   Lexer schreiben, der Zeichenkette in Liste von Tokens zerlegt / übersetzt
2. aus konkreter Syntax einen Parser bauen, der Listen von Tokens in Ableitungsbäume umwandelt; diese möglichst gleich als baumförmige Objektstrukturen der abstrakten Syntax erzeugen

Beispiel

gesamtes Vorgehen beispielhaft gezeigt anhand einfacher Sprache arithmetischer Ausdrücke

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp} \rangle + \langle \mathit{IExp} \rangle \mid \langle \mathit{IExp} \rangle * \langle \mathit{IExp} \rangle \mid (\langle \mathit{IExp} \rangle) \mid \\
&\quad \langle \mathit{Num} \rangle \mid \langle \mathit{Id} \rangle \mid \texttt{rand} \\
\langle \mathit{Num} \rangle &::= \langle \mathit{Digit} \rangle^{+} \\
\langle \mathit{Digit} \rangle &::= \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \mathit{Id} \rangle &::= \langle \mathit{Alpha} \rangle \big(\langle \mathit{Alpha} \rangle \cup \langle \mathit{Digit} \rangle \big)^{*} \\
\langle \mathit{Alpha} \rangle &::= \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}
\end{aligned}
$$

beachte: Grammatik ist mehrdeutig

Beispiel mit eindeutiger Grammatik

Präzedenzregel in Grammatik einbauen und zusätzlich Ende der Eingabe in Grammatik verlangen, damit der Parser später vollständige Eingaben verarbeitet

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \Big(+ \langle \mathit{IExp}_2 \rangle \Big)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \Big(* \langle \mathit{IExp}_3 \rangle \Big)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IExpAtom} \rangle \\
\langle \mathit{IExpAtom} \rangle &::= \langle \mathit{Num} \rangle \mid \langle \mathit{Id} \rangle \mid \texttt{rand} \\
&\vdots
\end{aligned}
$$

(in abstrakter Syntax würde man $\langle \mathit{Num} \rangle$ und $\langle \mathit{Id} \rangle$ als Tokens betrachten)

Token

sinnvolle Gruppierung von Teilen einer Zeichenkette in Tokens:
`EOF`, `Identifier`, `Keyword`, `Number`, `Symbol`

einige Tokens haben Werte:

- ein String bei `Identifier` – der Name
- ein Integer bei `Number` – sein Wert
- ein Name bei `Keyword` – um welches es sich handelt; hier gibt es jedoch lediglich `rand`
- ein Name bei `Symbol` – es gibt die Symbole `(`,`)`,`+`,`*`, die gut als Aufzählungstyp modelliert werden können

Lexer

nimmt Zeichenkette und wandelt diese in Liste von Tokens um, z.B.

```
47 + x7 + rand) * (016 +)rand34
```

wird zu

Number(47), Symbol("+"), Identifier("x7"), Symbol("+"), Keyword(rand), Symbol(")"), Symbol("*"), Symbol("("), Number(16), Symbol("+"), Symbol(")"), Identifier("rand34")

beachte:

- Lexer überprüft nicht, ob Zeichenkette herleitbar ist (Parser)
- Lexing kann auch fehlschlagen, hier z.B. bei `x - 3`
- Java-Klasse `Scanner` hier nicht geeignet, um Lexer zu konstruieren

Parser

zur Erinnerung: Parser erhält Liste von Tokens und konstuiert daraus Ableitungsbaum bzgl. (abstrakter) Syntax, der diese Liste als Blätter und Startsymbol der Grammatik als Wurzel enthält

es gibt allgemeines Verfahren zum Parsen kontext-freier Grammatiken

- läuft in Zeit $\mathcal{O}(n^3)$ bei Eingabelänge n
- konstruiert endliche Repräsentation aller Ableitungsbäume

beides ist nicht von Vorteil (bei eindeutigen Grammatiken)

Ausweg: Unterklassen der kontext-freien Sprachen, die einfacheres und damit evtl. auch schnelleres (z.B. in $\mathcal{O}(n)$) Parsen erlauben

Bottom-Up vs. Top-Down Parsen

im Prinzip gibt es zwei Arten von Parsern

- Bottom-Up
  - finde Teil der Eingabe der Form $B_1 \ldots B_k$, sodass es Regel $A ::= B_1 \ldots B_k$ gibt; ersetze diesen Teil durch A
  - endet, wenn gesamte Eingabe durch Startsymbol S ersetzt wurde
  - Problem: normalerweise keine eindeutige Ersetzung möglich
- Top-Down
  - interessant ist nur, ob Eingabe $t_1 \ldots t_n$ aus Startsymbol S herleitbar ist
  - beginnt sozusagen an der Wurzel S
  - versucht, sukzessive Teilbäume darunter durch Anwenden einer Regel zu konstruieren
  - Problem: normalerweise stehen mehrere Regeln zur Auswahl
  - endet, wenn Blätterfront der Eingabeliste entspricht

Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid \mathtt{0} \mid \mathtt{1}$$

Parsen:

$$0 \qquad \wedge \qquad 0 \qquad \vee \qquad 1$$

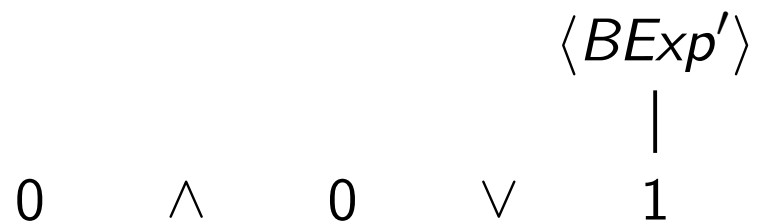
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:

$$\begin{array}{ccccccc} \langle BExp' \rangle & & & & & & \langle BExp' \rangle \\ | & & & & & & | \\ 0 & \wedge & 0 & \vee & 1 \end{array}$$

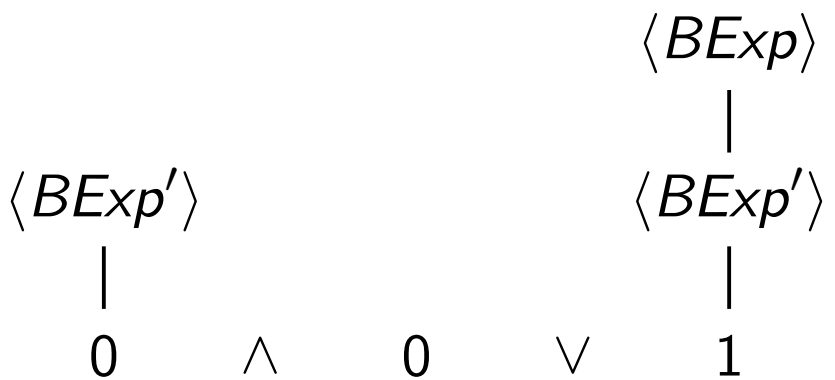
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



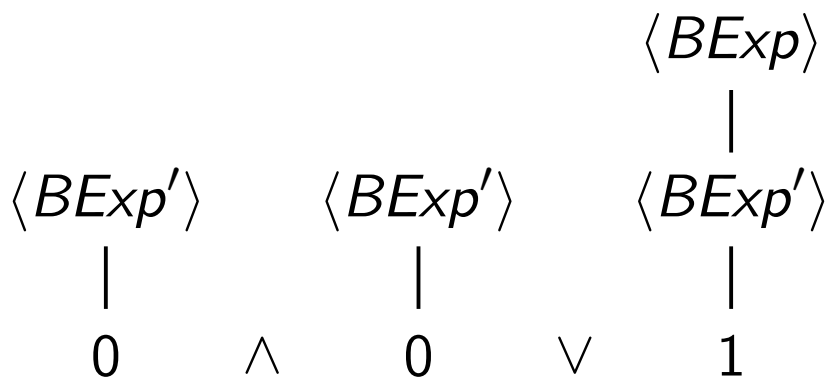
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



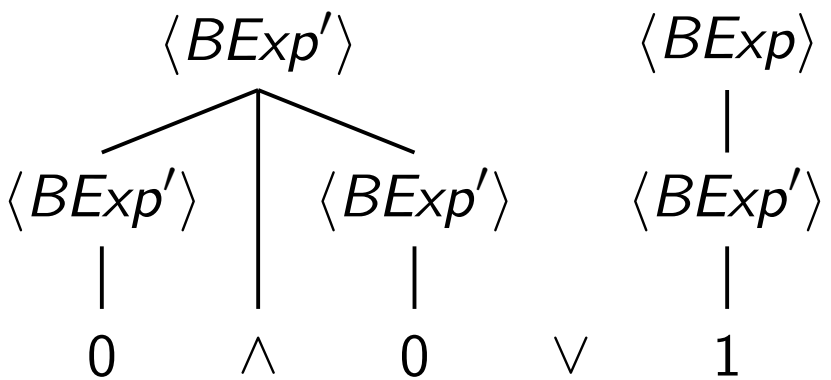
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



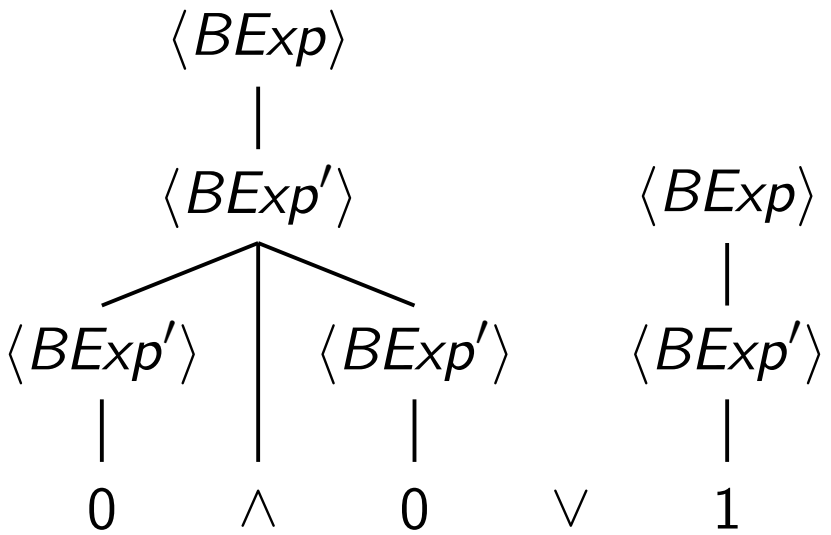
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



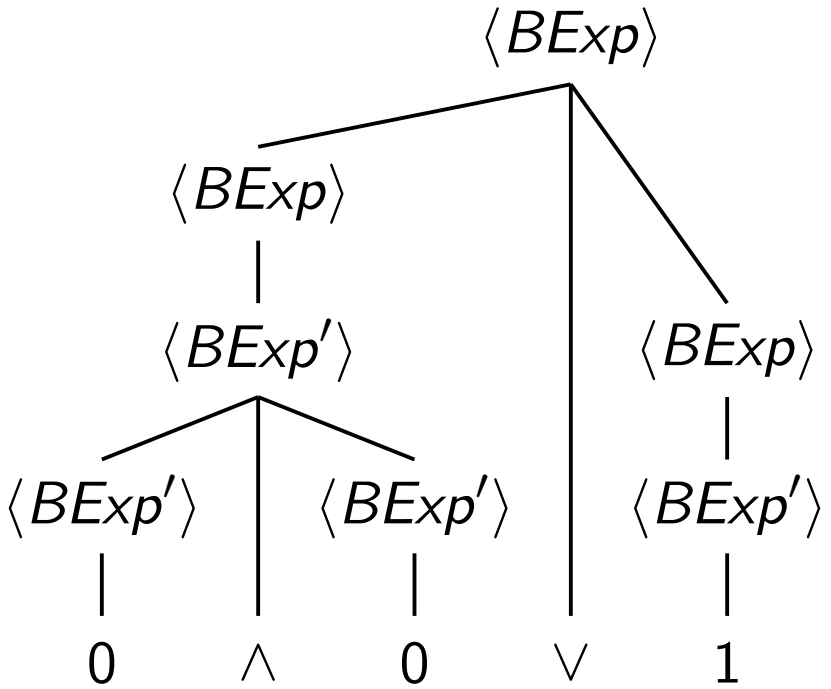
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



Top-Down Parser

allgemeine Funktionsweise:

`Parse`(Nonterminal A, TokenList $[t_1, \ldots, t_n]$)

wähle Regel $A ::= B_1 \ldots B_m$
wähle $i_1 \leq i_2 \leq \ldots i_m \leq i_{m+1}$ mit $i_1 = 1$, $i_{m+1} = n + 1$
`for` $j = 1, \ldots, m$ `do`
`Parse`(B_i, $[t_{i_j}, \ldots, t_{i_{j+1}-1}]$)

hier zur Vereinfachung nur ein *Recognizer*, kein *Parser*

muss also noch beim Rücksprung konstruierten Ableitungsbaum liefern

Verfahren ist nichtdeterministisch, braucht also noch Strategien, um Wählen deterministisch zu machen; hängt von Sprachklasse ab

Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:

$$\langle BExp \rangle$$

$$0 \qquad \wedge \qquad 0 \qquad \vee \qquad 1$$

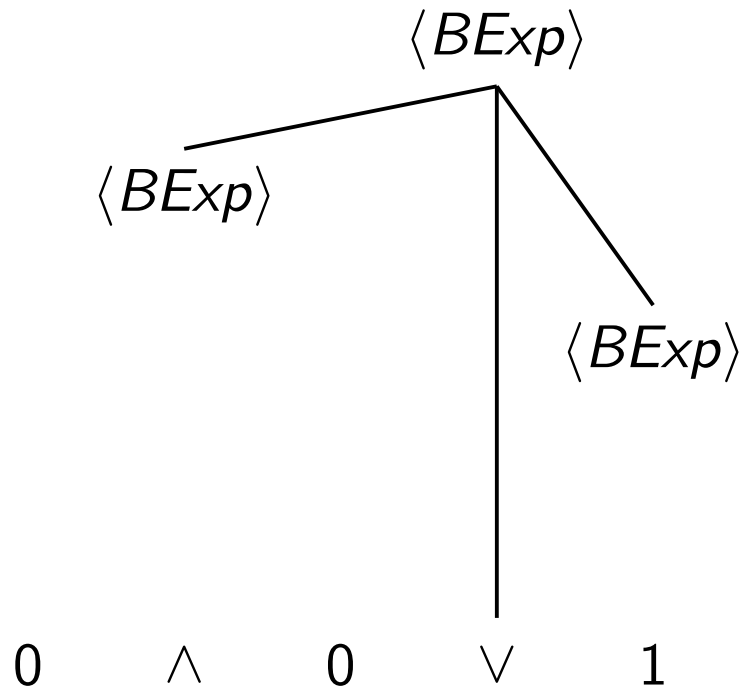
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



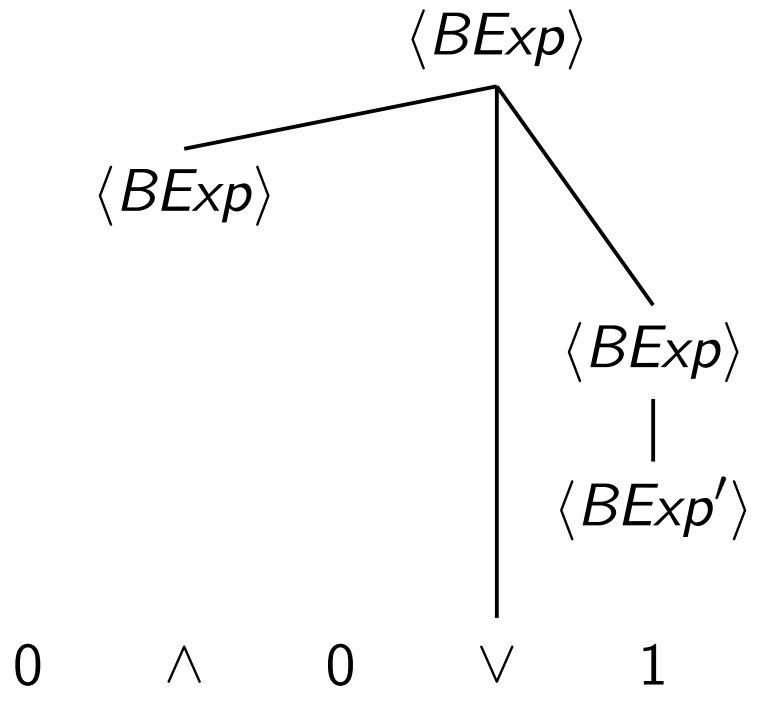
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



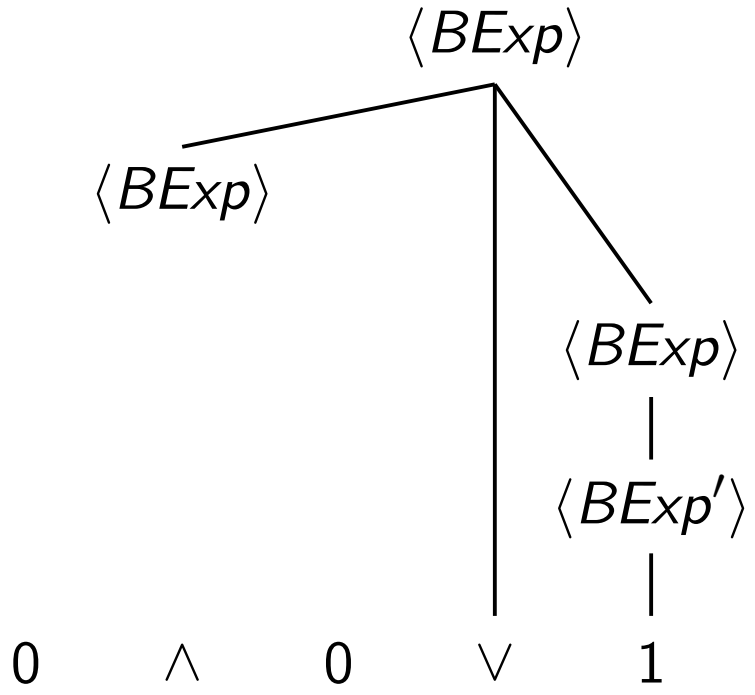
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



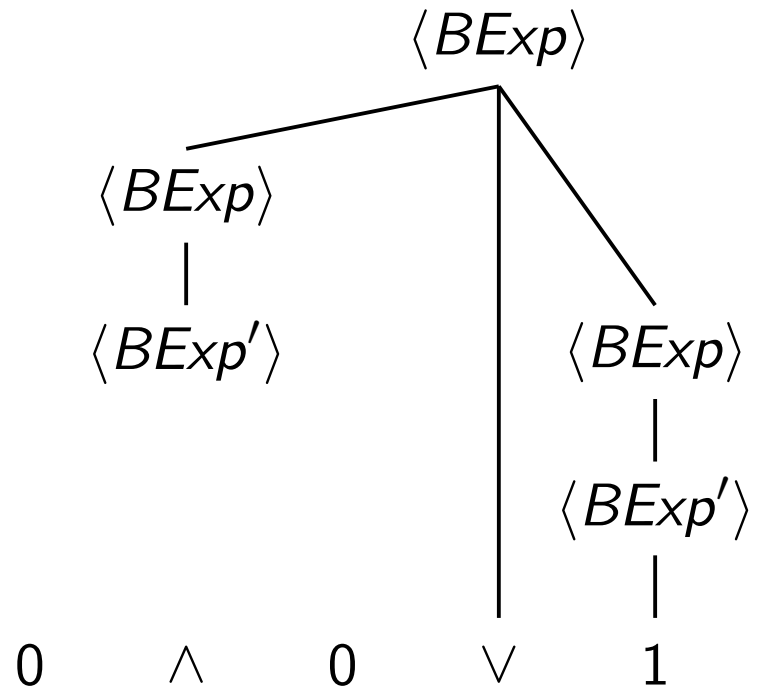
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



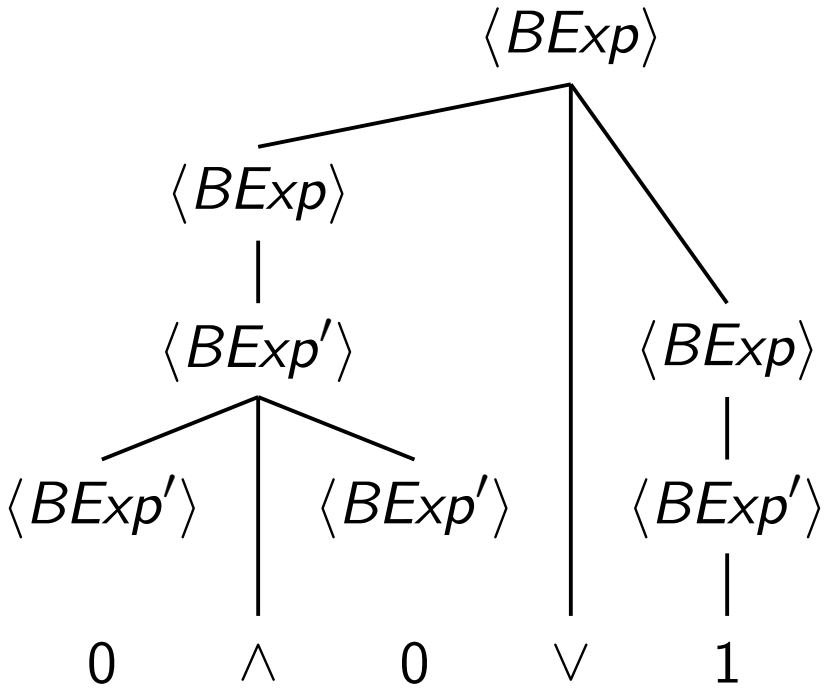
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



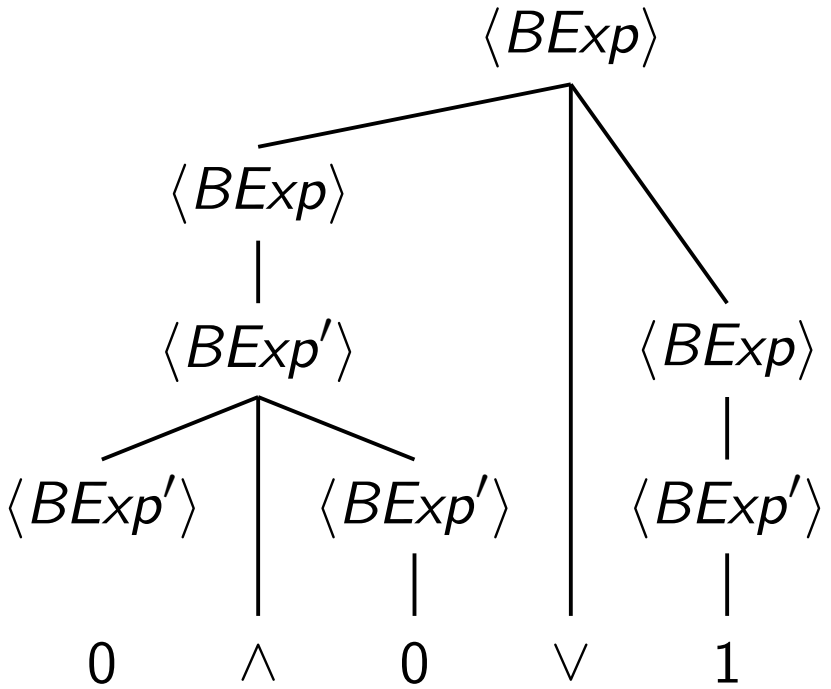
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



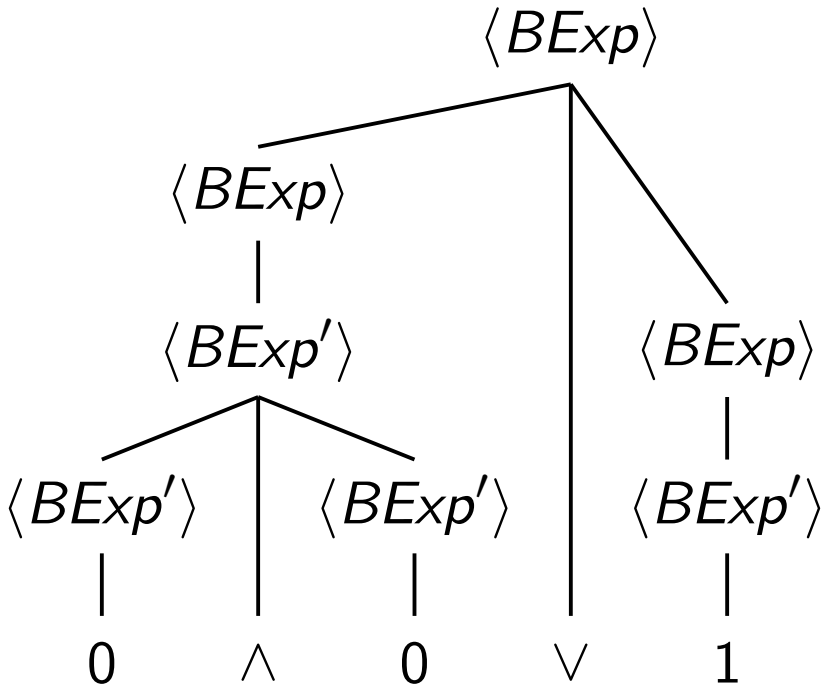
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



LL(k)-Sprachen

größeres Problem beim top-down Parsen: Wahl der Regel

LL(k)-Sprache ist Sprache, für die es eine LL(k)-Grammatik gibt

Def.: Sei G Grammatik über Alphabet Σ, α Satzform, $k \in \mathbb{N}$

$$first_k(\alpha) \;=\; \{v \in \Sigma^k \mid \exists \beta, \alpha \Rightarrow^* v\beta\}$$

Def.: G ist LL(k)-Grammatik, wenn für alle Nonterminals A und alle Regeln $A ::= \alpha$, $A ::= \beta$ mit $\alpha \neq \beta$ und alle Satzformen γ und Wörter v mit $S \Rightarrow^* vA\gamma$:

$$first_k(\alpha\gamma) \cap first_k(\beta\gamma) = \emptyset$$

soll heißen: bei einer LL(k)-Sprache erkennt man anhand von höchstens k der nächsten Tokens, welche Regel anzuwenden ist

LL(k)-Parsen

allgemeine Vorgehensweise:

`Parse`(Nonterminal A, TokenList $[t_1, \ldots, t_n]$)

wähle anhand von $t_1, \ldots, t_k$ eindeutige Regel $A ::= B_1 \ldots B_m$
wähle $i_1 \leq i_2 \leq \ldots i_m \leq i_{m+1}$ mit $i_1 = 1$, $i_{m+1} = n + 1$
`for` $j = 1, \ldots, m$ `do`
`Parse`(B_i, $[t_{i_j}, \ldots, t_{i_{j+1}-1}]$)

noch zu eliminieren: Nichtdeterminismus bzgl. Wahl der Zerlegung

- finde Zerlegung beim Parsen selbst
- parse erst $[t_1, \ldots, t_n]$ bzgl. B_1
- dies reduziert dann $[t_1, \ldots, t_{i-1}]$ für ein i zu einem Ableitungsbaum mit Wurzel B_1
- parse dann $[t_i, \ldots, t_n]$ bzgl. B_2, etc.

Beispiel

dies ist eine LL(1)-Grammatik

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IExpAtom} \rangle \\
\langle \mathit{IExpAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

betrachte Regeln für $\langle \mathit{IExp}_1 \rangle$ und $\langle \mathit{IExp}_2 \rangle$ z.B. als Abkürzung für

$$
\begin{aligned}
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \langle \mathit{IExpSeq}_1 \rangle \\
\langle \mathit{IExpSeq}_1 \rangle &::= + \langle \mathit{IExp}_2 \rangle \ \langle \mathit{IExpSeq}_1 \rangle \mid \epsilon
\end{aligned}
$$

beachte: Kleene-Sterne hier immer zu lesen als “soviel wie möglich mit dieser Regel parsen”

Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$

`(` `3` $+$ `rand` `)` $*$ `4` `EOF`

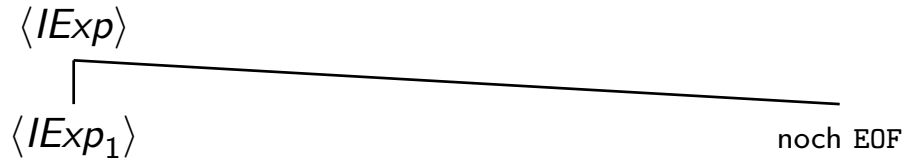
Beispiel LL(1)-Parsen

$\langle IExp \rangle$

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$

(3 + rand) * 4 EOF

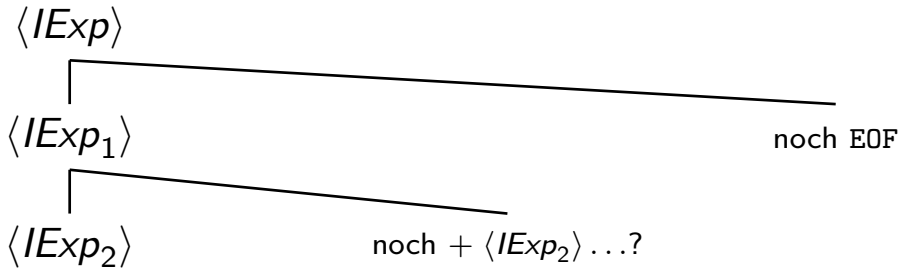
Beispiel LL(1)-Parsen



$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

`(  3  +  rand  )  *  4  EOF`

Beispiel LL(1)-Parsen

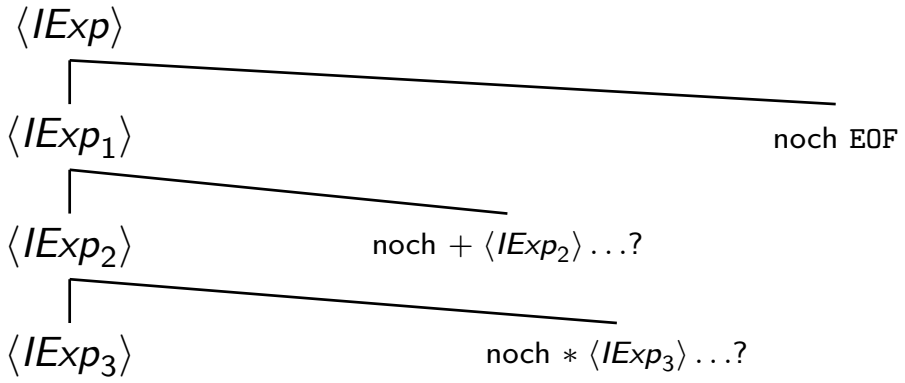


$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= \texttt{(} \ \langle \mathit{IExp}_1 \rangle \ \texttt{)} \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

`(` `3` `+` `rand` `)` `*` `4` `EOF`

Beispiel LL(1)-Parsen

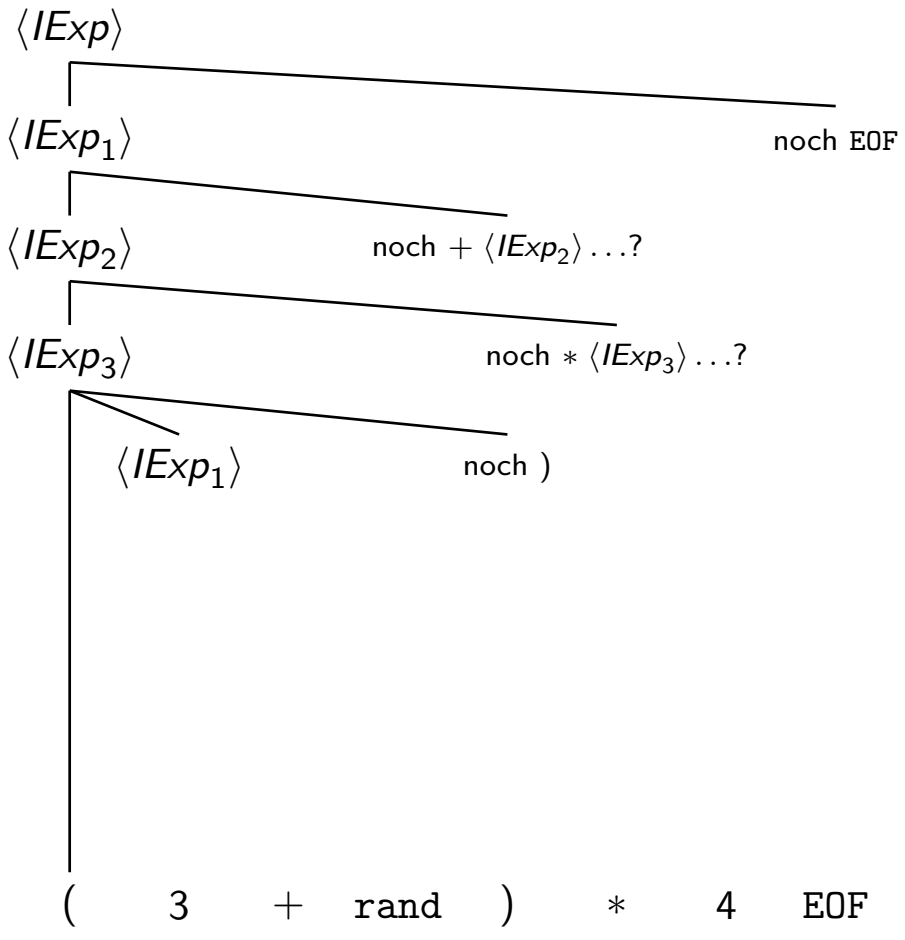
$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



`(` `3` `+` `rand` `)` `*` `4` `EOF`

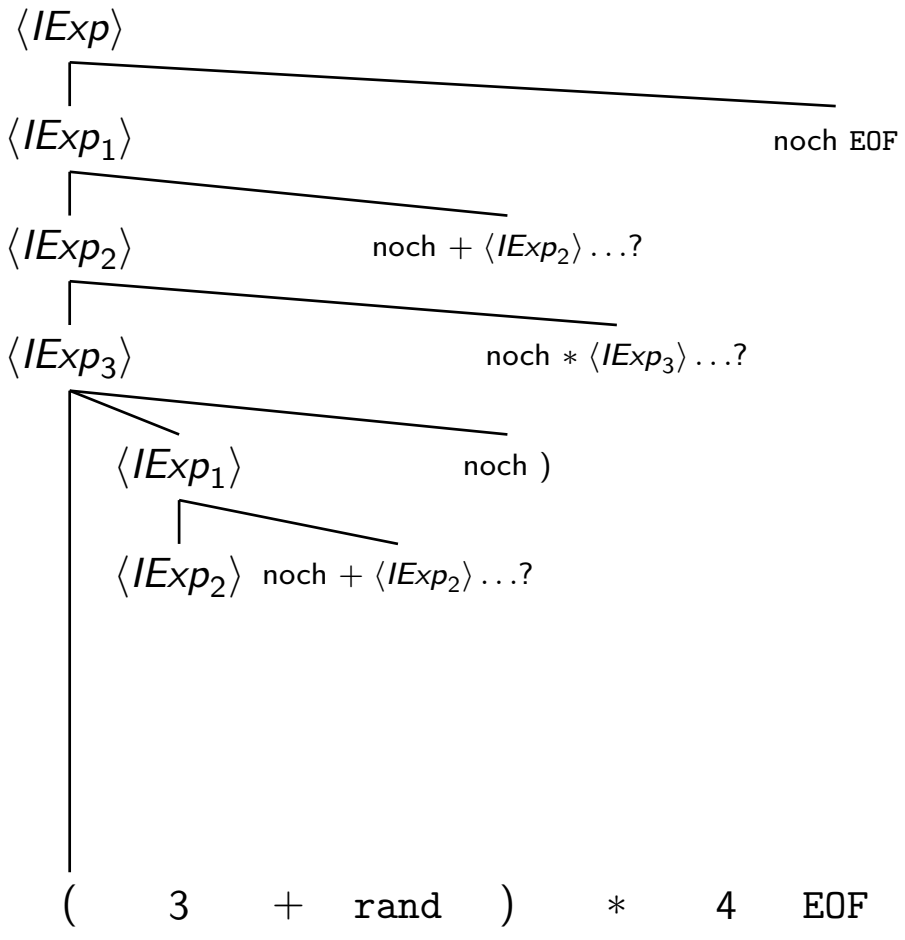
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



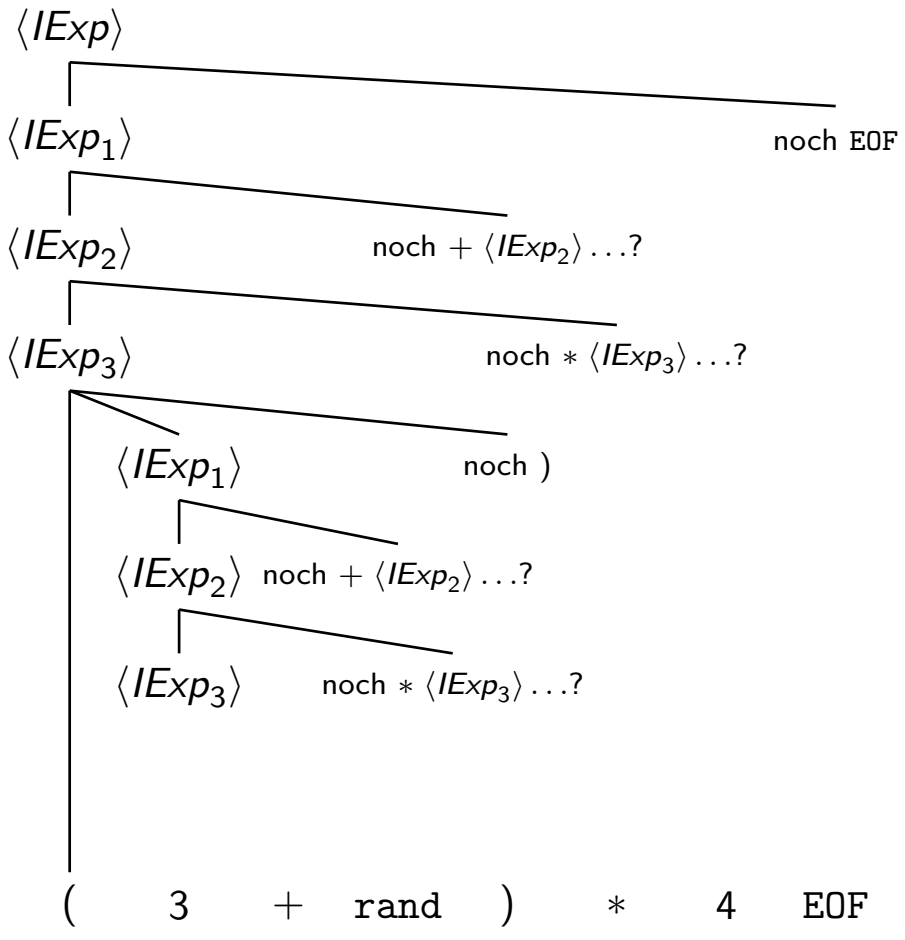
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



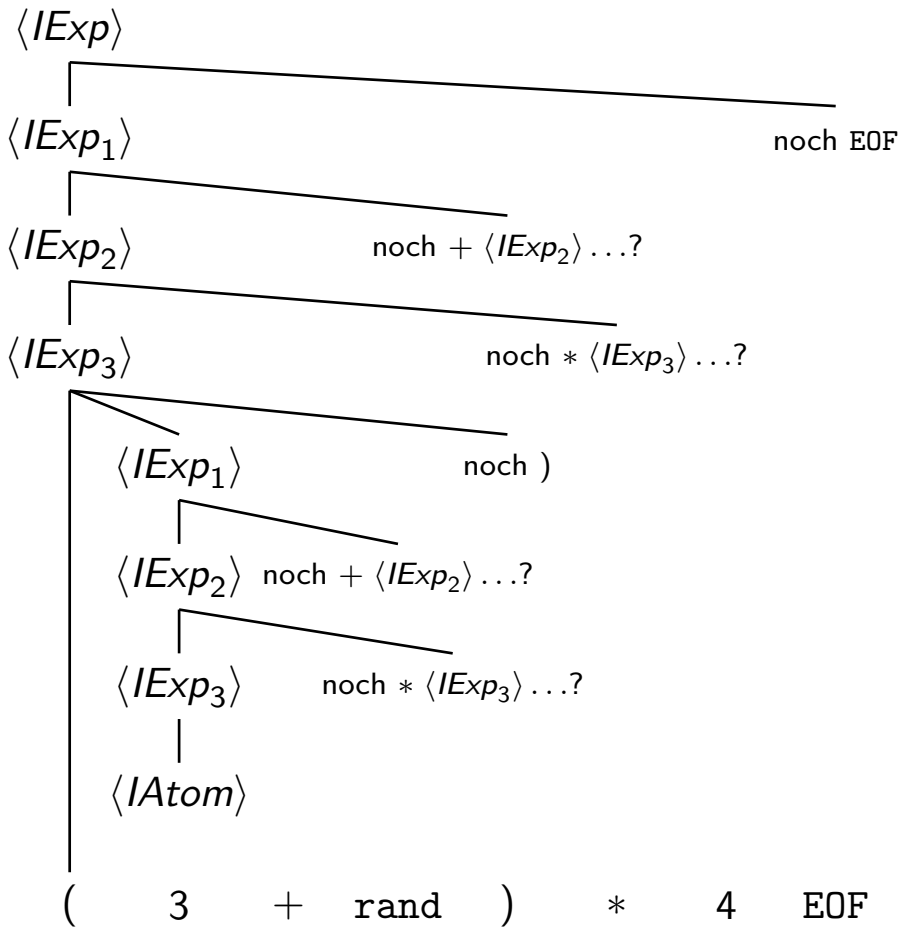
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



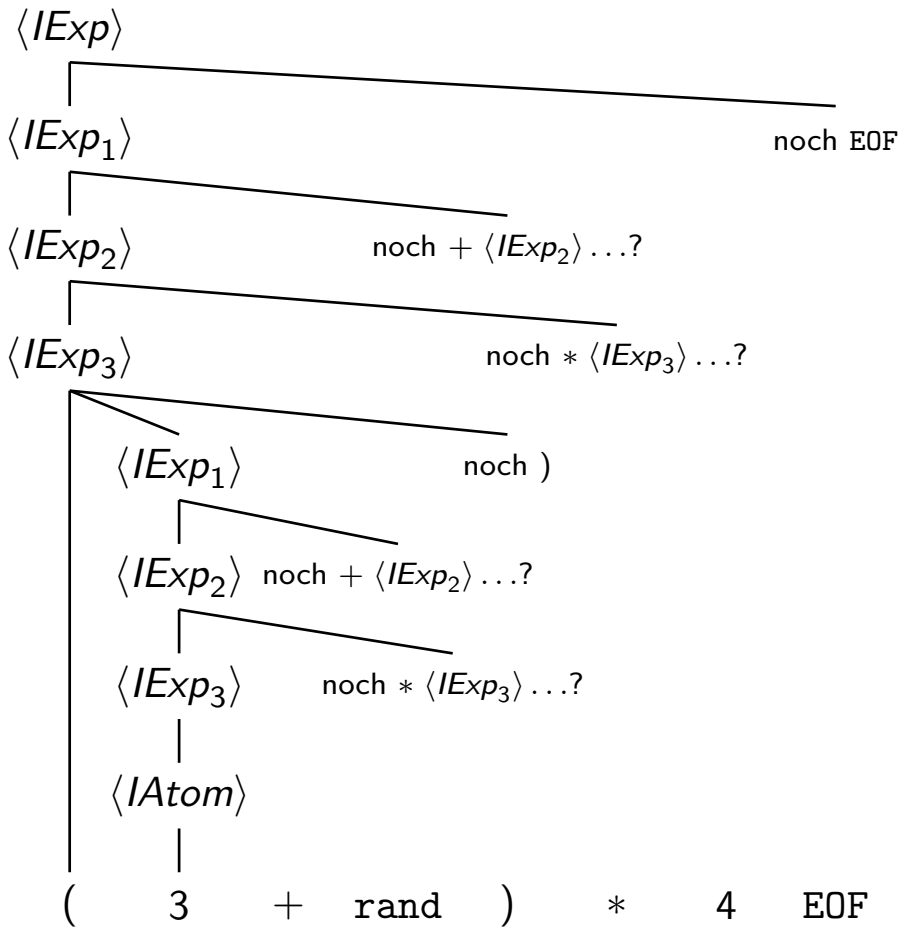
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



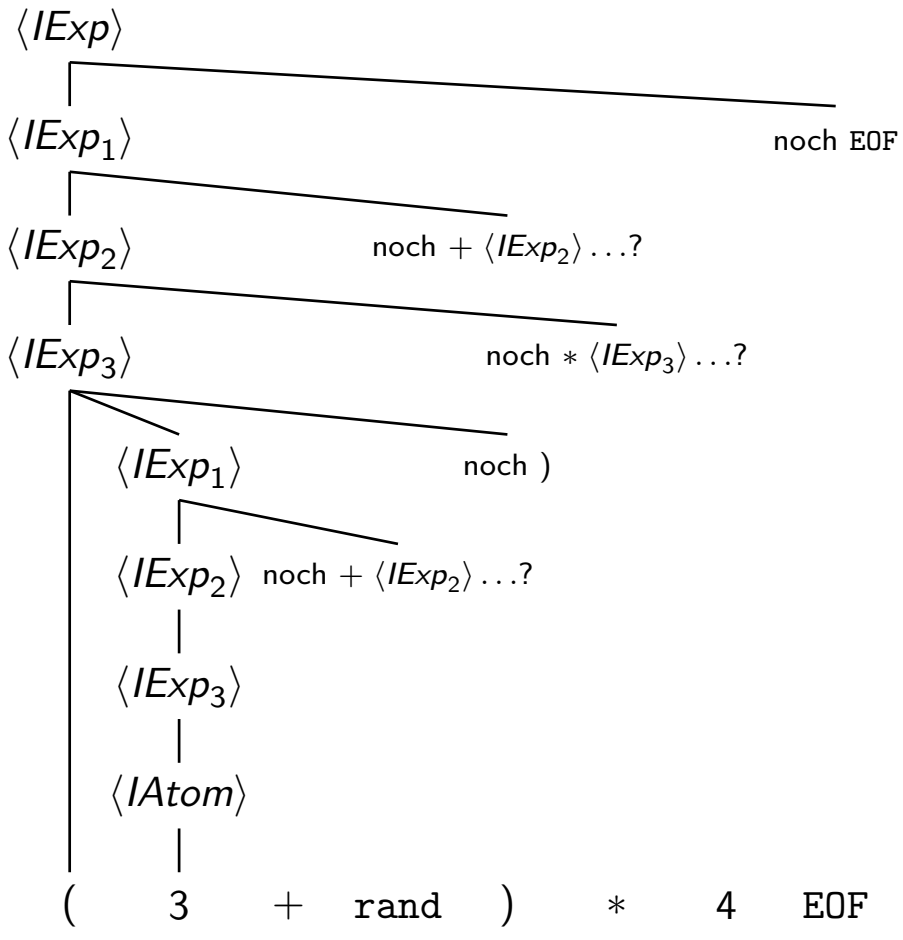
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



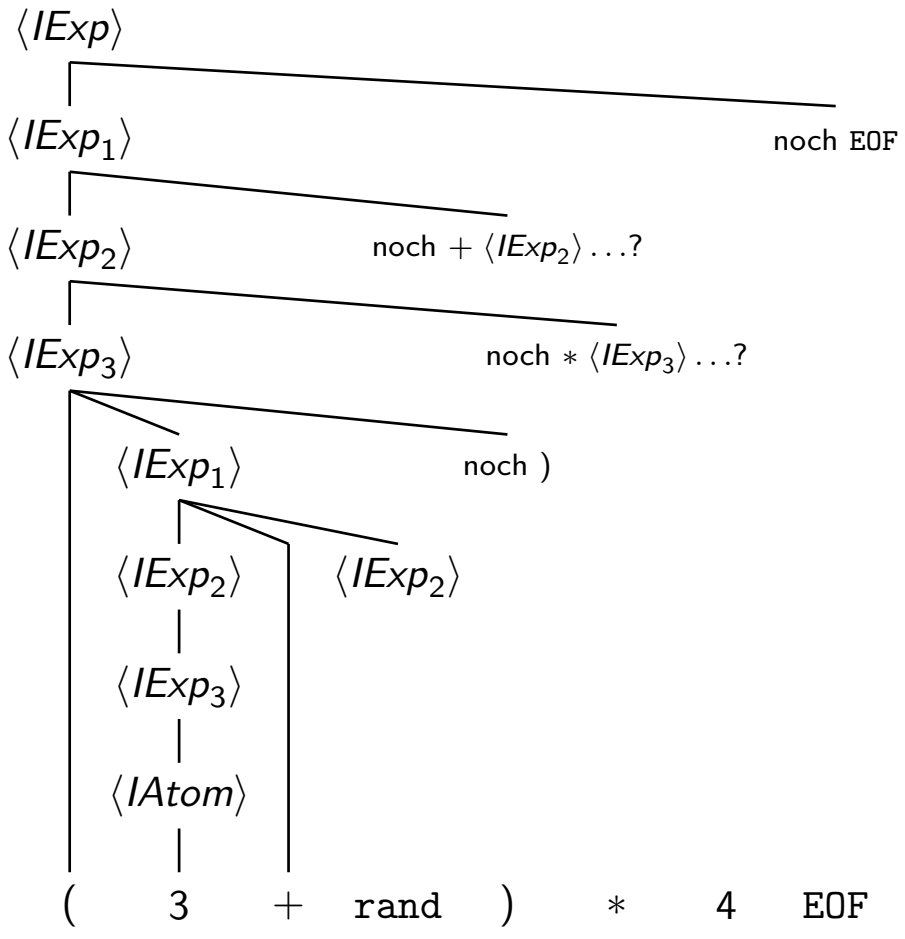
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



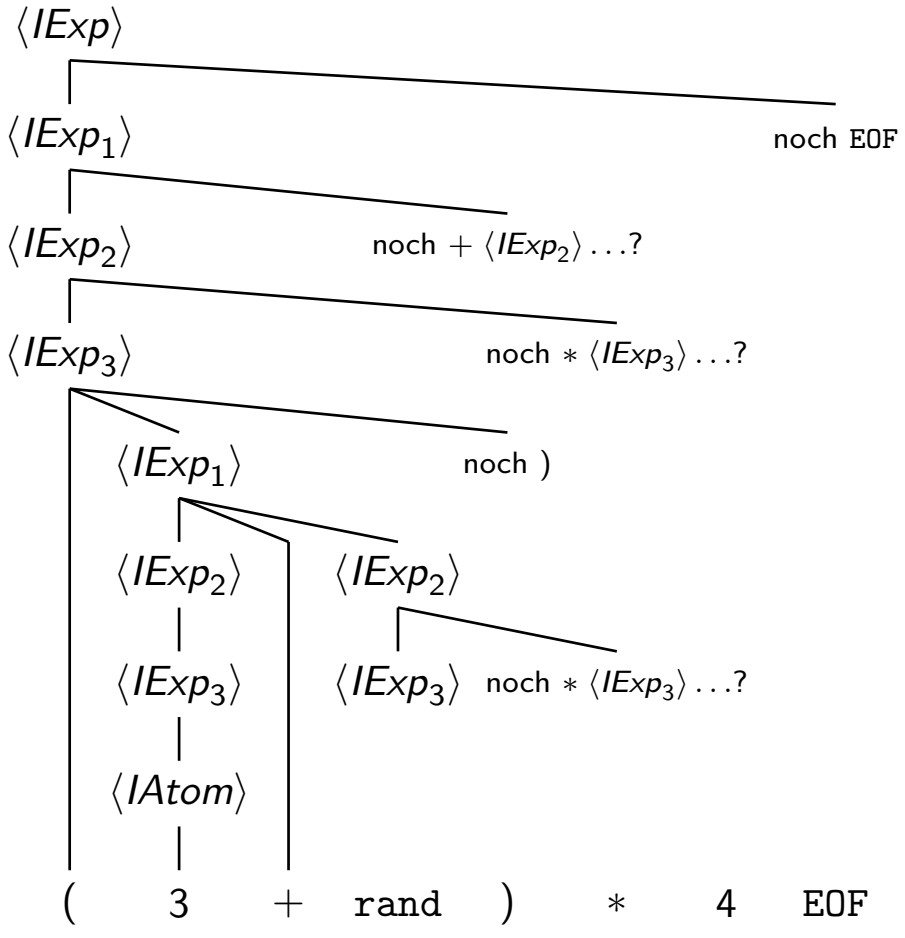
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \ \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$



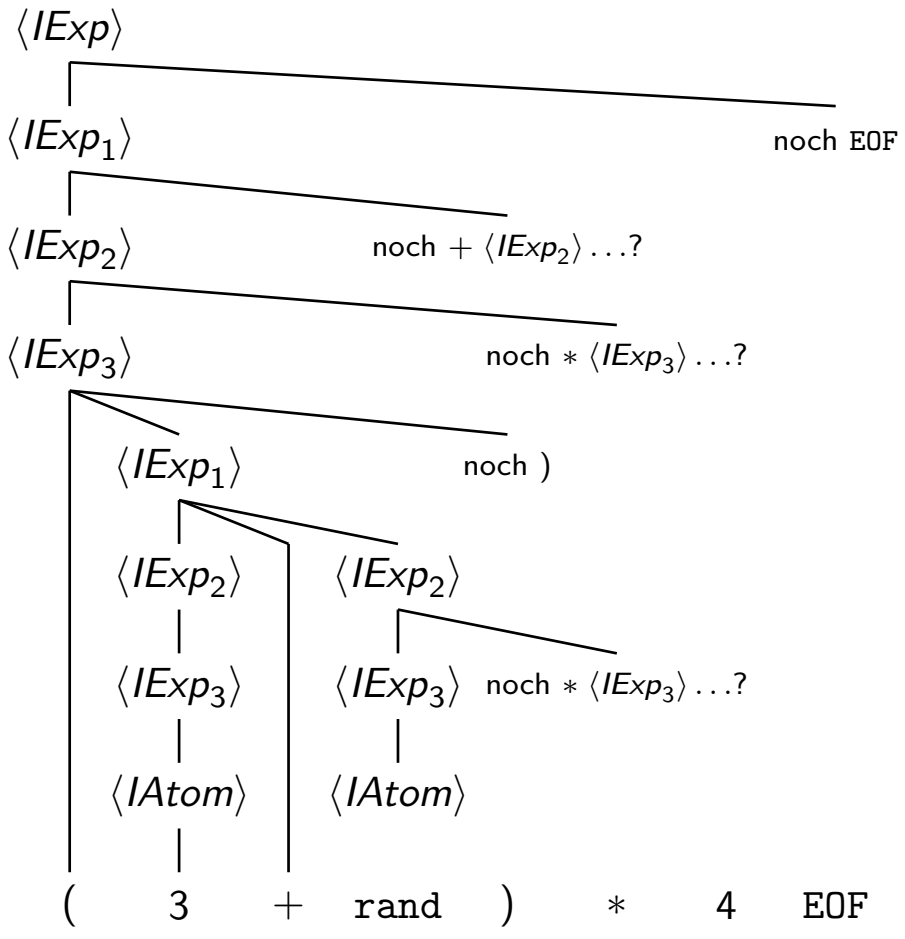
Beispiel LL(1)-Parsen

$$
\begin{array}{lcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



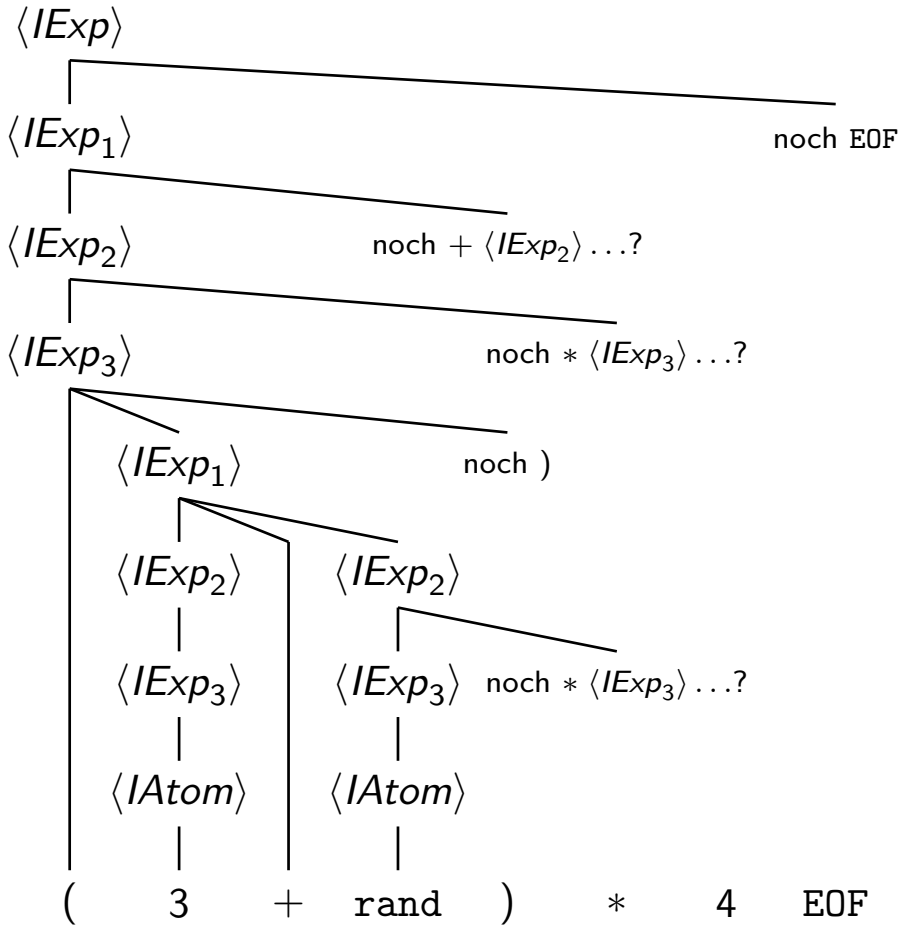
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



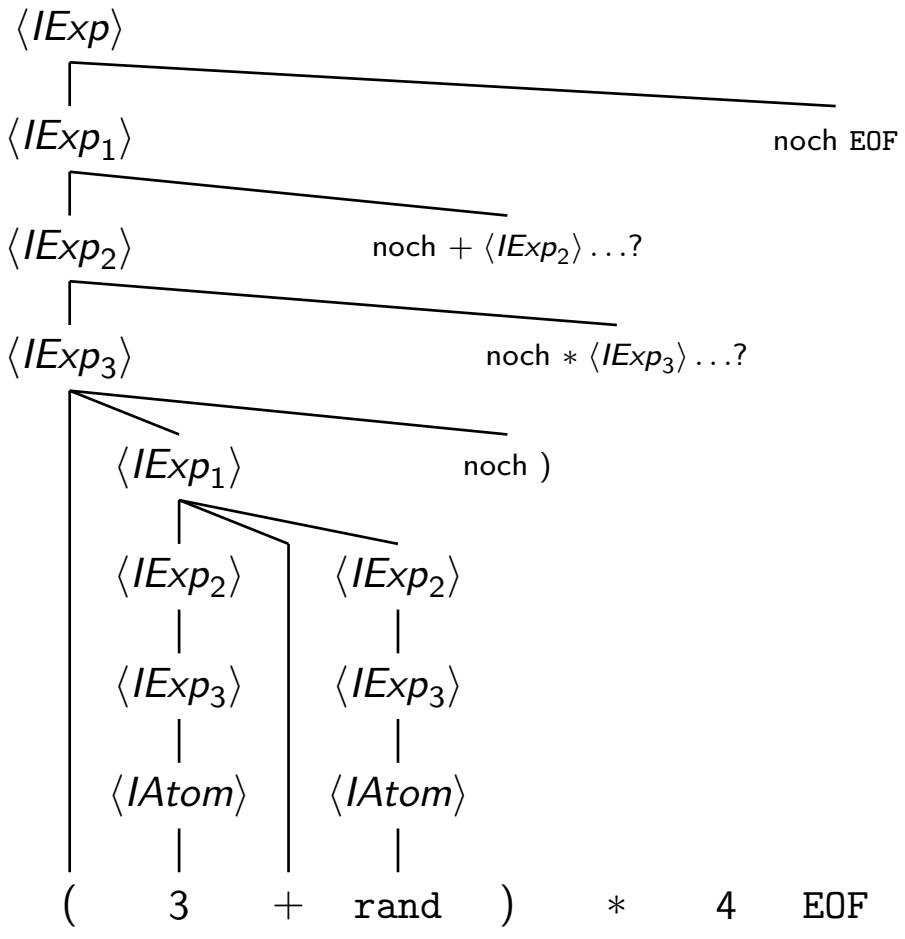
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



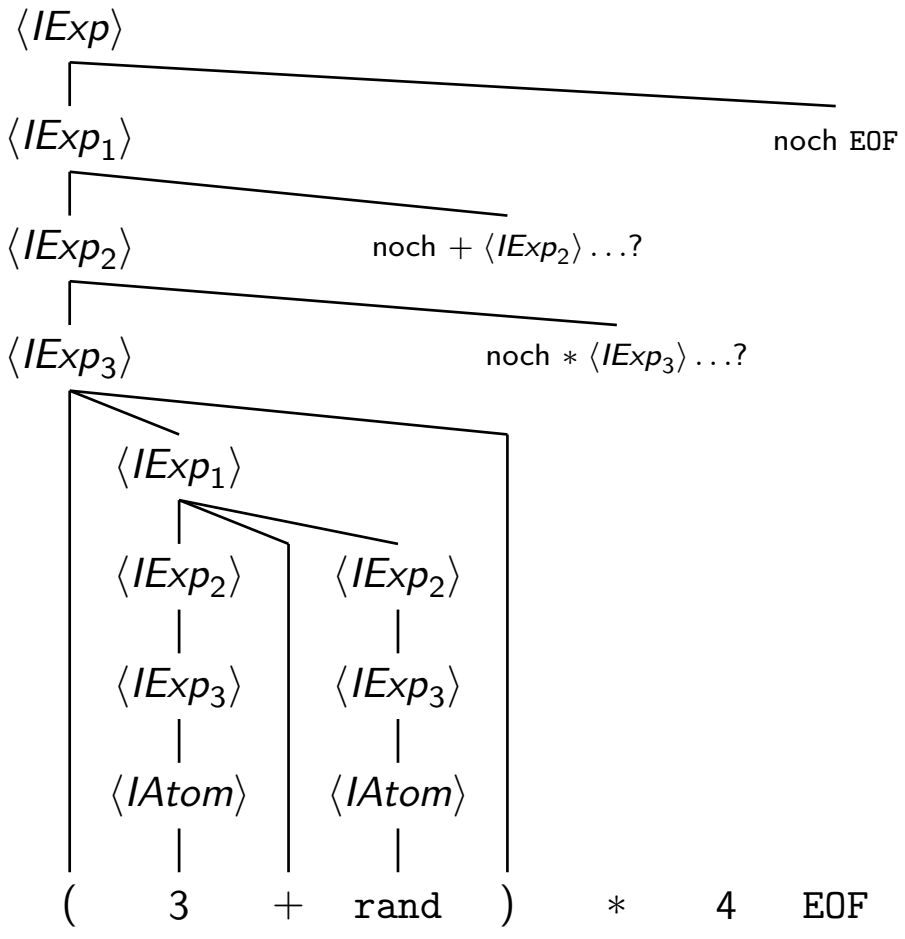
Beispiel LL(1)-Parsen

$$
\begin{array}{lcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



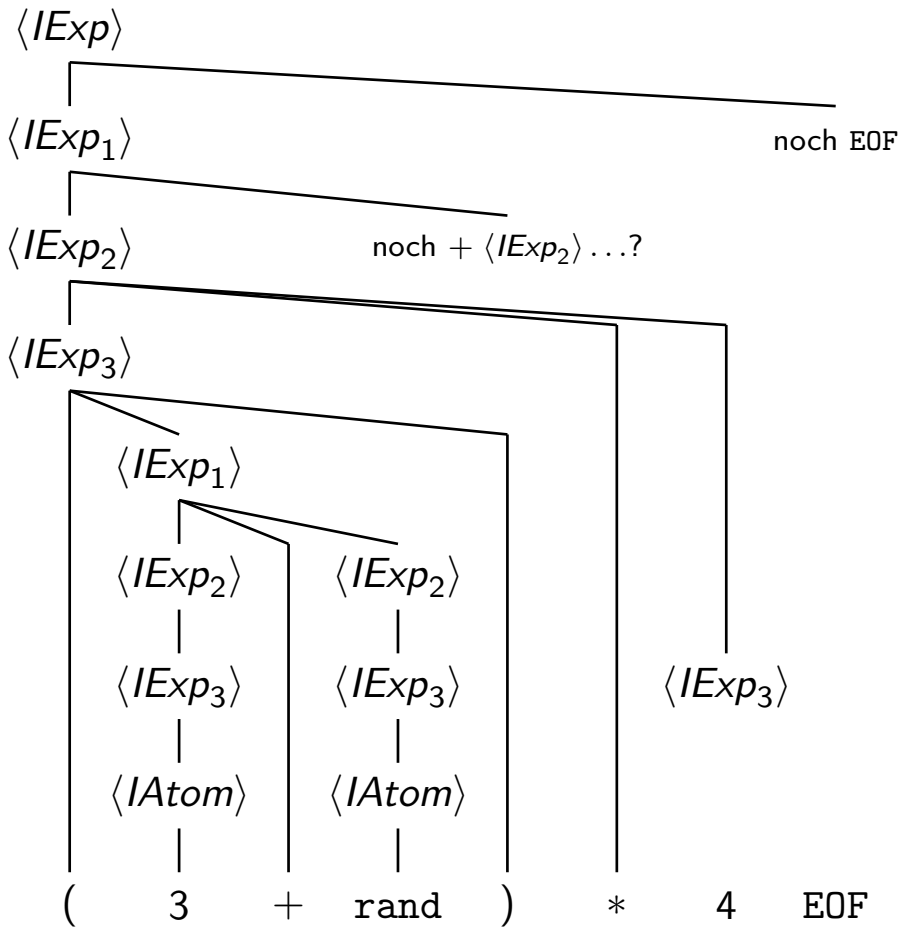
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \ \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$



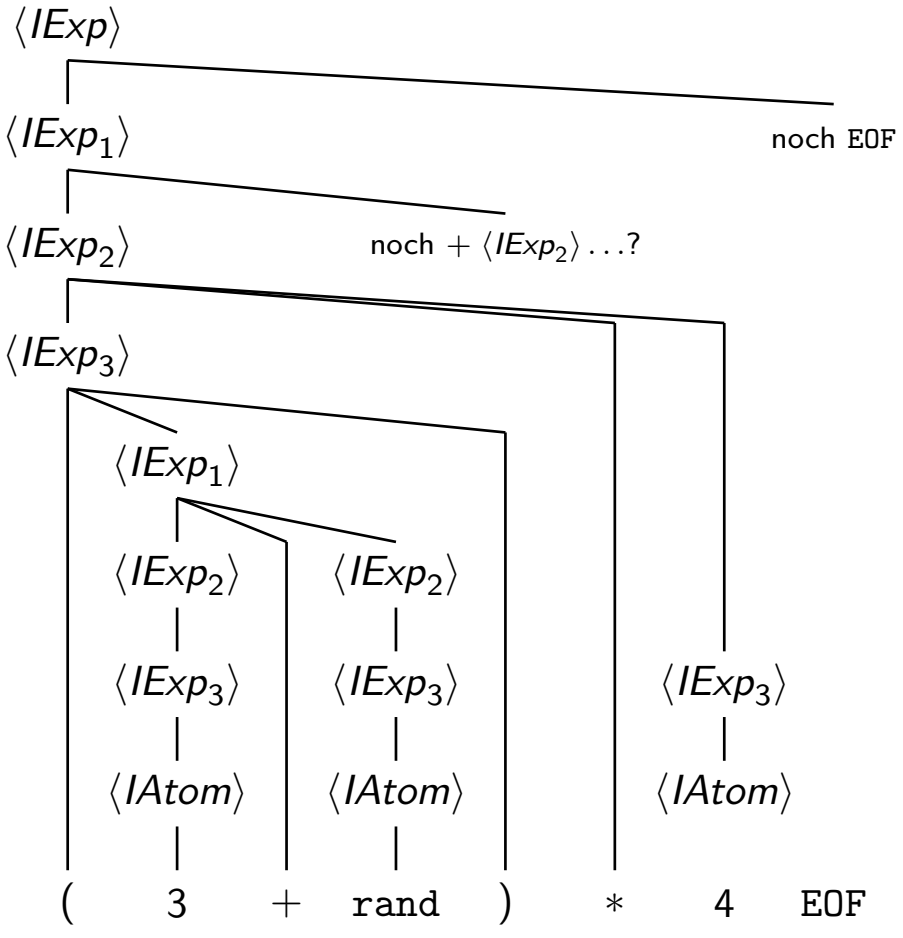
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



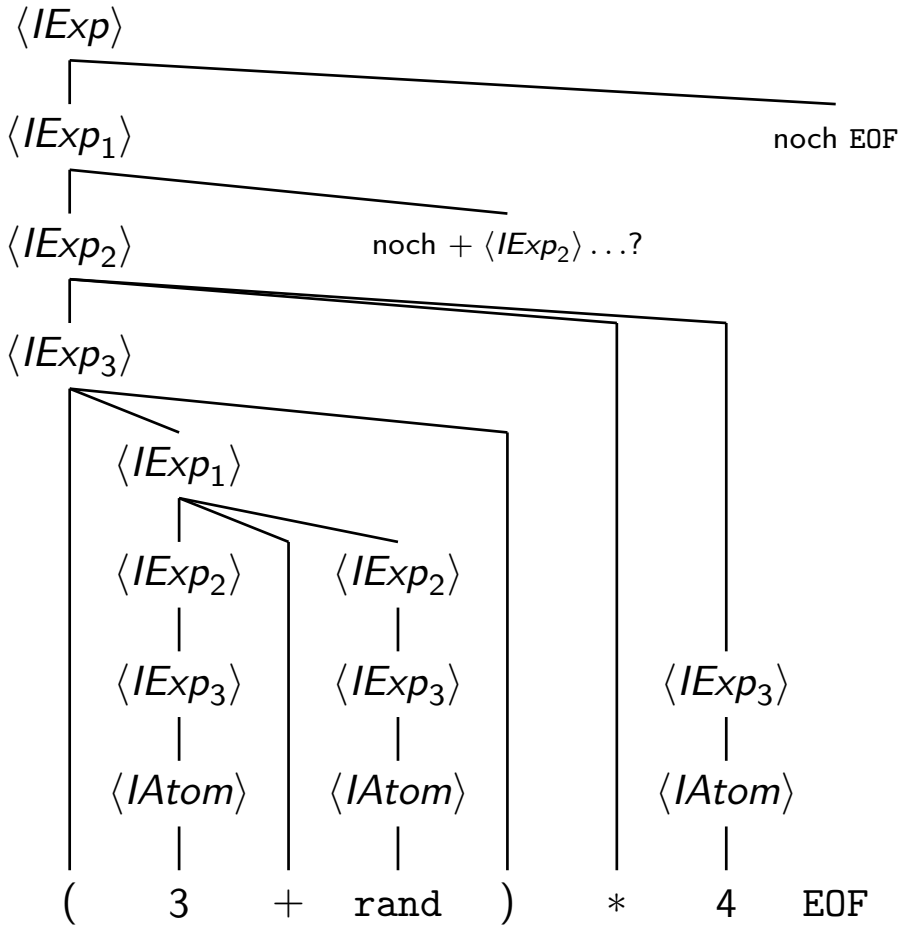
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



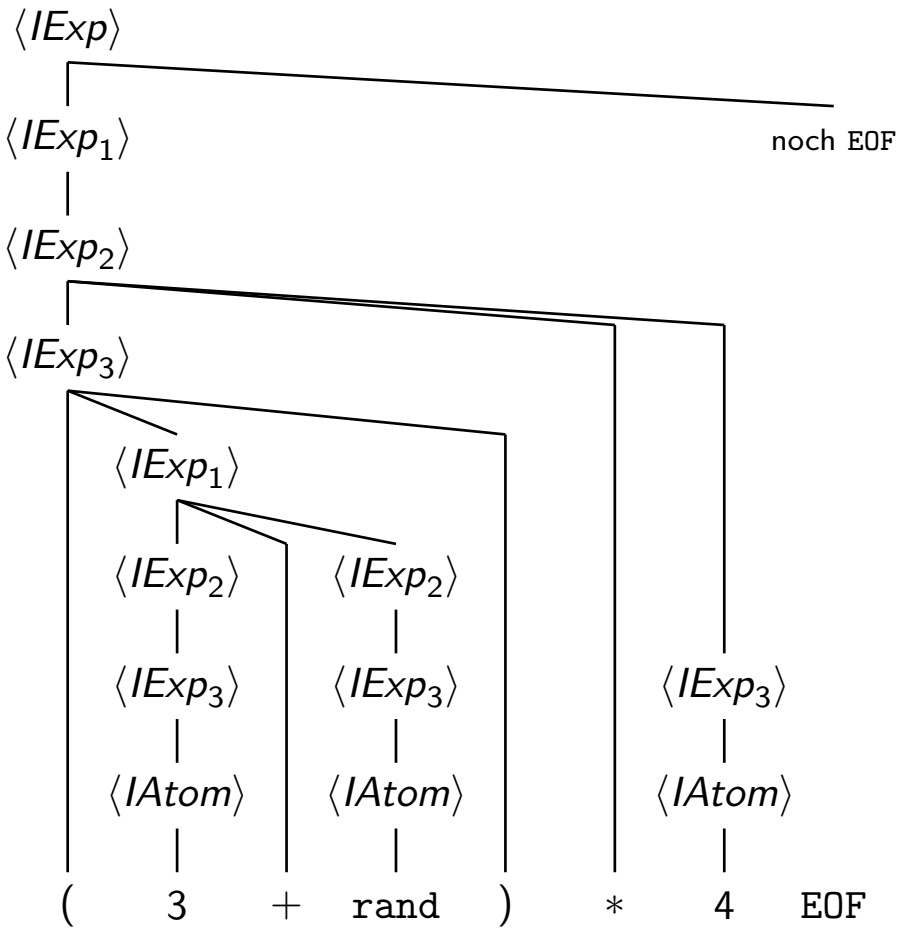
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \ \left(+ \ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \ \left(* \ \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$



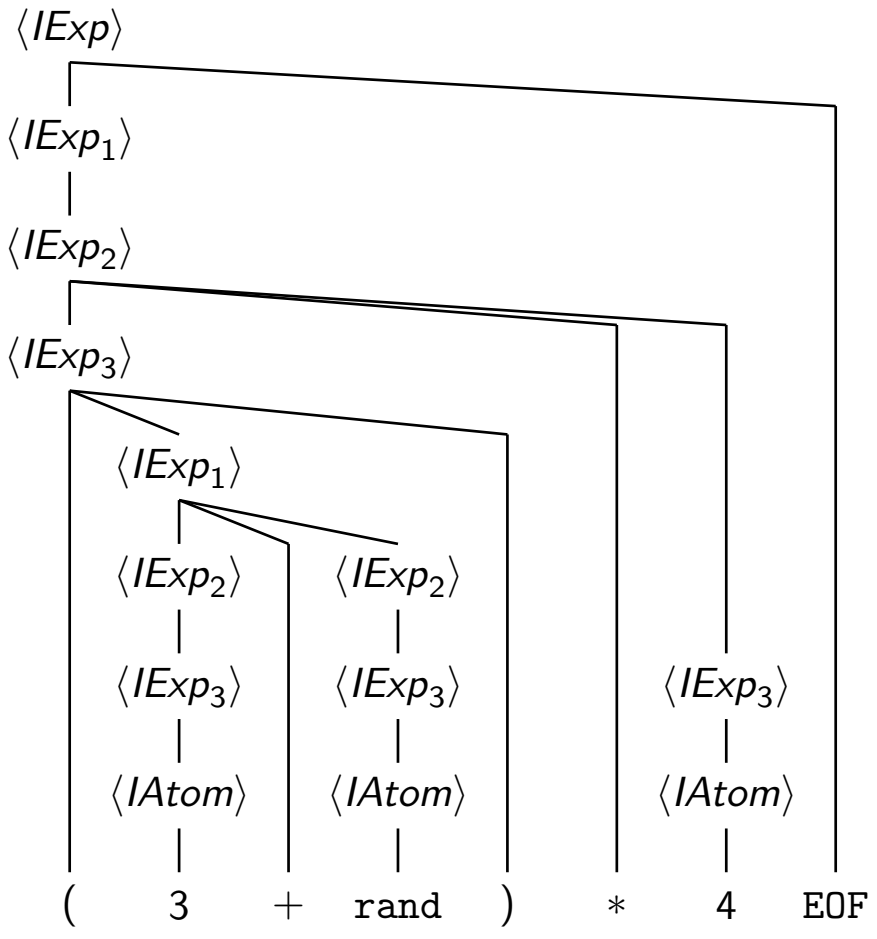
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \ \left(+ \ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \ \left(* \ \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$



Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



Zu beachten

- Ableitungsbaum bzgl. konkreter Syntax beim LL(k)-Parsen ist Baum der rekursiven Aufrufe, Rückgabe sollte aber Struktur bzgl. abstrakter Syntax sein (hier in Bsp. nicht gezeigt)

- Bsp.: "IntParser" auf Homepage

- Parser und Lexer kann man auch automatisch mit *Parsergeneratoren*, aus Grammatik erzeugen lassen – hier nicht weiter behandelt, in Compilern Parser normalerweise auch von Hand geschrieben

- 12-Variablen Regel in Spezifikation $\rightsquigarrow$ Menge der korrekten AntBrain-Programme nicht kontextfrei

  Regel beim Parsen zuerst einmal ignorieren, Anzahl vorkommender Variablen in Programm kann leicht danach an Baum in abstrakter Syntax bestimmt werden

Operationale Semantik

Semantik

legt Bedeutung von Programmen einer Sprache fest

verschiedene Arten von Semantik

- operationale Semantik: *wie* wird das Programm *ausgeführt*?
  - *small-step*: erkläre, wie man schrittweise zum Ergebnis kommt
  - *big-step*: setze Programme und Ergebnisse in Relation
- denotationelle Semantik: *welche Funktion* wird von dem Programm berechnet?
- axiomatische Semantik: *welche Eigenschaften* erfüllt das Programm?

hier: small-step operational

Notwendigkeit einer formalen Semantik

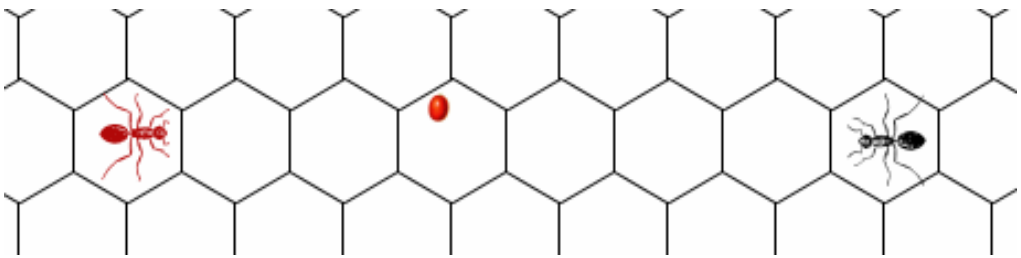
Programmierer und Compiler / Interpreter sollten Programm auf gleiche Art interpretieren

- unvermeidbar bei sicherheitsrelevanten Anwendungen
- evtl. störend sonst, z.B. browser-abhängige Darstellung von HTML-Dokumenten, Interpretation von JavaScript-Programmen

Bsp.:

```
while true {
  if sense(here,food)
  then pickup
  else skip;
  walk
}
```

```
while true {
  if sense(here,food)
  then pickup
  else walk
}
```



Operationale Semantik

zur Erinnerung: small-step Semantik erklärt schrittweise Ausführung eines Programms

Schritte zeichnen sich aus durch

- Zustandsänderungen in der ausführenden Maschine
- Änderungen des auszuführenden Programms

Bsp.: führe `if x > 0 then { turn(1); walk } else skip` aus

1. werte `x > 0` aus (evtl. auch schrittweise)
2. je nach Ergebnis führe `turn(1); walk` oder `skip` aus

jetzt nur noch formal fassen ...

Formale operationale Semantik

kann z.B. angegeben werden durch Funktion

step: *Programmzustände* $\times$ *Eingaben* $\rightarrow$ *Aktionen* $\times$ *Programmzustände*

Programmzustand = Variablenbelegung ρ und noch auszuführende Programme $P_1 : \ldots : P_k$

Funktion *step* wird von Interpreter realisiert

Ausführen des Programms P:

- zu Beginn alle Variablen undefiniert: $\rho_0(x) = \perp$ für alle x
- Eingabe im i-ten Schritt ist σ_{i-1}
- iteriere:
  1. berechne $step((\rho_0, P), \sigma_0)$, Ergebnis ist $(a, (\rho_1, P_1 : \ldots : P_n))$
  2. führe a aus (Bsp.: Ausgabe)
  3. berechne $step((\rho_1, P_1 : \ldots : P_n), \sigma_1)$ ...

Variablenbelegungen

Bsp.: was ist der Wert des Ausdrucks `x > 0` ?

Antwort hängt von Wert von `x` ab

essentieller Teil des Programmzustands: Werte der im Programm auftretenden Variablen

Variablenbelegung ist endliche Funktion von Variablen auf Werte, kann auch partiell sein (in Java z.B. durch Wert `null` realisiert)

Verwendung z.B. in der Auswertung von Ausdrücken: Notation $\|\mathtt{x} > 0\|_\rho$ zu lesen als "Wert des Ausdrucks $\mathtt{x} > 0$ *unter* der Variablenbelegung ρ"

Bsp.: falls $\rho(\mathtt{x}) = 5$, dann $\|\mathtt{x} > 0\|_\rho = 1$

Stacks

wieso eigentlich "noch auszuführende Programme $P_1 : \ldots : P_n$"?

Bsp.1: wie wird `while true do { x = x-1; walk }` ausgeführt?

- führe `x = x-1; walk` aus
- führe danach wieder `while true do { x = x-1; walk }` aus

und wie wird `x = x-1; walk` ausgeführt?

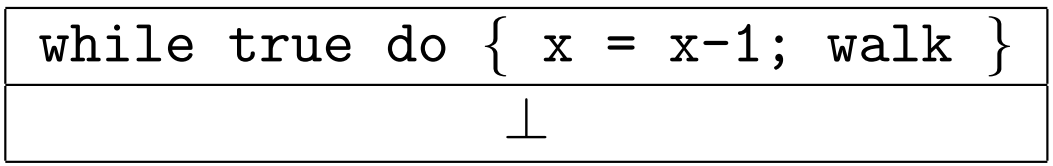
- führe erst `x = x-1` aus
- führe danach `walk` aus, aber noch *vor* nächstem Schleifendurchlauf!

Stack ist geeignete Struktur, um die Liste der noch auszuführenden Programme in ihrer Reihenfolge zu merken

Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

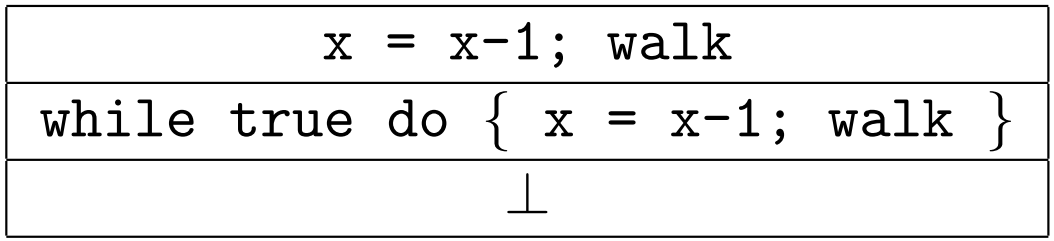
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

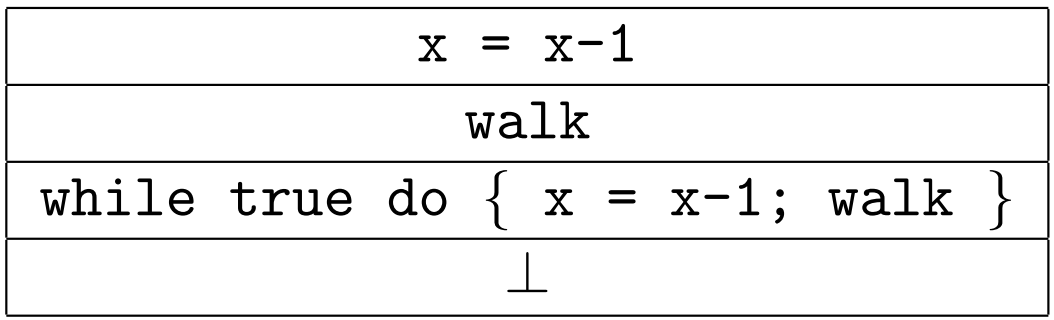
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

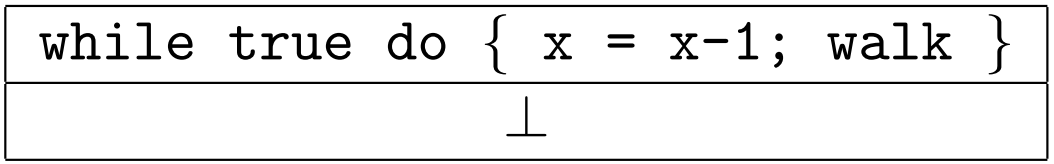
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)

`walk`
`while true do { x = x-1; walk }`
$\perp$

Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



Interpreter für AntBrain

zur Erinnerung: muss inbesondere *step* realisieren

Modellierung (Vorschlag):

- Klasse `Ant` realisiert Ameise mit Interpreter für AntBrain mittels Funktion *step*
- Eingabe σ als Parameter an *step* (hier: zwei Zellen)
- Programmzustand $(\rho, P_1 : \ldots : P_n)$ über Instanzvariablen realisiert, weil sie aktualisiert und in nächstem Schritt wiederverwendet werden müssen

```
class Ant {
  private Random random;
  private Map<String, Integer> rho;
  private Deque<Prg> stack;              // Deque ersetzt Stack
  ...
  public Action step(Cell here, Cell ahead) {
    ...
  }
}
```

Zu beachten

- Interpreter kann wiederum mehrere Schritte (auf der JVM) brauchen, um einen Schritt in der operationalen Semantik von AntBrain auszurechnen
- es bietet sich an, Funktion `step` rekursiv zu implementieren
- Termination der `step`-Funktion i.A. ein Problem, hier aber gewährleistet (wegen Verkleinerung des obersten Programms auf dem Stack)

Endabnahme

Organisatorisches

wie angekündigt finden statt in der 1. Woche nach der Vorlesungszeit

Dauer ca. 90min

Ort: wie Testabnahme im Z1.09 in der Oettingenstr.

Videoaufnahme; wird nur zur Protokollierung der Prüfung verwendet

Ablauf

1. jeder Teilnehmer der Gruppe präsentiert einen Aspekt der entwickelten Software (Bsp.: Server, GUI, Modellierung, . . .)

   Dauer ca. 5min, dazu noch ca. 5min Zeit für Fragen

   Einsehen einzelner Teile des relevanten Sourcecodes und Vorführung des (relevanten Teils) des Programms auf lokalem Rechner sollte schnell möglich sein; entweder auf Nachfrage hin oder weil eingebettet in Präsentation

2. Gruppe führt Programm vor

   Funktionalität von Server und Client wird (hoffentlich) jeweils mit fremden Server und Client getestet

   Einsehen des Sourcecodes, der Dokumentation, etc. sollte auch hier schnell möglich sein

3. weitere Fragen bzgl. nicht angesprochener Aspekte

Hinweise zur Endabnahme

- Präsentation vorbereiten und üben
- Einzelteile innerhalb der Gruppe abstimmen z.B. bzgl. Reihenfolge, Inhalt, Präsentationstechnik, etc.
- Präsentation (als ganzes) soll Zuhörer überzeugen, dass gute Software entwickelt wurde
- Zeit nicht darauf verwenden, uns zu erzählen, was wir schon kennen (z.B. die Spezifikation); eigene Ideen herausstellen
- Funktionalität über Gruppengrenzen hinweg kann vorher getestet werden

Präsentationen im Allgemeinen

Entwurf von Präsentation beginnt mit Entscheidungen bzgl.

1. Inhalt
   - was genau möchte man präsentieren? was nicht? was möchte man sagen?
2. Wahl des Mediums
   - muss auf Inhalt abgestimmt sein
   - Folien bieten sich oft an (kein verwirrendes Herumscrollen)
3. Struktur der Präsentation
   - top-down Ansatz: z.B. Problem $\rightsquigarrow$ Ansatz $\rightsquigarrow$ Ausführung
   - Zeit lassen für Fragen

Die Verwendung von Folien

i.A. sicheres Medium (gegen Ausfall von Netzverbindung, Nicht-Finden von Codestellen, etc.)

- sind nur dazu da, Gesagtes visuell zu unterstützen
  - Stichpunkte / Bilder statt vollständige Sätze
  - sollten für sich alleine sehr unvollständig und unverständlich sein
  - Überfüllung vermeiden
- je nach Inhalt und Füllung ca. 1-2min pro Folie einplanen
- zwingen dazu, Präsentation im Detail zu strukturieren und auf Wesentliches zu konzentrieren
  - Bsp.: Sourcecode einer ganzen Klasse passt nicht auf eine Folie; stattdessen verwendete Signaturen und Datenstrukturen benennen, evtl. reduziertes UML-Diagramm
- Farben eignen sich für Hervorhebungen

Hinweise zu Präsentationen i.A.

- Präsentieren ist das Informieren der Zuhörer; diese haben limitierte Auffassungsgabe; auf wesentliche Punkte konzentrieren und diese klar und verständlich formulieren
- nicht nervös sein; Zuhörer wollen etwas hören; man selbst ist der Experte auf dem Gebiet
- auf Positives konzentrieren; Formulierungen der Form "*ich muss gleich sagen, dass ich nicht wusste, wie ...*" sind fehl am Platz; Schwachstellen dennoch klar benennen
- dafür sorgen, dass man verstanden wird; klar und deutlich und nicht zur Wand gewandt sprechen
- Kommunikation mit Zuhörern während der Präsentation
  - nicht arrogant / langweilig / etc. erscheinen
  - nicht über ihre Köpfe hinwegreden
  - sich nicht in uninteressanten Details verlieren
  - auf Zwischenfragen reagieren können; vorbereitet sein

Zufallszahlen

Zufallszahlen in AntBrain

Spezifikation, Teil II:

> *Zum Beispiel könnte ein Objekt vom Typ* `Match` *die Spielfelder nach jeweils 1000 Spielrunden speichern; bei einer Anfrage nach den Aktionen in der 1012. Runde könnte es dann mit seiner Berechnung in der 1000. Runde anfangen und müsste nur noch zwölf Runden ausrechnen.*

Bei Neuberechnung müssen die gleichen „Zufallszahlen“ benutzt werden wie beim ersten Mal.

Zufallszahlen

Wichtig in vielen Anwendungen

- Spiele (Würfel seit über 5000 Jahren benutzt)
- Stichprobennahme
- Simulation physikalischer Vorgänge
- Approximation schwer lösbarer Probleme durch wiederholte Zufallsexperimente
  - Primaltests mit großer Richtigkeitswahrscheinlichkeit
  - numerische Integration
  - ...
- Kryptographie (Erzeugung zufälliger Schlüssel)

Erzeugung von Zufallszahlen

Echte Zufallszahlen durch physikalische Prozesse

- Würfeln, Münzen werfen
- radioaktive Zerfallprozesse
- atmosphärisches Rauschen
- Quanteneffekte
- Lavalampen, CCD-Kameras im Dunkeln

langsam, benötigt spezielle Hardware

Tabellen von Zufallszahlen, z.B.
RAND Corporation, A Million Random Digits with 100,000 Normal Deviates, 1951
(Taschenbuch-Neuauflage 2002)

Erzeugung von Zufallszahlen

Pseudozufallszahlen

Deterministischer Rechner fängt mit einigen zufälligen Bits (Seed) an und erzeugt daraus eine Folge zufällig *aussehender* Zahlen.

Folge der Zahlen hängt nur vom Seed ab.

schnell, keine Zusatzhardware nötig, wiederholbare Experimente

Beispiel: `java.util.Random`

- Konstruktor `Random(long seed)`
- Konstruktor `Random()` benutzt als Seed die Zeit in ms seit dem 1.1.1970.
- Methoden der Klasse sind komplett deterministisch
- Random ist serialisierbar: beim Serialisieren wird Seed gespeichert.

Kopieren von `Random`-Objekten

Ein Objekt `random` vom Typ `java.util.Random` kann kopiert werden, da diese Klasse serialisierbar ist.

```
Random copy = null;
ObjectOutputStream o = null;
try {
  ByteArrayOutputStream buf = new ByteArrayOutputStream();
  o = new ObjectOutputStream(buf);
  o.writeObject(random);
  ObjectInputStream i = new ObjectInputStream(
        new ByteArrayInputStream(buf.toByteArray()));
  copy = (Random) i.readObject();
} catch (Exception ex) {
  assert(false);
}
```

Jetzt liefern `random` und `copy` die gleiche Folge von „Zufallszahlen“.

Pseudozufallszahlen

Verschiedene Qualitätsanforderungen (z.B. bzgl. Vorhersagbarkeit) für verschiedene Zwecke:

- Windows Solitaire
- Online Casino
- Kryptographie

Beispiel: Debian SSL-Bug (2005–2008)

- Seed für Pseudozufallsgenerator hing nur von Prozessnummer ($\leq 32768 = 2^{15}$) ab.

$\Rightarrow$ Bei der zufälligen Erzeugung von Schlüsseln konnten nur 2^{15} verschiedene Schlüssel herauskommen (statt z.B. 2^{1024} bei einer typischen Schlüssellänge von 1024 Bit).

$\Rightarrow$ Schlecht erzeugte Zufallszahlen machten SSL-Verschlüsselung und SSH-Zugänge unsicher.

Pseudozufallszahlengeneratoren

Verschiedene Verfahren, die verschiedenen Ansprüchen an Geschwindigkeit und Qualität genügen.

- Lineare Kongruenzen (`java.util.Random`)
  schnell, leicht zu implementieren, nicht gut genug z.B. für Anwendungen in der Kryptographie
- Generatoren auf Basis von Hashfunktionen, z.B. für kryptographische Anwendungen (`java.util.SecureRandom`), langsam
- Blum-Blum-Shub-Generator: wenn eine bestimmte komplexitätstheoretische Vermutung gilt, dann kann kein effizienter Algorithmus diese pseudozufälligen Zahlen von echten Zufallszahlen unterscheiden. Für viele Anwendungen zu langsam.

Pseudozufallszahlen durch lineare Kongruenzen

Einfache Methode zur Erzeugung von Pseudozufallszahlen [Lehmer, 1949]

Parameter $a, c, m \in \mathbb{N}$.

Seed $x_0 \in \mathbb{N}$ mit $0 \leq x_0 < m$.

Definiere Folge von natürlichen Zahlen $x_0, x_1, x_2, \ldots$ durch

$$x_{i+1} = a \cdot x_i + c \bmod m$$

Betrachte $\frac{x_1}{m}, \frac{x_2}{m}, \ldots$ als Folge von Zufallszahlen im Intervall $[0, 1)$.

(Die Anzahl der Nachkommastellen, die sinnvoll benutzt werden können, hängt von m ab.)

Pseudozufallszahlen durch lineare Kongruenzen

Beispiel: `java.util.Random`

$$a = 25214903917 \qquad c = 11 \qquad m = 2^{48}$$

$$x_{i+1} = a \cdot x_i + c \bmod m$$

(Diese Zahlen werden seit den 80iger Jahren im Zufallszahlengenerator von Unix System V benutzt.)

Der i-te Aufruf der Funktion `nextInt()` liefert die 32 führenden Bits der 48-Bit-Zahl x_i zurück.

leicht vereinfachter Quellcode:

```
long seed;
public int nextInt() {
  seed = (a * seed  + c) & ((1L << 48) - 1);
  return (int)(seed >> 16);
}
```

Pseudozufallszahlen durch lineare Kongruenzen

Auswahl der Parameter m, a und c

- möglichst große Periode: Die Folge $x_1, x_2, \ldots$ wird nach höchstens m Schritten periodisch

$$x_{i+1} = a \cdot x_i + c \bmod m$$

- statistische Tests, z.B.
  - χ^2-Test: alle Zahlen kommen mit der erwarteten Häufigkeit vor
  - Spektraltest: betrachte Verteilung von Tupeln $(x_i, x_{i+1}, \ldots, x_{i+t})$
  - viele mehr ...

Generator von Blum, Blum und Shub

Auch Pseudozufallsgeneratoren, die Pseudozufallszahlen von hoher Qualität liefern, sind durch einfache Algorithmen gegeben.

Beispiel: Generator von Blum, Blum und Shub (1982)

Folge von Zufallszahlen $x_0, x_1, \cdots \in \mathbb{N}$ definiert durch

$$x_{i+1} = x_i^2 \bmod N,$$

wobei x_0 und N teilerfremd sind und $N = p \cdot q$ für zwei (große) Primzahlen $p \neq q$ mit $p, q = 3 \mod 4$.

Unter der Annahme, dass es keinen effizienten Algorithmus zur Faktorisierung großer Zahlen gibt, kann kein effizienter Algorithmus diese Zufallszahlen von echten Zufallszahlen unterscheiden.

Entwurfsmuster (Design Patterns)

Entwurfsmuster (Design Patterns)

In der alltäglichen Programmierarbeit tauchen viele Probleme auf, die man schon einmal gelöst hat und die man in der Zukunft wieder lösen müssen wird.

Entwurfsmuster sind Dokumentation von Lösungs- und Designansätzen.

- Wiederverwendung von erfolgreichen Entwurfsansätzen
- Dokumentation von Lektionen, die bei der Lösung von realen Problemen gelernt wurden
- Kommunikation
  - Entwurfsmuster haben Namen und erleichtern so die Beschreibung von Struktur und Funktionsweise von Programmen
  - fremde Programme, die einem bekannten Entwurfsmuster entsprechen, sind leichter lesbar

Was ist ein Entwurfsmuster?

Entwurfsmuster beschreiben Wissen über objektorientiertes Design.

Ein Entwurfsmuster

- löst ein bestimmtes Problem
- beschreibt ein *erprobtes* Lösungskonzept
- ist nicht direkt offensichtlich (es wäre nicht sinnvoll eine ohnehin offensichtliche Lösung zu beschreiben)
- erfasst das Zusammenspiel von Komponenten (das sich im Allgemeinen nicht direkt in der Programmiersprache ausdrücken lässt)

Entwurfsmuster

Ein Entwurfsmuster besteht aus folgenden Teilen:

Name	wichtig für Kommunikation
Problem	Beschreibung der Situation, in der das Muster angewendet werden kann, und deren Kontext.
Lösung	informelle Beschreibung der Komponenten des Designs, ihrer Beziehungen, Aufgaben und Pflichten sowie ihrer Kommunikation.
Beispiele	Illustration der Lösung an repräsentativen Beispielen
Evaluation	Diskussion von Vor- und Nachteilen des Entwurfsansatzes

Entwurfsmuster

Strukturmuster beschreiben die Zusammensetzung von Objekten und Klassen zu größeren Strukturen.

Beispiele: *Composite*, Adapter, Decorator, Flyweight, Proxy, ...

Verhaltensmuster beschreiben Möglichkeiten zur objektorientierten Implementierung von Algorithmen und zur Interaktion zwischen Objekten und Klassen.

Beispiele: Iterator, Observer, *Visitor*, Interpreter, ...

Erzeugungsmuster beschreiben Möglichkeiten zur Objekterzeugung.

Beispiele: *Abstract Factory*, Builder, Singleton, ...

...

Strukturmuster – Composite

Problem:

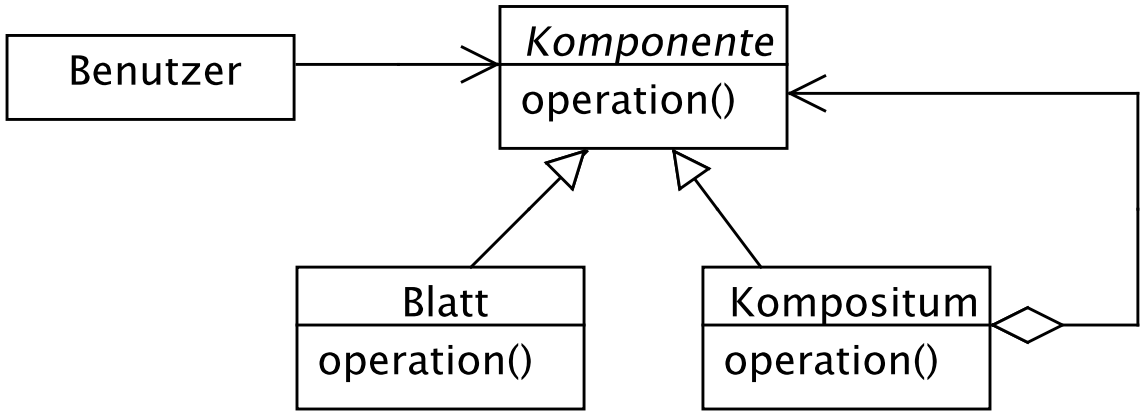
- Repräsentiere eine Menge von Objekten, in der einzelne Objekte aus anderen Objekten in der Menge zusammengesetzt sein können.
- Zusammengesetze Objekte sollen wie einzelne Objekte behandelt werden können.

Beispiele:

- Eine geometrische Figur ist entweder
  - eine Linie,
  - ein Kreis,
  - ein aus mehreren geometrischen Figuren bestehendes Bild.
- Ein arithmetischer Ausdruck ist entweder
  - eine konstante Zahl,
  - eine Variable,
  - eine Verknüpfung zweier arithmetischer Ausdrücke mit einer Binäroperation.

Strukturmuster – Composite

Das Entwurfmuster Composite beinhaltet folgende Repräsentation:



Komponente:

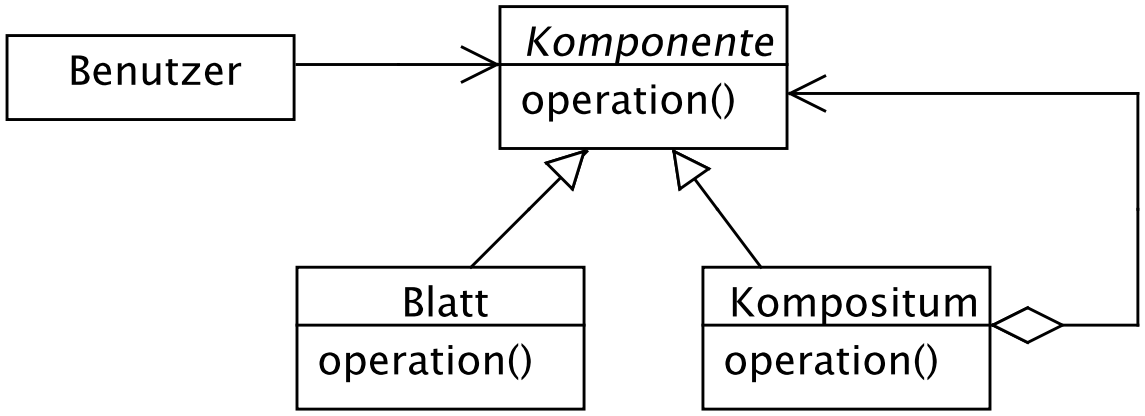
- Interface für die Objekte in der Komposition (z.B. Auswerten bei arithmetischen Ausdrücken)
- Interface für gemeinsames Verhalten (z.B. Verschieben bei geometrischen Figuren)
- Interface für den Zugriff auf die Teilkomponenten.

Blatt: primitives Objekt

Kompositum: zusammengesetztes Objekt

Benutzer: interagiert nur mit der Komponente

Strukturmuster – Composite



Beispiel: arithmetische Ausdrücke in AntBrain

Komponente	IExp
Blatt	IExpConst, IExpVar
Kompositum	IExpBinOp
operation	eval

Strukturmuster – Composite

Evaluation

- definiert Teil-Ganzes-Hierarchien von primitiven und zusammengesetzten Objekten
- Benutzung dieser Hierarchien ist einfach. Zusammengesetzte Objekte können wie primitive Objekte benutzt werden.
- Hierarchie leicht durch neue Objekte erweiterbar, ohne dass Benutzer ihren Code ändern müssen
- Benutzer kann keine neue Operation selbst definieren und zur Hierarchie hinzufügen (Beispiel: `IExp.getFreeVariables()`)
- Quelltext der `operation()` über mehrere Klassen verstreut (Beispiel: `IExp.eval()`).

Verhaltensmuster – Visitor

Probleme der Datenrepräsentation eines Composite-Patterns:

- Benutzer kann keine neue Operation selbst definieren und zur Hierarchie hinzufügen.
- Quelltext der `operation()` über mehrere Klassen verstreut.

Das Visitor-Pattern löst diese Probleme:

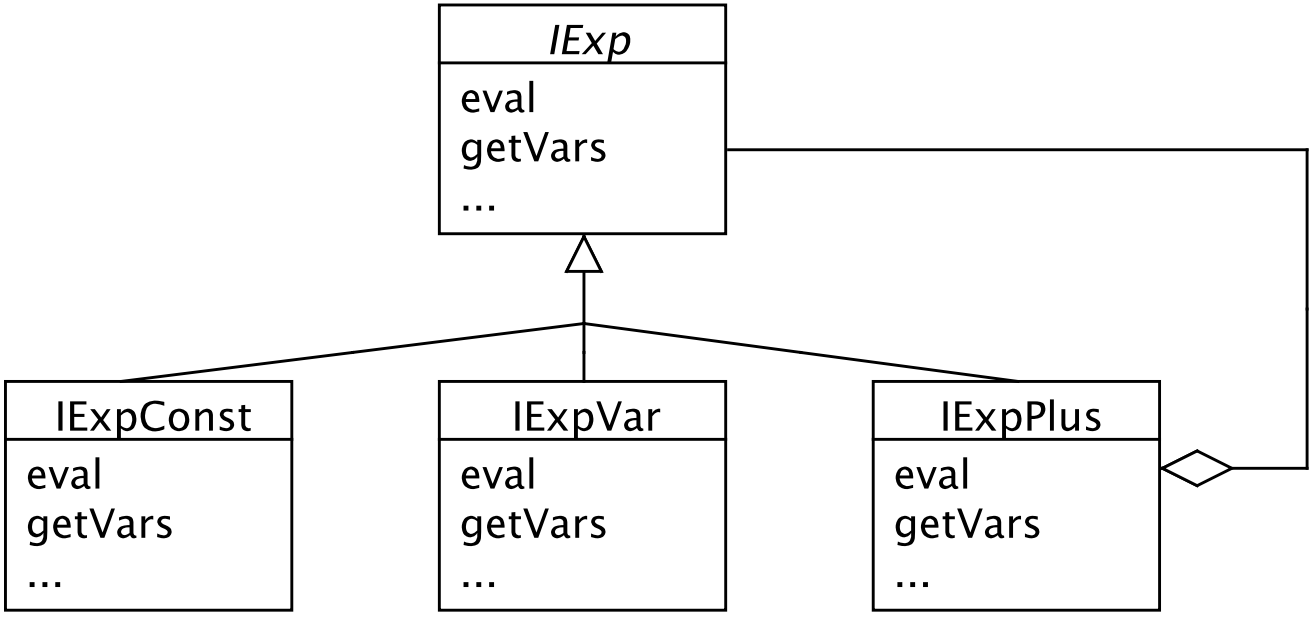
- Es erlaubt, neue Operationen zu definieren, ohne die Klassen der Struktur zu ändern.
- Es erlaubt, Operationen, die auf einer Objektstruktur ausgeführt werden sollen, kompakt zu repräsentieren.

Verhaltensmuster – Visitor

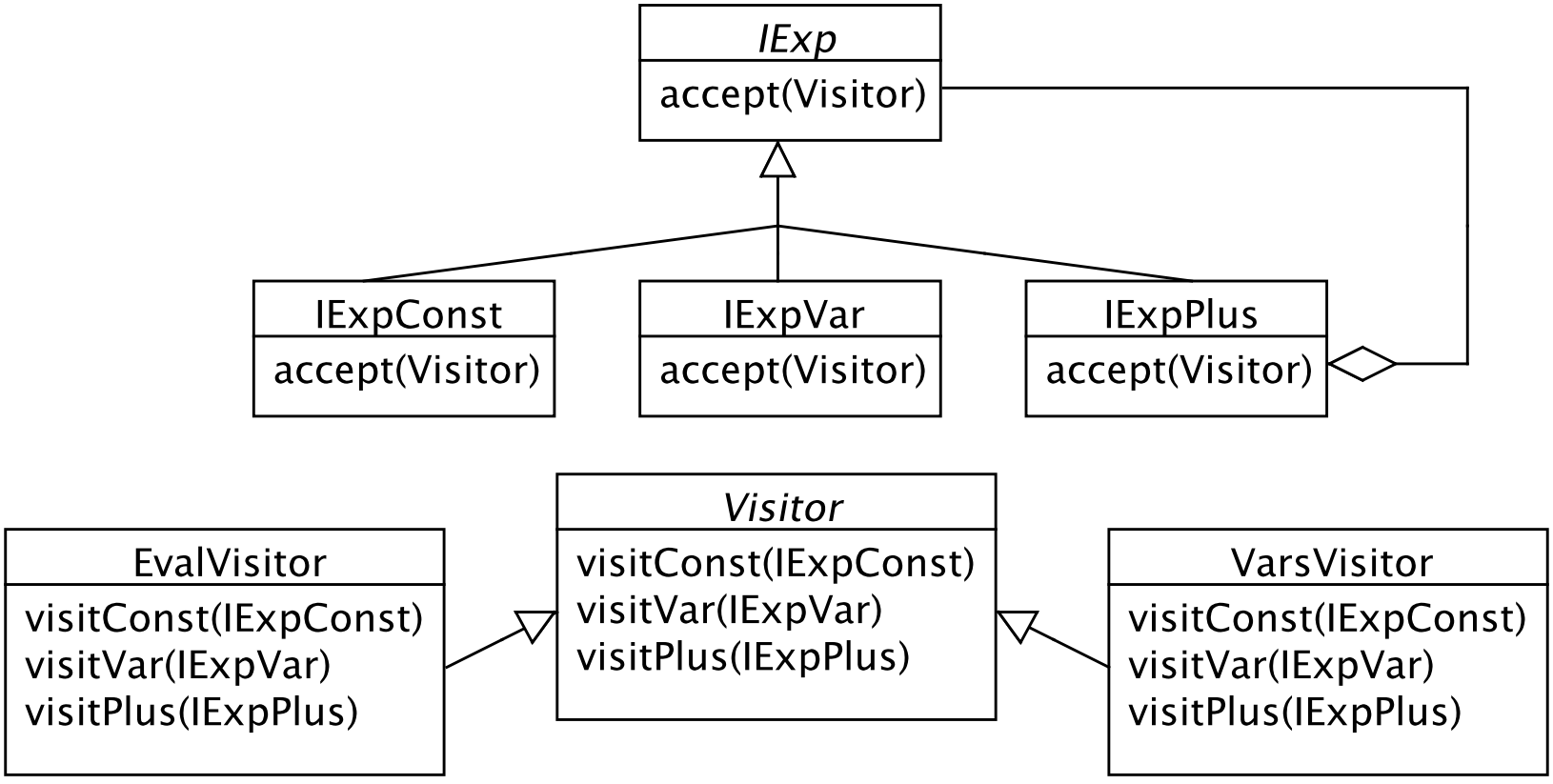
Idee:

- Sammle die Definitionen der Operationen auf der Objektstruktur in einem Visitor-Objekt.
- Ersetze die verschiedenen Operationen durch eine einzige `accept`-Methode.

Beispiel



Verhaltensmuster – Visitor



`accept(v)` in `IExpConst` durch `v.visitConst(this)` gegeben, usw.

$\Rightarrow$ Auslagerung der Methodendefinitionen in Visitor-Objekte.

Verhaltensmuster – Visitor

```
abstract class IExp {
  abstract <T> T accept(IExpVisitor<T> v);
}

class IExpVar extends IExp {
  String varName;
  <T> T accept(IExpVisitor<T> v) { return v.visitVar(this); }
}

class IExpConst extends IExp {
  Integer value;
  <T> T accept(IExpVisitor<T> v) { return v.visitConst(this); }
}

class IExpPlus extends IExp {
  IExp e1, e2;
  <T> T accept(IExpVisitor<T> v) { return v.visitPlus(this); }
}
```

Verhaltensmuster – Visitor

```
abstract class IExpVisitor<T> {
    abstract T visitVar(IExpVar v);
    abstract T visitConst(IExpConst c);
    abstract T visitPlus(IExpPlus p);
}
```

Funktionen für arithmetische Ausdrücke können nun in zentral in einem `IExpVisitor`-Objekt aufgeschrieben werden.

Beispiele:

- Auswertung eines Ausdrucks: `EvalVisitor`
- Variablen in einem Ausdruck: `VarsVisitor`

Verhaltensmuster – Visitor

Auswertung eines Ausdrucks `iexp`:

```
iexp.accept(new EvalVisitor(memory));

class EvalVisitor extends IExpVisitor<Integer> {
    Map<String, Integer> memory;

    EvalVisitor(Map<String, Integer> memory) {
      this.memory = memory;
    }
    Integer visitVar(IExpVar v) {
      return memory.get(v.varName);
    }
    Integer visitConst(IExpConst c) {
      return c.value;
    }
    Integer visitPlus(IExpPlus p) {
      return p.e1.accept(this) + p.e2.accept(this);
    }
}
```

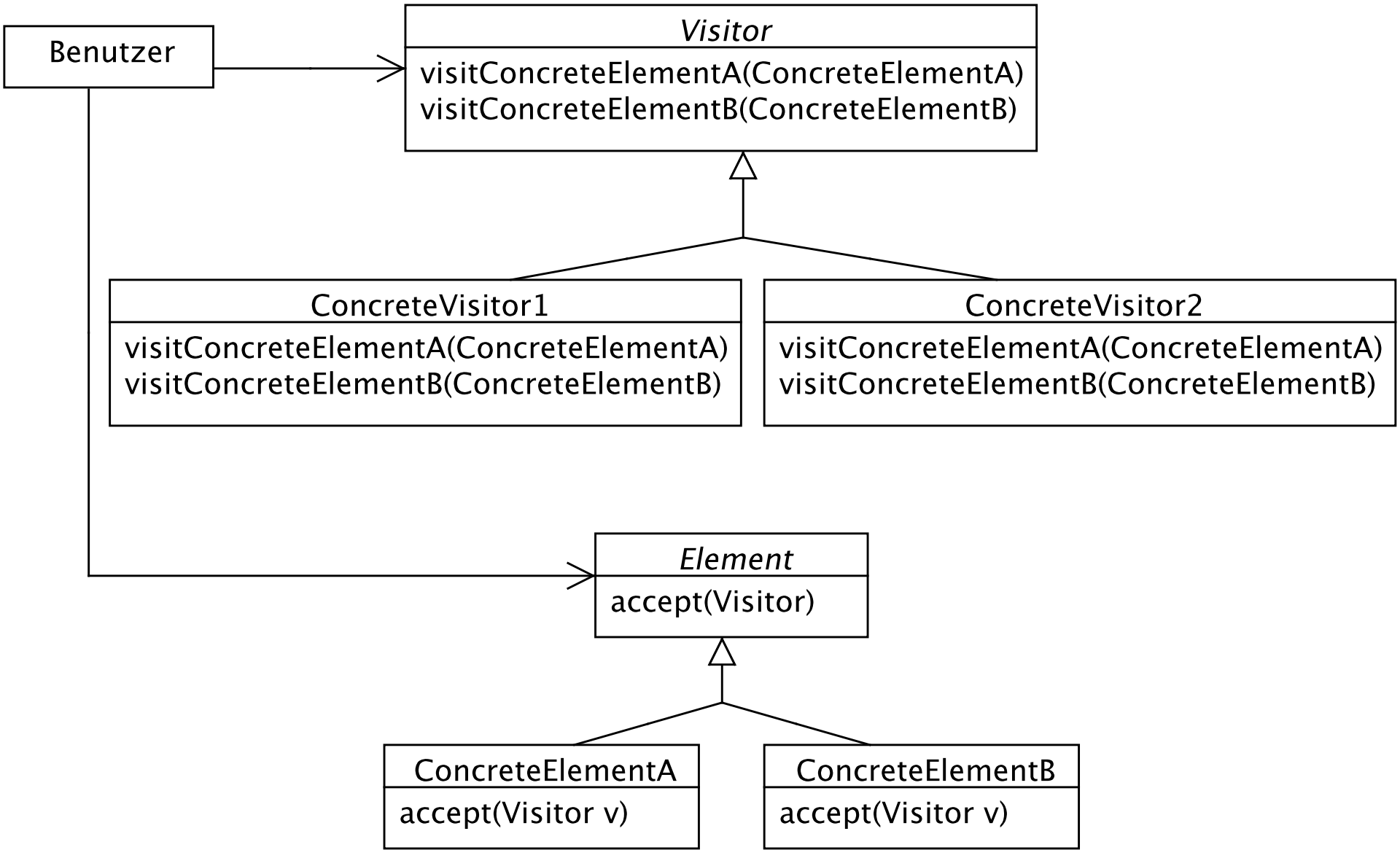
Verhaltensmuster – Visitor

Berechnung der Variablenmenge eines Ausdrucks `iexp`:

```
iexp.accept(new VarsVisitor());
```

```
class VarsVisitor extends IExpVisitor<Set<String>> {

    Set<String> visitVar(IExpVar v) {
      return java.util.Collections.singleton(v.varName);
    }
    Set<String> visitConst(IExpConst c) {
      return java.util.Collections.emptySet();
    }
    Set<String> visitPlus(IExpPlus p) {
      Set<String> s = new HashSet<String>();
      s.addAll(p.e1.accept(this));
      s.addAll(p.e2.accept(this));
      return s;
    }
}
```

Verhaltensmuster – Visitor



Verhaltensmuster – Visitor

Evaluation

- Es ist leicht, neue Operation zu definieren, ohne die Klassen in der Hierarchie ändern zu müssen.
- Die vielen Fälle einer einzigen Operation werden in einem Visitor zusammengefasst. (Vergleiche: In der Klassenhierarchie waren die verschiedenen Fälle verschiedener Operationen zusammengefasst)
- Hinzufügen einer neuen Klasse zur Hierarchie ist schwierig, da man auch alle Visitor-Klassen anpassen muss.
- Zur Definition der einzelnen Funktionen in einem Visitor-Objekt braucht man Zugriff auf den Zustand der Objekte in der Objekthierarchie.
  $\Rightarrow$ Kapselung nicht immer möglich.

Erzeugungsmuster – Abstract Factory

Konstruktoren sind nicht immer die beste Möglichkeit Objekte zu erzeugen.

Beispiel: Abstraktion von Systemtreibern

- Portabilität durch Abstraktion von Systemeigenschaften (Beispiel: Austausch von Datenbanksystemen)
- Programm muss auf Treiber zurückgreifen und Treiberobjekte erzeugen
- je nach System müssen verschiedene Treiber erzeugt werden (Beispiel: SQL Server oder PostgreSQL)

Erzeugungsmuster – Abstract Factory

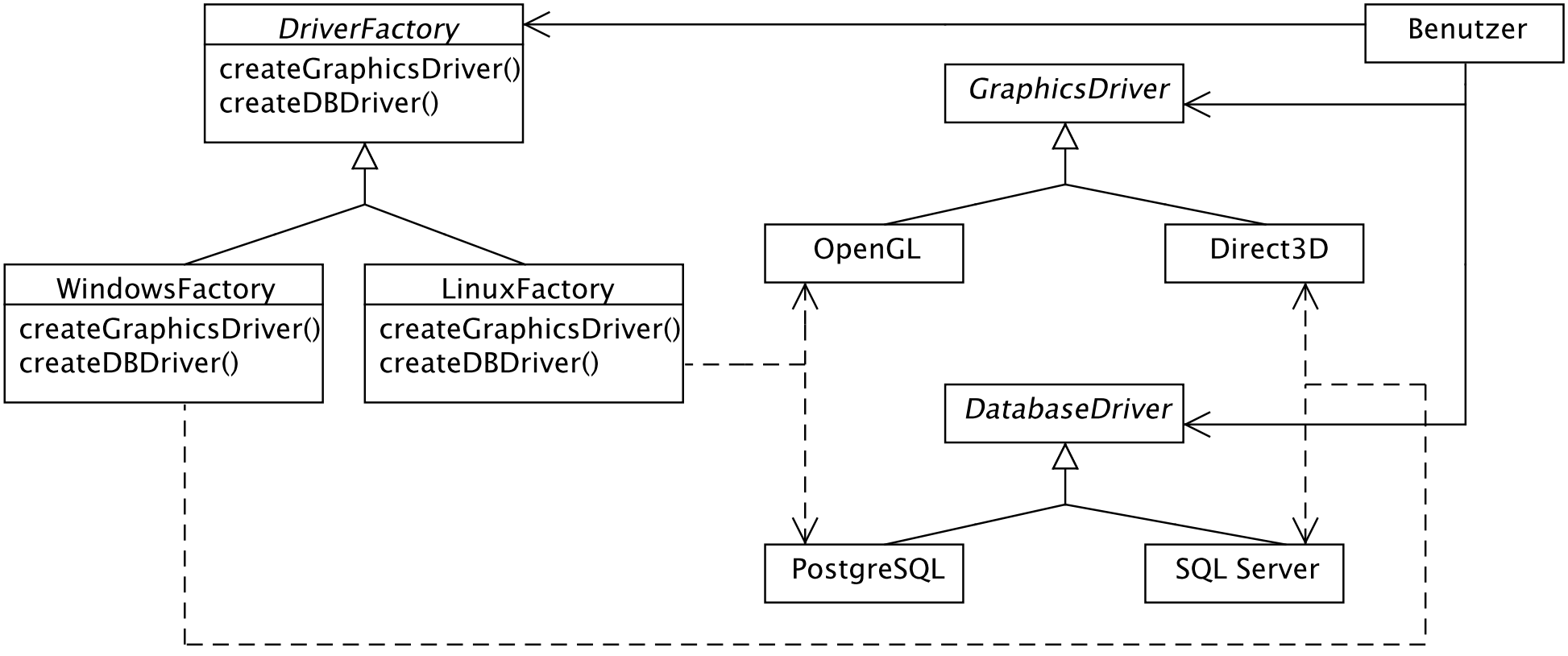
Abstract Factory

Schnittstelle zur Erzeugung von Familien zusammenhängender oder verwandter Objekte (z.B. Treiber), ohne dass dazu die konkreten Objektklassen bekannt sein müssen.

Idee:
„Erzeuge einen Datenbanktreiber.“
statt
„Erzeuge einen Treiber für PostgreSQL.“

Erzeugungsmuster – Abstract Factory

Struktur:



Erzeugungsmuster – Abstract Factory

`createGraphicsDriver()` und `createDBDriver()` sind Beispiele für *Factory-Methoden*

Factory-Methoden können in bestimmten Situation besser geeignet sein als Konstruktoren:

- Wiederverwendung von Objekten

  ```
  public static Boolean valueOf(boolean b);
  ```

- Erzeugung von geeigneten Unterklassen. Beispiel:

  ```
  public static <T> Set<T> synchronizedSet(Set<T> s);
  ```

  in `java.util.Collections` macht Menge thread-sicher
  - kann in zukünftigen Java-Versionen effizientere Implementierungen zurückliefern, ohne dass Benutzer ihren Code ändern müssten
  - Klasse, welche die Synchronisierung konkret implementiert, kann in `java.util` privat bleiben

Optimierungen

Notwendigkeit

häufige Situation: Programm funktioniert im Prinzip fehlerfrei, aber nicht mit gewünschter Performanz

Symptome des Mangels an Performanz:

- `OutOfMemoryError` wird geworfen
- kein flüssiger Programmablauf
- keine Termination bei größeren Eingaben
- ...

Bedarf an Optimierung besteht aber im Grunde immer (z.B. Testumgebung ungleich der Umgebung, in der Software eingesetzt wird)

Responsability

nicht nur wichtig, dass ein Programm i.A. schnell ist

bei Interaktion mit Benutzer auch psychologischer Effekt nicht zu unterschätzen

Programm, dass während langer Berechnung *irgendetwas* sichtbar macht, wird normalerweise als besser wahrgenommen

Bsp.: Fortschritt anzeigen, bewegende Bilder, ...

ebenfalls wichtig aus diesem Grund: Programm soll auf Eingaben *sofort* reagieren

Bsp.: längere Suche in einer Datenbank

Responsability kann z.B. erreicht werden durch Trennung von Threads für Interaktion und für rechenintensive Teile

Vorsicht

lauffähige Software zu optimieren ist prinzipiell sinnvoll, muss aber mit Vorsicht durchgeführt werden

Optimierungen bzgl. Laufzeitverhalten können auch nachteilige Effekte haben, gehen z.B. zu Lasten der Lesbarkeit des Codes / der Korrektheit / etc.

$\rightsquigarrow$ keine Optimierungen ohne Grund und erkennbaren Effekt durchziehen!

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.* – William Wulf

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.* – Donald Knuth

Optimieren und Verschlechtern

eine Optimierung kann mehrere Aspekte des Programms betreffen, z.B.

- Laufzeitverhalten
  - Geschwindigkeit
  - Platzverbrauch
- Sourcecode
  - Les- und Wartbarkeit
- Korrektheit
- Dokumentation
- Machbarkeit
- ...

Änderung ist oft Optimierung nur bzgl. eines Aspektes, aber Verschlechterung bzgl. anderer

Beispiele

- Zeit vs. Platz vs. Machbarkeit

  Repräsentation einer *Menge* als

  1. `boolean[]` mit Zugriff in $\mathcal{O}(1)$, Platz $\mathcal{O}(n)$
  2. Suchbaum mit Zugriff in $\mathcal{O}(m)$, Platz $\mathcal{O}(m)$

  (m = Anzahl eingefügter Elemente, n = Anzahl möglicher Elemente)

  beachte: $n \geq m$ und $n = \infty, m < \infty$ sogar möglich oder Grundmenge unbekannt

- Platz vs. Lesbarkeit

  Array der Größe n mit intuitiven Indizes $1, \ldots, n$ der Elemente

- Geschwindigkeit vs. Lesbarkeit

  vorzeitiger Abbruch einer Schleife; `while` statt `for`

Optimierungen durchführen

vor Änderung am Sourcecode muss abgewägt werden:

- ist Änderung wirklich eine Optimierung im Ganzen?
  - welche Aspekte werden optimiert?
  - welche anderen sind betroffen?
- Aufwand
  - wieviel Arbeit erfordert die Optimierung?
  - um wieviel wird das Programm dadurch verbessert?
- Design
  - steht geänderter Code z.B. im Widerspruch zu üblichem objekt-orientiertem Design?

danach/dabei muss z.B.

- Dokumentation angepasst werden
- Korrektheit neu überprüft werden

Optimierungspotential finden

bei nicht-offensichtlichem Mangel an Performanz:

- sinnvolle Wahl von Datenstrukturen

  welche Operationen werden häufig in dem Programm ausgeführt und welche Datenstrukturen sind bekanntermaßen am besten dafür?

- Einsatz von guten Algorithmen

  welches abstrakte Problem steckt hinter der Anwendung und welcher (bekannte) Algorithmus ist dafür der beste?

  Bsp.: Erreichbarkeit aller freien Felder auf Spielbrett kann als Nicht-Erreichbarkeit in ungerichteten Graphen aufgefasst werden $\rightsquigarrow$ Breitensuche

- Profiling

- Vermeidung von Java-Operationen, die als teuer bekannt sind

Profiling

Tabellieren des Ressourcenverbrauchs einzelner Programmteile bei einem Programmdurchlauf (vgl. Problematik beim Testen)

verschiedene Tools zum Profiling von Java-Programmen erhältlich; Eclipse und NetBeans z.B. haben auch integrierten Profiler; hier keine Vorstellung eines bestimmten

typische nützliche Information, die der Profiler zurückgibt:

- Häufigkeit und Dauer einzelner Methodenaufrufe (beachte Verschachtelungen!)
- Anzahl und Größe angelegter Objekte einer Klasse
- Anzahl von Objekten, die vom Garbage Collector eingesammelt wurden

anhand dessen Entscheidungen bzgl. Optimieren oder auch Nicht-Optimieren treffen

Teure Operationen in Java

- Anlegen von Objekten
- Exceptions werfen
- viele Klassen benutzen
- geboxte Datentypen (`Boolean`, `Integer`, etc.)
- manche Operationen auf Strings, insbesondere Konkatenation
- garbage collection
- type cast
- synchronisierte Methoden
- ...

Tips zur Programmbeschleunigung

- besonderes Augenmerk auf Schleifen

```
public static int[] countArr = new int[1];
private static Vector collection = new Vector(10000);
private static int points = 3;

// 1. Versuch
for(long i=0; i<collection.size()*50000; i++) {
  countArr[0]=countArr[0] + 5 * points;
}

// 2. Versuch
int count=countArr[0];
int addpoints=5*points;
int i=collection.size()*50000;
for(; --i>=0; ){
  count+=addpoints;
}
countArr[0]=count;
```

Performanzgewinn: 53894ms $\rightsquigarrow$ 846ms

Tips zur Programmbeschleunigung

- Datentyp `int` erlaubt schnelleren Zugriff als `long`, `byte`, `short` und als Objekte sowieso
- lokale Variablen verwenden
- Schleifen nicht durch Exceptions terminieren
- Iterationen statt Rekursionen
- Objektpools statt Neuerzeugung (siehe auch Factory Pattern)
- ...

Hinweise

- Geschwindigkeit moderner Rechner nicht unterschätzen!

  Platzoptimierung häufig wichtiger als Zeitoptimierung

- moderne Compiler nicht unterschätzen!

  viele Optimierungen (im Kontrollfluss) werden von Compilern automatisch durchgeführt

- Änderungen im Rahmen von Optimierungen können Korrektheitsannahmen wie z.B. Invarianten verletzen; Überprüfung / Umformulierung / neue Tests nötig

- `http://www.javaperformancetuning.com/`

Garbage Collection

Was ist Garbage Collection?

Wieso bricht folgendes nicht mit `OutOfMemoryError` ab?

```
Integer i;
Random r = new Random();
while (true) {
  i = new Integer(r.nextInt());
}
```

Objekte belegen Speicherplatz im Heap

JVM verwaltet Heap im Hintergrund

- neu erzeugte Objekte werden in freiem Speicher angelegt
- *garbage collection*: Speicherplatz nicht mehr referenzierter Objekte wird wieder freigegeben

  beachte: unreferenziertes Objekt kann (eigentich) nicht mehr referenziert werden; in Java kein Zugriff auf Heap möglich

Vor- und Nachteile

in einigen Sprachen wird Heapverwaltung dem Programmierer überlassen (z.B. C, C++)

Garbage Collection vermeidet zwei häufige Programmierfehler:

- Objekt wird freigegeben und danach verwendet
- Objekt wird *nicht* freigegeben und *nicht mehr* verwendet

Nachteile:

- keine Kontrolle über den Zeitpunkt, zu dem Objekte freigegeben werden; Garbage Collection findert zur Laufzeit statt
- wie findet man unreferenzierte Objekte?

  Garbage Collector muss alle verwendeten Objekte explizit in Tabellen halten

Reference Counting

eine Möglichkeit zum Erkennen unreferenzierter Objekte

einfachstes Verfahren, in modernen Compilern nicht mehr benutzt

Objekte werden annotiert mit Anzahl der Verweisen auf sie; Buchführung bei

- Objekterzeugung
- Zuweisung
- Parameterübergabe (Achtung! Java benutzt call-by-value, aber Objekte werden generell über Referenzen angesprochen)
- Beendigung einer Methode
- Freigabe eines Objekts

Vorteil: direkte Freigabe möglich; Nachteile: Overhead in Platz und Zeit, Probleme bei zyklischen Datenstrukturen

Mark-and-Sweep

zwei Phasen

1. Breitensuche von Programmvariablen (Wurzelzeigern) aus markiert alle noch referenzierten Objekte
2. alle unmarkierten Speicherbereiche werden freigegeben

Vorteile:

- zyklische Strukturen unproblematisch
- kein Zeit-Overhead bei Zuweisungen, etc.
- sehr geringer Platz-Overhead

Nachteil:

- Programmlauf muss unterbrochen werden
- Laufzeit hängt von Größe des Heaps ab
- Breitensuche selbst braucht Speicherplatz! ($\rightsquigarrow$ Deutsch-Schorr-Waite-Algorithmus)

Problem: Fragmentierung

nach gewisser Zeit können belegte und freie Speicherbereiche weit verstreut sein

im Extremfall wäre noch genügend freier Speicher vorhanden, Objekt kann aber nicht angelegt werden, da kein genügend großer freier Speicherbereich zur Verfügung steht

Abhilfe: *mark-and-compact*, zu mark-and-sweep zusätzlich dritter Durchlauf, in dem belegte Speicherbereiche verschoben werden

offensichtliche Vor- und Nachteile, zusätzlich:

- entweder Rückwärtsreferenzen oder intelligente Berechnung der neuen Speicherstellen nötig
- evtl. keine einfache Verschiebung von Objekten möglich

Stop-and-Copy

unterteile Heap in zwei Hälften, eine davon bleibt immer frei

ist andere Hälfte voll, dann kopiere referenzierte Objekte in freie Hälfte; vertausche danach Funktion der beiden Hälften

Vorteile:

- Kompaktierung geschieht automatisch
- nur ein Durchlauf statt 3 nötig

Nachteile:

- nur halber Heap-Space steht zur Verfügung
- Objektkopieren ist aufwändig; langlebige Objekte werden hin und her verschoben

Generationen

zur Vermeidung von Kopieren langlebiger Objekte werden *Generationen* eingeführt

Bsp.: 2 Generationen *young* and *old*

- neue Objekte kommen in die Generation young
- haben Objekte eine gewisse Zahl an Kopiervorgängen in der Generation young überlebt, dann werden sie in die Generation old verschoben
- führt zu 4-Teilung des Heaps
- Garbage Collection in jeder Generation separat

im Prinzip beliebig viele Generationen vorstellbar

Verbesserungen

verschiedene Verbesserungen möglich

zielen i.A. darauf ab, den Programmfluss durch GC möglichst wenig zu unterbrechen

- paralleles GC

  nutzt mehrere CPUs aus; (Achtung! nicht "parallel zum eigentlichen Programmablauf")

- inkrementelles GC

  vermeidet längeres Anhalten des Programms; GC wird z.B. nur auf kleinen Stücken des Heaps ausgeführt (z.B. *train* Algorithmus)

- nebenläufiges GC

  GC wird in eigenen Threads ohne Anhalten des Programms (außer durch den Scheduler) ausgeführt

GC in der J2SE 5.0 Hotspot JVM

- 3 Generationen: *young*, *old* und *permanent*
- Generation young unterteilt in zwei *survivor spaces*: *fromSpace* und *toSpace* für stop-and-copy-Verfahren
- Generationen *old* und *permanent* werden per mark-and-compact verwaltet
- serielles GC ist Standard, aber parallele und nebenläufige Verfahren können per Kommandozeile gewählt werden
- Garbage Collector hat Logging-Mechanismus
- Objekte haben Finalisierungsflag
  - noch nicht finalisierte Objekte werden von GC in Queue zum Finalisieren gesteckt
  - bereits finalisierte Objekte werden direkt entfernt

Die Methode `finalize`

analog zu Konstruktoren gibt es in Java auch Destruktoren:
Methode `protected void finalize()` in Klasse `Object`

wird vom Garbage Collector vor Freigabe des Objekts aufgerufen

gedacht z.B. zur Freigabe von Systemressourcen, die von Objekt gehalten werden

zu beachten:

- GC ruft `finalize` auf jedem Objekt höchstens einmal auf
- keine rekursiven Aufrufe auf Objekten in Instanzvariablen nötig
- Exceptions in `finalize` unterbrechen Methode, werden ansonsten aber ignoriert

`finalize` sollte nicht benutzt werden (Rereferenzierung möglich, `OutOfMemoryError` kann auftreten, ...)

Beispiel

```
public class GC {
    public static void main(String[] args) {
        while (true) {
            new Muell();
        }
    }
}

class Muell {
    protected void finalize() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {};
    }
}
```

führt ultimativ zu `OutOfMemoryError`, aber wieso?

Datenstrukturen in Java

Datenstrukturen

Datenstrukturen ermöglichen Verwaltung von / Zugriff auf Daten (hier: Objekte)

Datenstrukturen unterscheiden sich duch

- Funktionalität
- Implementierung

modulares Design: Datenstrukturen gleicher Funktionalität aber verschiedener Implementierung sollten untereinander austauschbar sein; evtl. Auswirkung auf Performanz, aber nicht auf syntaktische und semantische Korrektheit

Objektorientierung: Funktionalität durch Interface beschrieben

Beispiele

Beispiele für bereitgestellte Funktionalität:

- (endliche) Menge (Interface `Set<E>`)
- (endliche) Liste (Interface `List<E>`)
- (endliche) Abbildung (Interface `Map<K,V>`)
- ...

Beispiele für unterschiedliche Implementierung:

- Mengen: `HashSet`, `TreeSet`, ...
- Listen: `ArrayList`, `LinkedList`, `Vector`, ...
- ...

Vorsicht: Implementierung eines Interfaces kann auch zu Spezialisierung führen: `EnumSet<E extends Enum<E>>`

Endliche Mengen

endliche Mengen sind Kollektionen mit Eigenschaften

- alle Elemente sind verschieden (bzgl. üblicher Gleichheit oder abgeleitet von $<$)

Funktionalität

- Test auf Enthaltensein eines Objekts
- Hinzufügen (auch schon vorhandener Objekte)
- Löschen (auch nicht vorhandener Objekte)
- Vereinigung, Durchschnitt, ...
- (Iteration über alle Elemente der Menge)
- ...

Das Interface `Set<E>`

```
interface Set<E> {
  ...
  boolean contains(Object o);
  boolean add(E e);
  boolean remove(Object o);
  boolean addAll(Collection<? extends E> c);
  boolean retainAll(Collection<?> c);
  ...
}
```

zu beachten: Typ `Object` im Argument

- lässt statisch auch z.B. Anfragen mit Argumenten nicht vom Typ `E` zu
- vermutlich historische Gründe
- Vergleiche benutzen Methode `equals`

Listen

endliche Liste ist lineare Datenstruktur; jedes (bis auf letztes) Element hat genau einen Nachfolger; jedes (bis auf erstes) Element hat genau einen Vorgänger

Funktionalität

- Einfügen, Löschen, Ersetzen (an bestimmter/beliebiger Stelle)
- Iteration durch alle Elemente
- Suchen eines Elementes

nicht unbedingt direkten Zugriff auf bestimmte Stelle wie bei Array

Spezialisierungen:

- Stack: Einfügen, Löschen nur am Anfang
- Queue: Löschen am Anfang, Einfügen am Ende

Das Interface `List<E>`

```
interface List<E> {
  ...
  boolean contains(Object o);
  boolean add(E e);
  boolean add(int i, E e);
  boolean remove(Object o);
  boolean addAll(Collection<? extends E> c);
  boolean retainAll(Collection<?> c);
  ...
}
```

beachte:

- Argumenttyp `Object` wie bei `Set<E>`
- `addAll(Collection<E>)` nicht gut, da Typkontexte Untertypbeziehungen nicht erhalten

Abbildungen

Bsp.: Zuordnungen mit endlichem Domain

1. nein: jeder Arena die darauf angemeldeten Spieler zuordnen
2. ja: pro Arena jedem Spieler eine Farbe zuordnen

(1) leicht durch Instanzvariable in Arena zu lösen; typischerweise kein Suchen nach einer bestimmten Arena nötig

(2) Spieler fungiert als Schlüssel, Farbe als Wert; typischerweise sucht man nach Farbe für gegebenen Spieler

Abbildung kann man sich als zweispaltige Tabelle vorstellen; keine zwei Zeilen mit gleicher erster Spalte (Schlüssel) vorhanden

Abbildungen – Funktionalität

geg. Menge K der Schlüssel, Menge V der Werte

- Abfragen des Werts zu einem Schlüssel k
- Eintragen einer neuen Abhängigkeit $k \mapsto v$
- Löschen eines Schlüssels (und seines Werts)
- Abfrage nach Vorhandensein eines Schlüssels
- (Iteration über alle Schlüssel-Wert-Paare)
- ...

Das Interface `Map<K,V>`

```
interface Map<K,V> {
  ...
  V get(Object k);
  V put(K k, V v);
  V remove(Object k);
  boolean containsKey(Object k);
  ...
  Set<Map.Entry<K,V>> entrySet();
  Set<K> keySet();
  Collection<V> values();
  ...
}
```

beachte:

- vorhandene Schlüssel bilden Menge, Werte jedoch nicht

Zugriff auf Objekte

unabhängig von Art der Datenstruktur besteht in einer Implementierung das Problem des Zugriffs

einfachster Fall: finde gegebenes Objekt in Datenstruktur

im Prinzip zwei Möglichkeiten

- Durchsuchen der Datenstruktur von festem Ausgangspunkt aus; Vergleiche zwischen gegebenem und vorhandenen Objekten

  erfordert schnell zu durchlaufende Stuktur $\rightsquigarrow$ Suchbäume

- direkter Sprung an eine Stelle innerhalb der Struktur, die von gegebenem Objekt abhängt (Hashing)

  Unterschied zu Array-artigen Strukturen: diese verwalten eher die Stellen, an denen Objekt abgelegt sind anstatt Objekte selbst

Suchbäume

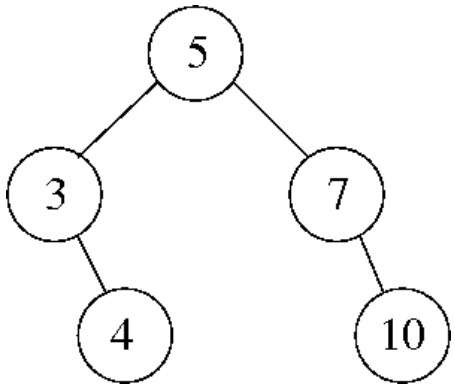
geg. Menge M mit totaler Ordung $<$

Def.: Suchbaum ist binärer M-beschrifteter Baum, sodass für alle Knoten v gilt:

- kommt u in linkem Teilbaum von v vor, so gilt $u < v$
- kommt u in rechtem Teilbaum von v vor, so gilt $u > v$

ab jetzt:

- $n = |M|$
- $m =$ Anzahl der Elemente im Baum
- $h =$ Höhe des Baums



Operationen auf Suchbäumen

- Suchen eines Objekts in $\mathcal{O}(h)$, worst-case $\mathcal{O}(m)$
- Einfügen ebenfalls
- Löschen ebenfalls
  - Blatt in $\mathcal{O}(1)$
  - innerer Knoten muss ersetzt werden durch größten in linkem oder kleinsten in rechtem Teilbaum

wichtig: Abfrage $u < v$ wird hier als $\mathcal{O}(1)$ angenommenen

in Java-Implementierung: Relation $<$ muss evtl. realisiert werden; je nach Typ der Menge M kann dies mehr oder weniger effizient sein

Balancierung

Zugriffe auf Suchbäume im worst-case nicht besser als bei Listen

besser jedoch auf balancierten Suchbäumen

Def.: Suchbaum ist balanciert, wenn sich in jedem Knoten die Höhen von linkem und rechten Teilbaum nur um Konstante unterscheiden

Operationen auf balancierten Suchbäumen:

- Suchen in $\mathcal{O}(\log m)$
- Einfügen und Löschen ebenfalls in $\mathcal{O}(\log m)$, erfordert evtl. Rebalancierung: Rechts-/Linksdrehung

Balancierte Suchbäume – Beispiele

- Red-Black Trees

  Knoten sind rot oder schwarz, gleiche Anzahl schwarzer Knoten auf jedem Pfad

  nicht balanciert laut obiger starker Definition (Höhen können sich um Faktor 2 unterscheiden)

- AVL Trees

  Knoten haben Balance-Wert: Höhenunterschied der beiden Teilbäume

Hauptunterschiede:

- Anzahl Balancierungen bei einer Operation: Red-Black $\mathcal{O}(1)$, AVL $\mathcal{O}(\log m)$
- AVL konzeptuell einfacher
- Red-Black oft in der Praxis besser

Alternativen

- Splay Trees

  Operation "splaying": bei Zugriff auf Knoten v wird dieser sukzessive zur Wurzel befördert

  Suchen, Einfügen, Löschen in amortisierter Zeit $\mathcal{O}(\log m)$; kein zusätzlicher Speicher nötig

  schneller Zugriff auf häufig verwendete Elemente;
  Anwendungen: Cache, Garbage Collector, ...

- Scapegoat Trees

  Suchen in worst-case $\mathcal{O}(\log m)$
  Einfügen / Löschen in amortisiert $\mathcal{O}(\log m)$
  ebenfalls kein zusätzlicher Speicher nötig

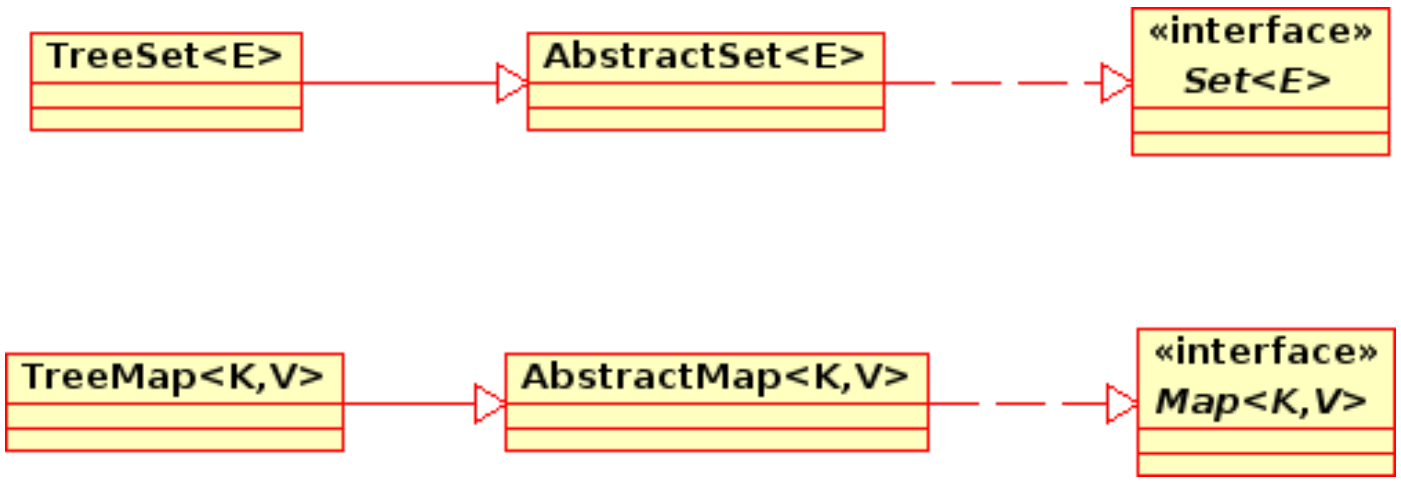
- ...

Suchbäume in Java

in Package `java.util` werden Red-Black Trees verwendet

- `TreeSet<E>` implementiert `Set<E>`
- `TreeMap<K,V>` implementiert `Map<K,V>`

genauer:



Die totale Ordnung in Suchbäumen

Interface für totale Ordnungen

```
interface Comparable<T> {
  int compareTo(T o);
}
```

Elemente von Klassen, die dieses Interface implementieren, können in Suchbäumen (also z.B. `TreeSet`, `TreeMap`) verwendet werden

Fragen:

1. wieso dann nicht gleich `TreeSet<E extends Comparable<E>>`?
2. was passiert bei `TreeSet<E>`, wenn `E` nicht `Comparable<E>` implementiert?

Comparable und Comparator

Antwort 2:

```
class TreeSet<E> {
  ...
  TreeSet() { ... }
  TreeSet(Comparator<? super E> c) { ... }
  ...
}
```

bei Mengenkonstruktion kann ein Objekt übergeben werden, welches Vergleiche auf `E` durchführt

statt `compareTo(E x)` in `E` wird `compare(T x, T y)` in `Comparator<T>` für Vergleiche benutzt

Antwort 1: `E` kann durch Implementation von `Comparable` auf eine Art total geordnet sein, für Verwaltung in Suchbäumen bietet sich aber evtl. andere totale Ordnung an

Konsistenz zwischen `compareTo` und `equals`

Def.: die durch `compareTo` bzw. `compare` definierte Ordnung auf Typ `E` ist konsistent zu `equals`, falls für alle Elemente `x`,`y` vom Typ `E` gilt:

> `x.compareTo(y)` bzw. `compare(x,y)` hat selben booleschen Wert wie `x.equals(y)`

beachte: `x.compareTo(null)` wirft `NullPointerException`, aber `x.equals(null)` ist `false`; `null` ist aber nicht Instanz einer Klasse

Konsistenz wichtig, denn

- Vertrag des Interfaces `Set<E>` bezieht sich auf `equals`
- Implementierung `TreeSet<E>` benutzt `compareTo` oder `compare` für Vergleiche

Hashing

zur Erinnerung

- Elemente einer Menge M werden in Tabelle abgelegt, genannt *Hashtabelle*
- Tabelleneintrag ist *Bucket*, kann mehrere Elemente halten
- Position eines Eintrags in der Tabelle ist gegeben durch Funktion $M \to \mathbb{N}$, genannt *Hashfunktion*
- *Kollision* = zwei Einträge an gleicher Position
- abzubildenende Elemente werden auch *Schlüssel* genannt

Ziel: Hashfunktion so wählen, dass Kollisionen minimiert werden und somit Zugriff in $\mathcal{O}(1)$ möglich ist

Bucket z.B. durch Liste realisiert; Zugriff linear in Listenlänge (also diese möglichst konstant im Vergleich zur Größe der Hashtabelle

Hashtabellen für Mengen und Abbildungen

Hashtabellen modellieren Mengen auf natürliche Art: Menge aller in der Tabelle enthaltenen Elemente; einzelne Buckets jeweils wieder als Mengen realisieren

Funktionalität für Mengen leicht zu realisieren

Bsp.: zur Suche von Element x, berechne Hashwert $h(x)$ von x, durchsuche Bucket an Stelle $h(x)$ nach Vorkommen von x

für Abbildungen vom Typ $f : K \to V$

- Hashfunktion h ist vom Typ $K \to \mathbb{N}$
- Bucket an Stelle $h(k)$ enthält Eintrag $(k, f(k))$

Hashfunktionen

verschiedene Methoden, jeweils mit unterschiedlicher Güte in verschiedenen Anwendungsfällen

- Extraktion: verwende lediglich Teil des Schlüssels
- Division: Abbildung auf Integer, dann modulo Tabellengröße
- Faltung: teile Schlüssel auf, verbinde Teile (z.B. Addition)
- Mitte-Quadrate: Abbildung auf Integer, Quadrierung, genügend großes Stück aus Mitte nehmen
- Radixtransformation: Abbildung auf Integer, Transformation in andere Basis, dann Division
- ...

hier keine detaillierte Diskussion der jeweiligen Güte

Dynamik einer Hashtabelle

Hashtabellen typischerweise parametrisiert durch

- initiale Kapazität: wieviele Tabelleneintrage zu Anfang
- (Schrittweite der Vergrößerungen: wieviele Einträge kommen hinzu, wenn Tabelle zu klein erscheint)
- Auslastungsfaktor: wieviele Elemente dürfen in der Tabelle höchstens vorhanden sein (in Relation zur Anzahl der Buckets), bevor diese vergrößert wird

gute Werte hängen natürlich von jeweiliger Anwendung ab

zu beachten: bei Tabellenvergrößerung werden alle eingetragenen Werte neu gehasht (z.B. per Divisionsmethode modulo neuer Tabellengröße)

- ganz andere Verteilung in der neuen Tabelle möglich
- Vergrößerungen aufwändig, also möglichst vermeiden; keine unnötigen Elemente in Tabelle, gute Parameterwahl, ...

Die Methode `hashCode` in Java

jedem Objekt hat standardmäßig einen Hashwert

```
class Object {
  ...
  int hashCode() { ... }
  ...
}
```

- Abbildung ist an sich sehr gute Hashfunktion
- Überschreiben bietet sich jedoch an (zwei an sich gleiche Objekte haben vermutlich verschiedene Hashwerte)
- noch bei Java 1.3: `hashCode` bildete lediglich Speicheradresse auf `int` ab; evtl. dynamische Änderung des Hashwerts durch Garbage Collection möglich

Verwendung von Hashtabellen

Methode `hashCode` liefert generischen Hashwert; Hashtabelle nutzt diesen, um Schlüssel auf Integer abzubilden; eigentlicher Hashwert (Position in geg. Tabelle) wird von Methoden in Hashtabellenimplementierung berechnet

wird `equals` überschrieben, so sollte man auch `hashCode` überschreiben

ebenfalls Konsistenzproblem:

- Vertrag von `Set<E>` bezieht sich auf `equals`
- Implementation `HashSet<E>` benutzt `hashCode`, um Bucket zu finden; nur innerhalb des Buckets werden Vergleiche gemacht

beim Überschreiben von `equals` zu beachten: nicht nur überladen! Argument ist `Object`

Überschreiben von `equals` und `hashCode`

folgendes muss bei `equals` beachtet werden

- Symmetrie: `x.equals(y)` hat immer selben Wert wie `y.equals(x)`
- Reflexivität: `x.equals(x)` ist immer `true` ausser wenn `x` = `null`
- Transitivität: wenn `x.equals(y)` und `y.equals(z)` beide `true` sind, dann auch `x.equals(z)`
- Konsistenz: wenn `x.equals(y)` den Wert `true` hat, dann gilt `x.hashCode()` = `y.hashCode()`
- Determiniertheit: die Werte von `equals` und `hashCode` hängen nur vom Zustand der involvierten Objekte ab

bei überschriebenem `hashCode` kann dennoch auf Standard-Hashwert zurückgegriffen werden: `System.identityHashCode(Object o)`

Die Hash-Klassen

- `HashSet<E>` implementiert Interface `Set<E>` auf Basis einer Hashtabelle
- zwei Implementierungen von `Map<K,V>` auf Basis von Hashtabellen:
  - `Hashtable<K,V>`
  - `HashMap<K,V>`

  Unterschiede: `Hashtable` ist synchronisiert, `HashMap` lässt auch `null` als Wert zu
- Varianten von `HashMap`:
  - `IdentityHashMap`: verwendet Referenzgleichheit (==) statt Objektgleichheit (`equals`)
  - `WeakHashMap`: erlaubt Freigabe von Objekten, die nur noch durch Hashtabelle referenziert werden
  - `LinkedHashSet` und `LinkedHashMap`: hält Objekte in Hashtabelle zusätzlich in doppelt verketteter Liste, um Einfügereihenfolge zu speichern

Generics in the Java Programming Language

Gilad Bracha

July 5, 2004

Contents

1 Introduction

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *generics*.

This tutorial is aimed at introducing you to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll soon see that there are both similarities and important differences. If you are not familiar with look-a-alike constructs from elsewhere, all the better; you can start afresh, without unlearning any misconceptions.

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.

Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer might be mistaken.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that List is a generic interface that takes a *type parameter* - in this case, Integer. We also specify a type parameter when creating the list object.

The other thing to pay attention to is that the cast is gone on line 3'.

Now, you might think that all we've accomplished is to move the clutter around. Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1'. However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntList is declared with type List<Integer>, this tells us something about the variable myIntList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

2 Defining Simple Generics

Here is a small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

This should all be familiar, except for the stuff in angle brackets. Those are the declarations of the *formal type parameters* of the interfaces List and Iterator.

Type parameters can be used throughout the generic declaration, pretty much where you would use ordinary types (though there are some important restrictions; see section 7).

In the introduction, we saw *invocations* of the generic type declaration List, such as List<Integer>. In the invocation (usually called a *parameterized type*), all occurrences of the formal type parameter (E in this case) are replaced by the *actual type argument* (in this case, Integer).

You might imagine that List<Integer> stands for a version of List where E has been uniformly replaced by Integer:

```
public interface IntegerList {
  void add(Integer x)
  Iterator<Integer> iterator();
}
```

This intuition can be helpful, but it's also misleading.

It is helpful, because the parameterized type List<Integer> does indeed have methods that look just like this expansion.

It is misleading, because the declaration of a generic is never actually expanded in this way. There aren't multiple copies of the code: not in source, not in binary, not on disk and not in memory. If you are a C++ programmer, you'll understand that this is very different than a C++ template.

A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration.

Type parameters are analogous to the ordinary parameters used in methods or constructors. Much like a method has *formal value parameters* that describe the kinds of values it operates on, a generic declaration has formal type parameters. When a method is invoked, *actual arguments* are substituted for the formal parameters, and the method body is evaluated. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.

A note on naming conventions. We recommend that you use pithy (single character if possible) yet evocative names for formal type parameters. It's best to avoid lower

case characters in those names, making it easy to distinguish formal type parameters from ordinary classes and interfaces. Many container types use E, for element, as in the examples above. We'll see some additional conventions in later examples.

3 Generics and Subtyping

Let's test our understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a List of String a List of Object. Most people's instinct is to answer: "sure!".

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

Here we've aliased ls and lo. Accessing ls, a list of String, through the alias lo, we can insert arbitrary objects into it. As a result ls does not hold just Strings anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error.

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

The problem with that intuition is that it assumes that collections don't change. Our instinct takes these things to be immutable.

For example, if the department of motor vehicles supplies a list of drivers to the census bureau, this seems reasonable. We think that a List<Driver> is a List<Person>, assuming that Driver is a subtype of Person. In fact, what is being passed is a **copy** of the registry of drivers. Otherwise, the census bureau could add new people who are not drivers into the list, corrupting the DMV's records.

In order to cope with this sort of situation, it's useful to consider more flexible generic types. The rules we've seen so far are quite restrictive.

4 Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language:

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
    System.out.println(i.next());
}}
```

And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
void printCollection(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is *not* a supertype of all kinds of collections!

So what *is* the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown") , that is, a collection whose element type matches anything. It's called a *wildcard type* for obvious reasons. We can write:

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

On the other hand, given a List<?>, we **can** call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is

therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

4.1 Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}
public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) { ... }
}
```

These classes can be drawn on a canvas:

```
public class Canvas {
  public void draw(Shape s) {
    s.draw(this);
  }
}
```

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {
  for (Shape s: shapes) {
    s.draw(this);
  }
}
```

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of *any* kind of shape:

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

There is a small but very important difference here: we have replaced the type List<Shape> with List<? extends Shape>. Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want.

List<? **extends** Shape> is an example of a *bounded wildcard.* The ? stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of Shape[1]. We say that Shape is the *upper bound* of the wildcard.

There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into shapes in the body of the method. For instance, this is not allowed:

```
public void addRectangle(List<? extends Shape> shapes) {
   shapes.add(0, new Rectangle()); // compile-time error!
}
```

You should be able to figure out why the code above is disallowed. The type of the second parameter to shapes.add() is ? **extends** Shape - an unknown subtype of Shape. Since we don't know what type it is, we don't know if it is a supertype of Rectangle; it might or might not be such a supertype, so it isn't safe to pass a Rectangle there.

Bounded wildcards are just what one needs to handle the example of the DMV passing its data to the census bureau. Our example assumes that the data is represented by mapping from names (represented as strings) to people (represented by reference types such as Person or its subtypes, such as Driver). Map<K,V> is an example of a generic type that takes two type arguments, representing the keys and values of the map.

Again, note the naming convention for formal type parameters - K for keys and V for values.

```
public class Census {
  public static void
     addRegistry(Map<String, ? extends Person> registry) { ...}
}
...
Map<String, Driver> allDrivers = ...;
Census.addRegistry(allDrivers);
```

5 Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection.

Here is a first attempt:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
  for (Object o : a) {
    c.add(o); // compile time error
}}
```

By now, you will have learned to avoid the beginner's mistake of trying to use Collection<Object> as the type of the collection parameter. You may or may not

[1] It could be Shape itself, or some subclass; it need not literally extend Shape.

have recognized that using Collection<?> isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use *generic methods.* Just like type declarations, method declarations can be generic - that is, parameterized by one or more type parameters.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) {
    c.add(o); // correct
}}
```

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferred to be Number
fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from the Collection libraries.

```
interface Collection<E> {
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c);
}
```

We could have used generic methods here instead:

```
interface Collection<E> {
  public <T> boolean containsAll(Collection<T> c);
  public <T extends E> boolean addAll(Collection<T> c);
  // hey, type variables can have bounds too!
}
```

However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy():

```
class Collections {
  public static  <T> void copy(List<T> dest, List<? extends T> src){...}
}
```

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T - we don't care which. The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

```
class Collections {
  public static  <T, S extends T>
        void copy(List<T> dest, List<S> src){...}
}
```

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src - nothing else depends on it. This is a sign that we can replace S with a wildcard. Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class Shape, and have drawAll() store its incoming argument into the history field.

```
static List<List<? extends Shape>> history =
new ArrayList<List<? extends Shape>>();
public void drawAll(List<? extends Shape> shapes) {
   history.addLast(shapes);
   for (Shape s: shapes) {
      s.draw(this);
   }}
```

Finally, again let's take note of the naming convention used for the type parameters. We use T for type, whenever there isn't anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor T in the alphabet, such as S. If a generic method appears inside a generic class, it's a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

6 Interoperating with Legacy Code

Until now, all our examples have assumed an idealized world, where everyone is using the latest version of the Java programming language, which supports generics.

Alas, in reality this isn't the case. Millions of lines of code have been written in earlier versions of the language, and they won't all be converted overnight.

Later, in section 10, we will tackle the problem of converting your old code to use generics. In this section, we'll focus on a simpler problem: how can legacy code and generic code interoperate? This question has two parts: using legacy code from within generic code, and using generic code within legacy code.

6.1 Using Legacy Code in Generic Code

How can you use old code, while still enjoying the benefits of generics in your own code?

As an example, assume you want to use the package com.Fooblibar.widgets. The folks at Fooblibar.com [2] market a system for inventory control, highlights of which are shown below:

```
package com.Fooblibar.widgets;
public interface Part { ...}
public class Inventory {
/**
 * Adds a new Assembly to the inventory database.
 * The assembly is given the name name, and consists of a set
 * parts specified by parts. All elements of the collection parts
 * must support the Part interface.
**/
  public static void addAssembly(String name, Collection parts) {...}
  public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
  Collection getParts(); // Returns a collection of Parts
}
```

Now, you'd like to add new code that uses the API above. It would be nice to ensure that you always called addAssembly() with the proper arguments - that is, that

[2] Fooblibar.com is a purely fictional company, used for illustration purposes. Any relation to any real company or institution, or any persons living or dead, is purely coincidental.

the collection you pass in is indeed a Collection of Part. Of course, generics are tailor made for this:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
...
}
public class Guillotine implements Part {
}
public class Main {
  public static void main(String[] args) {
    Collection<Part> c = new ArrayList<Part>();
    c.add(new Guillotine()) ;
    c.add(new Blade());
    Inventory.addAssembly("thingee", c);
    Collection<Part> k = Inventory.getAssembly("thingee").getParts();
}}
```

When we call addAssembly, it expects the second parameter to be of type Collection. The actual argument is of type Collection<Part>. This works, but why? After all, most collections don't contain Part objects, and so in general, the compiler has no way of knowing what kind of collection the type Collection refers to.

In proper generic code, Collection would always be accompanied by a type parameter. When a generic type like Collection is used without a type parameter, it's called a *raw type.*

Most people's first instinct is that Collection really means Collection<Object>. However, as we saw earlier, it isn't safe to pass a Collection<Part> in a place where a Collection<Object> is required. It's more accurate to say that the type Collection denotes a collection of some unknown type, just like Collection<?>.

But wait, that can't be right either! Consider the call to getParts(), which returns a Collection. This is then assigned to k, which is a Collection<Part>. If the result of the call is a Collection<?>, the assignment would be an error.

In reality, the assignment is legal, but it generates an *unchecked warning*. The warning is needed, because the fact is that the compiler can't guarantee its correctness. We have no way of checking the legacy code in getAssembly() to ensure that indeed the collection being returned is a collection of Parts. The type used in the code is Collection, and one could legally insert all kinds of objects into such a collection.

So, shouldn't this be an error? Theoretically speaking, yes; but practically speaking, if generic code is going to call legacy code, this has to be allowed. It's up to you, the programmer, to satisfy yourself that in this case, the assignment is safe because the contract of getAssembly() says it returns a collection of Parts, even though the type signature doesn't show this.

So raw types are very much like wildcard types, but they are not typechecked as stringently. This is a deliberate design decision, to allow generics to interoperate with pre-existing legacy code.

Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic

type system usually provides are void. However, you are still better off than you were without using generics at all. At least you know the code on your end is consistent.

At the moment there's a lot more non-generic code out there then there is generic code, and there will inevitably be situations where they have to mix.

If you find that you must intermix legacy and generic code, pay close attention to the unchecked warnings. Think carefully how you can justify the safety of the code that gives rise to the warning.

What happens if you still made a mistake, and the code that caused a warning is indeed not type safe? Let's take a look at such a situation. In the process, we'll get some insight into the workings of the compiler.

6.2 Erasure and Translation

```
public String loophole(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // compile-time unchecked warning
    return ys.iterator().next();
  }
```

Here, we've aliased a list of strings and a plain old list. We insert an Integer into the list, and attempt to extract a String. This is clearly wrong. If we ignore the warning and try to execute this code, it will fail exactly at the point where we try to use the wrong type. At run time, this code behaves like:

```
  public String loophole(Integer x) {
    List ys = new LinkedList;
    List xs = ys;
    xs.add(x);
    return (String) ys.iterator().next(); // run time error
  }
```

When we extract an element from the list, and attempt to treat it as a string by casting it to String, we will get a ClassCastException. The exact same thing happens with the generic version of loophole().

The reason for this is, that generics are implemented by the Java compiler as a front-end conversion called *erasure*. You can (almost) think of it as a source-to-source translation, whereby the generic version of loophole() is converted to the non-generic version.

As a result, **the type safety and integrity of the Java virtual machine are never at risk, even in the presence of unchecked warnings**.

Basically, erasure gets rid of (or *erases*) all generic type information. All the type information betweeen angle brackets is thrown out, so, for example, a parameterized type like List<String> is converted into List. All remaining uses of type variables are replaced by the upper bound of the type variable (usually Object). And, whenever the resulting code isn't type-correct, a cast to the appropriate type is inserted, as in the last line of loophole.

The full details of erasure are beyond the scope of this tutorial, but the simple description we just gave isn't far from the truth. It's good to know a bit about this, especially if you want to do more sophisticated things like converting existing APIs to use generics (see section 10), or just want to understand why things are the way they are.

6.3 Using Generic Code in Legacy Code

Now let's consider the inverse case. Imagine that Fooblibar.com chose to convert their API to use generics, but that some of their clients haven't yet. So now the code looks like:

```
package com.Fooblibar.widgets;
public interface Part { ...}
public class Inventory {
/**
 * Adds a new Assembly to the inventory database.
 * The assembly is given the name name, and consists of a set
 * parts specified by parts. All elements of the collection parts
 * must support the Part interface.
**/
 public static void addAssembly(String name, Collection<Part> parts) {...}
 public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
 Collection<Part> getParts(); // Returns a collection of Parts
}
```

and the client code looks like:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
...
}
public class Guillotine implements Part {
}
public class Main {
 public static void main(String[] args) {
   Collection c = new ArrayList();
   c.add(new Guillotine()) ;
   c.add(new Blade());
   Inventory.addAssembly("thingee", c); // 1: unchecked warning
   Collection k = Inventory.getAssembly("thingee").getParts();
}}
```

The client code was written before generics were introduced, but it uses the package com.Fooblibar.widgets and the collection library, both of which are using generic types. All the uses of generic type declarations in the client code are raw types.

Line 1 generates an unchecked warning, because a raw Collection is being passed in where a Collection of Parts is expected, and the compiler cannot ensure that the raw Collection really is a Collection of Parts.

As an alternative, you can compile the client code using the source 1.4 flag, ensuring that no warnings are generated. However, in that case you won't be able to use any of the new language features introduced in JDK 1.5.

7 The Fine Print

7.1 A Generic Class is Shared by all its Invocations

What does the following code fragment print?

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

You might be tempted to say false, but you'd be wrong. It prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

Indeed, what makes a class generic is the fact that it has the same behavior for all of its possible type parameters; the same class can be viewed as having many different types.

As consequence, the static variables and methods of a class are also shared among all the instances. That is why it is illegal to refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable.

7.2 Casts and InstanceOf

Another implication of the fact that a generic class is shared among all its instances, is that it usually makes no sense to ask an instance if it is an instance of a particular invocation of a generic type:

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // illegal
```

similarly, a cast such as

```
Collection<String> cstr = (Collection<String>) cs; // unchecked warning
```

gives an unchecked warning, since this isn't something the run time system is going to check for you.

The same is true of type variables

```
<T> T badCast(T t, Object o) {return (T) o; // unchecked warning
    }
```

Type variables don't exist at run time. This means that they entail no performance overhead in either time nor space, which is nice. Unfortunately, it also means that you can't reliably use them in casts.

7.3 Arrays

The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type.You can declare array *types* whose element type is a type variable or a parameterized type, but not array *objects*.

This is annoying, to be sure. This restriction is necessary to avoid situations like:

```
List<String>[] lsa = new List<String>[10]; // not really allowed
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // unsound, but passes run time store check
String s = lsa[1].get(0); // run-time error - ClassCastException
```

If arrays of parameterized type were allowed, the example above would compile without any unchecked warnings, and yet fail at run-time. We've had type-safety as a primary design goal of generics. In particular, the language is designed to guarantee that **if your entire application has been compiled without unchecked warnings using javac -source 1.5, it is type safe**.

However, you can still use wildcard arrays. Here are two variations on the code above. The first forgoes the use of both array objects and array types whose element type is parameterized. As a result, we have to cast explicitly to get a String out of the array.

```
List<?>[] lsa = new List<?>[10]; // ok, array of unbounded wildcard type
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = (String) lsa[1].get(0); // run time error, but cast is explicit
```

IIn the next variation, we refrain from creating an array object whose element type is parameterized, but still use an array type with a parameterized element type. This is legal, but generates an unchecked warning. Indeed, the code is unsafe, and eventually an error occurs.

```
List<String>[] lsa = new List<?>[10]; // unchecked warning - this is unsafe!
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = lsa[1].get(0); // run time error, but we were warned
```

Similarly, attempting to create an array object whose element type is a type variable causes a compile-time error:

```
<T> T[] makeArray(T t) {
   return new T[100]; // error
}
```

Since type variables don't exist at run time, there is no way to determine what the actual array type would be.

The way to work around these kinds of limitations is to use class literals as run time type tokens, as described in section 8.

8 Class Literals as Run-time Type Tokens

One of the changes in JDK 1.5 is that the class java.lang.Class is generic. It's an interesting example of using genericity for something other than a container class.

Now that Class has a type parameter T, you might well ask, what does T stand for? It stands for the type that the Class object is representing.

For example, the type of String.class is Class<String>, and the type of Serializable.class is Class<Serializable>. This can be used to improve the type safety of your reflection code.

In particular, since the newInstance() method in Class now returns a T, you can get more precise types when creating objects reflectively.

For example, suppose you need to write a utility method that performs a database query, given as a string of SQL, and returns a collection of objects in the database that match that query.

One way is to pass in a factory object explicitly, writing code like:

```
interface Factory<T> { T make();}
public <T> Collection<T> select(Factory<T> factory, String statement) {
      Collection<T> result = new ArrayList<T>();
      /* run sql query using jdbc */
      for (/* iterate over jdbc results */ ) {
         T item = factory.make();
         /* use reflection and set all of item's fields from sql results */
         result.add(item);
      }
      return result;
}
```

You can call this either as

```
select(new Factory<EmpInfo>(){ public EmpInfo make() {
                                    return new EmpInfo();
                                }}
     , "selection string");
```

or you can declare a class EmpInfoFactory to support the Factory interface

```
class EmpInfoFactory implements Factory<EmpInfo> {
    ...
    public EmpInfo make() { return new EmpInfo();}
}
```

and call it

```
select(getMyEmpInfoFactory(), "selection string");
```

The downside of this solution is that it requires either:

- the use of verbose anonymous factory classes at the call site, or
- declaring a factory class for every type used and passing a factory instance at the call site, which is somewhat unnatural.

It is very natural to use the class literal as a factory object, which can then be used by reflection. Today (without generics) the code might be written:

```
Collection emps = sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static Collection select(Class c, String sqlStatement) {
    Collection result = new ArrayList();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        Object item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
    }
    return result;
}
```

However, this would not give us a collection of the precise type we desire. Now that Class is generic, we can instead write

```
Collection<EmpInfo> emps =
                sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static <T> Collection<T> select(Class<T>c, String sqlStatement) {
    Collection<T> result = new ArrayList<T>();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        T item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
        }
    return result;
}
```

giving us the precise type of collection in a type safe way.

This technique of using class literals as run time type tokens is a very useful trick to know. It's an idiom that's used extensively in the new APIs for manipulating annotations, for example.

9 More Fun with Wildcards

In this section, we'll consider some of the more advanced uses of wildcards. We've seen several examples where bounded wildcards were useful when reading from a data structure. Now consider the inverse, a write-only data structure.

The interface Sink is a simple example of this sort.

```
interface Sink<T> {
  flush(T t);
}
```

We can imagine using it as demonstrated by the code below. The method writeAll() is designed to flush all elements of the collection coll to the sink snk, and return the last element flushed.

```
public static <T> T writeAll(Collection<T> coll, Sink<T> snk){
      T last;
      for (T t : coll) {
          last = t;
          snk.flush(last);
       }
       return last;
}
...
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // illegal call
```

As written, the call to writeAll() is illegal, as no valid type argument can be inferred; neither String nor Object are appropriate types for T, because the Collection element and the Sink element must be of the same type.

We can fix this by modifying the signature of writeAll() as shown below, using a wildcard.

```
public static <T> T writeAll(Collection<? extends T>, Sink<T>){...}
...
String str = writeAll(cs, s); // call ok, but wrong return type
```

The call is now legal, but the assignment is erroneous, since the return type inferred is Object because T matches the element type of s, which is Object.

The solution is to use a form of bounded wildcard we haven't seen yet: wildcards with a *lower bound*. The syntax ? **super** T denotes an unknown type that is a supertype of T[3]. It is the dual of the bounded wildcards we've been using, where we use ? **extends** T to denote an unknown type that is a subtype of T.

```
public static <T> T writeAll(Collection<T> coll, Sink<? super T> snk){...}
...
String str = writeAll(cs, s); // Yes!
```

Using this syntax, the call is legal, and the inferred type is String, as desired.

[3]Or T itself. Remember, the supertype relation is reflexive.

Now let's turn to a more realistic example. A java.util.TreeSet<E> represents a tree of elements of type E that are ordered. One way to construct a TreeSet is to pass a Comparator object to the constructor. That comparator will be used to sort the elements of the TreeSet according to a desired ordering.

```
TreeSet(Comparator<E> c)
```

The Comparator interface is essentially:

```
interface Comparator<T> {
  int compare(T fst, T snd);
}
```

Suppose we want to create a TreeSet<String> and pass in a suitable comparator, We need to pass it a Comparator that can compare Strings. This can be done by a Comparator<String>, but a Comparator<Object> will do just as well. However, we won't be able to invoke the constructor given above on a Comparator<Object>. We can use a lower bounded wildcard to get the flexibility we want:

```
TreeSet(Comparator<? super E> c)
```

This allows any applicable comparator to be used.

As a final example of using lower bounded wildcards, lets look at the method Collections.max(), which returns the maximal element in a collection passed to it as an argument.

Now, in order for max() to work, all elements of the collection being passed in must implement Comparable. Furthermore, they must all be comparable to *each other.*

A first attempt at generifying this method signature yields

```
public static <T extends Comparable<T>>
      T max(Collection<T> coll)
```

That is, the method takes a collection of some type T that is comparable to itself, and returns an element of that type. This turns out to be too restrictive.

To see why, consider a type that is comparable to arbitrary objects

```
class Foo implements Comparable<Object> {...}
...
Collection<Foo> cf = ...;
Collections.max(cf); // should work
```

Every element of cf is comparable to every other element in cf, since every such element is a Foo, which is comparable to any object, and in particular to another Foo. However, using the signature above, we find that the call is rejected. The inferred type must be Foo, but Foo does not implement Comparable<Foo>.

It isn't necessary that T be comparable to **exactly** itself. All that's required is that T be comparable to one of its supertypes. This give us: [4]

[4]The actual signature of Collections.max() is more involved. We return to it in section 10

```
public static <T extends Comparable<? super T>>
    T max(Collection<T> coll)
```

This reasoning applies to almost any usage of Comparable that is intended to work for arbitrary types: You always want to use Comparable<? **super** T>.

In general, if you have an API that only uses a type parameter T as an argument, its uses should take advantage of lower bounded wildcards (? **super** T). Conversely, if the API only returns T, you'll give your clients more flexibility by using upper bounded wildcards (? **extends** T).

9.1 Wildcard Capture

It should be pretty clear by now that given

```
Set<?> unknownSet = new HashSet<String>();
...
/** Add an element t to a Set s */
public static <T> void addToSet(Set<T> s, T t) {...}
```

The call below is illegal.

```
addToSet(unknownSet, "abc"); // illegal
```

It makes no difference that the actual set being passed is a set of strings; what matters is that the expression being passed as an argument is a set of an unknown type, which cannot be guaranteed to be a set of strings, or of any type in particular.

Now, consider

```
class Collections {
...
    <T> public static Set<T> unmodifiableSet(Set<T> set) { ... }
}
...
Set<?> s = Collections.unmodifiableSet(unknownSet); // this works! Why?
```

It seems this should not be allowed; yet, looking at this specific call, it is perfectly safe to permit it. After all, unmodifiableSet() does work for any kind of Set, regardless of its element type.

Because this situation arises relatively frequently, there is a special rule that allows such code under very specific circumstances in which the code can be proven to be safe. This rule, known as *wildcard capture*, allows the compiler to infer the unknown type of a wildcard as a type argument to a generic method.

10 Converting Legacy Code to Use Generics

Earlier, we showed how new and legacy code can interoperate. Now, it's time to look at the harder problem of "generifying" old code.

If you decide to convert old code to use generics, you need to think carefully about how you modify the API.

You need to make certain that the generic API is not unduly restrictive; it must continue to support the original contract of the API. Consider again some examples from java.util.Collection. The pre-generic API looks like:

```
interface Collection {
  public boolean containsAll(Collection c);
  public boolean addAll(Collection c);
}
```

A naive attempt to generify it is:

```
interface Collection<E> {
  public boolean containsAll(Collection<E> c);
  public boolean addAll(Collection<E> c);
}
```

While this is certainly type safe, it doesn't live up to the API's original contract. The containsAll() method works with any kind of incoming collection. It will only succeed if the incoming collection really contains only instances of E, but:

- The static type of the incoming collection might differ, perhaps because the caller doesn't know the precise type of the collection being passed in, or perhaps because it is a Collection<S>,where S is a subtype of E.
- It's perfectly legitimate to call containsAll() with a collection of a different type. The routine should work, returning false.

In the case of addAll(), we should be able to add any collection that consists of instances of a subtype of E. We saw how to handle this situation correctly in section 5.

You also need to ensure that the revised API retains binary compatibility with old clients. This implies that the erasure of the API must be the same as the original, ungenerified API. In most cases, this falls out naturally, but there are some subtle cases. We'll examine one of the subtlest cases we've encountered, the method Collections.max(). As we saw in section 9, a plausible signature for max() is:

```
public static <T extends Comparable<? super T>>
      T max(Collection<T> coll)
```

This is fine, except that the erasure of this signature is

```
public static Comparable max(Collection coll)
```

which is different than the original signature of max():

```
public static Object max(Collection coll)
```

One could certainly have specified this signature for max(), but it was not done, and all the old binary class files that call Collections.max() depend on a signature that returns Object.

We can force the erasure to be different, by explicitly specifying a superclass in the bound for the formal type parameter T.

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<T> coll)
```

This is an example of giving *multiple bounds* for a type parameter, using the syntax T1& T2 ... & Tn. A type variable with multiple bounds is known to be a subtype of all of the types listed in the bound. When a multiple bound is used, the first type mentioned in the bound is used as the erasure of the type variable.

Finally, we should recall that max only reads from its input collection, and so is applicable to collections of any subtype of T.

This brings us to the actual signature used in the JDK:

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

It's very rare that anything so involved comes up in practice, but expert library designers should be prepared to think very carefully when converting existing APIs.

Another issue to watch out for is *covariant returns*, that is, refining the return type of a method in a subclass. You should not take advantage of this feature in an old API. To see why, let's look at an example.

Assume your original API was of the form

```
public class Foo {
  public Foo create(){...}// Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
  public Foo create(){...} // actually creates a Bar
}
```

Taking advantage of covariant returns, you modify it to:

```
public class Foo {
  public Foo create(){...} // Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
  public Bar create(){...} // actually creates a Bar
}
```

Now, assume a third party client of your code wrote

```
public class Baz extends Bar {
  public Foo create(){...} // actually creates a Baz
}
```

The Java virtual machine does not directly support overriding of methods with different return types. This feature is supported by the compiler. Consequently, unless

the class Baz is recompiled, it will not properly override the create() method of Bar. Furthermore, Baz will have to be modified, since the code will be rejected as written - the return type of create() in Baz is not a subtype of the return type of create() in Bar.

11 Acknowledgements

Erik Ernst, Christian Plesner Hansen, Jeff Norton, Mads Torgersen, Peter von der Ahé and Philip Wadler contributed material to this tutorial.

Thanks to David Biesack, Bruce Chapman, David Flanagan, Neal Gafter, Örjan Petersson, Scott Seligman, Yoshiki Shibata and Kresten Krab Thorup for valuable feedback on earlier versions of this tutorial. Apologies to anyone whom I've forgotten.