

Programmierparadigmen

Kapitel 2 Objektorientierung und weiterführende Konzepte

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



1



Ausgangssituation & Lernziele

- Bekannt:
 - Grundlegendes Verständnis von Java
 - Kapselung durch Klassen und Vererbung
- Ziele:
 - Verständnis der Probleme bei Vererbung und Typersetzbarkeit in objektorientierten Programmiersprachen
 - Kennenlernen der Grundideen generischer und abstrahierender Konzepte in Objekt-orientierter Programmierung (OOP)
 - Praktische Erfahrungen anhand von Java & C++
- Ausdrucksstärke erhöhen, Komplexität verbergen





TELEMATIK

- Leitfrage: Wie kann die Komplexität von Software beherrscht werden?
- Gegliedert in Unterpunkte:
 - · Wie können Änderungen an komplexen Code-Basen beherrscht werden?
 - Wie können Algorithmen generisch für verschiedene Datentypen implementiert werden und Code-Duplizierung vermieden werden?
 - · Wie können Datentypen sicher generalisiert werden?

3



Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte



Überblick

- Unit-Testing
- Reflections
- Assertions/Pre-/Postconditions/Invarianten
- Exceptions
- Generizät von Datentypen
 - · Grenzen von Typsubstitution
 - Ko- und Kontravarianz
 - Liskovsches Substitutionsprinzip
 - Generics in Java





Unit-Testing – Motivation

- Große Software-Systeme entwickeln sich über lange Zeiträume
- Wie können Änderungen an komplexen Code-Basen beherrscht werden?
- Veränderung über Zeit + Komplexität der Software
 - → Änderungen führen mglw. zu Auswirkungen, die für Einzelne nicht immer überschaubar sind
 - → Software muss von Grund auf durchgetestet werden
- Verbreitetes Vorgehen: zusätzlichen Code schreiben, der eigentlichen Code automatisch "überprüft"
 - Nicht vollständig möglich (siehe z.B. Halteproblem)
 - · Eher Heuristik
- Test-Code wird bei Ereignissen oder periodisch ausgeführt
 - Vor Releases, nach Commit in Repository, während der Entwicklung ...

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

5





Eigenschaften von Unit-Tests

- Software schlecht als Ganzes testbar → Zergliederung von Software in sinnvolle Einheiten
- Individuelle Tests dieser Einheiten
- Dabei: reproduzierbar & vollautomatisierbar
- → Ziel: Wann immer Änderungen in komplexen Programmen vorgenommen werden, möglichst vollständiger Test, da Programmierer nicht mehr alles überblicken
- Messung der Vollständigkeit der Tests schwierig
- Gängig: Messung von Überdeckung (Coverage) in Bezug auf Anzahl Funktionen, Code-Zeilen oder Verzweigungen
- Gute Praxis: Wenn ein Bug beim Testen oder Live-Betrieb auftritt → Schreiben eines zusätzlichen Tests um Wiederauftreten zu erkennen



Unit-Testing in Java



- De facto Standard: JUnit Framework
- "Best Practice" für einfachen Einsatz:
 - Java Code in ein oder mehrere Klassen im Ordner src speichern
 - Im Ordner tests jeweils eine Klasse anlegen, die Funktionen einer Implementierungsklasse prüft
 - Konvention Testklasse einer Klasse Name heißt NameTest
 - Eigentliche Tests werden in Methoden implementiert, die als Tests annotiert sind
 - Typischer Ansatz: für bekannte Werte ausführen und Ergebnis mit Grund wahrheit vergleichen, bspw. mit assertEquals-Funktion
- Viele weitere Features, z.B. Deaktivieren von Tests, Timeouts, GUI Coverage, Mocks
 - · Hier nicht weiter vertieft
 - Siehe https://junit.org/junit5/docs/current/user-guide/

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

7





JUnit - Einfaches Beispiel

Multiplikationsklasse

```
public class Multi {
    int mul(int a, int b) {
      return a * b;
    }
}
```

Korrespondierende Testklasse

```
import static org.junit.jupiter.api.Assertions.*;

class MultiTest {
    @org.junit.jupiter.api.Test
    void mul() {
        Multi m = new Multi();
        assertEquals(m.mul(1,2), 2, "should work");
        assertEquals(m.mul(2,0), 1, "should explode");
    }
}
```





Unit Testing – Richtiges Abstraktionsniveau

- Hauptschwierigkeiten von Unit-Tests:
 - Richtiges Abstraktionsniveau
 - · "Herauslösen" von zu testendem Code aus Umgebung
- Zwei wesentliche Möglichkeiten:
- Individuelles Testen von Klassen:
 - Vernachlässigt Zusammenspiel zwischen Klassen
 - Oft sehr aufwändig, da andere Klassen für Unit-Tests nachgebildet werden müssen (Mocks)
 - · Was bei zyklischen Abhängigkeiten?
- Gemeinsames Testen von Klassen:
 - Erfordert Eingreifen in gekapselte Funktionalitäten
 - Private & protected Member-Variablen & Methoden!
 - o Eigentlich nicht möglich?!

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

9





Unit Testing - Erreichbarer Code

Weiteres Problem: Testen von Fehlerbehandlung

```
class Bank {
   private void einzahlen(String p, double euro) {
        // Code der Kontostand auf der Festplatte speichert...
        // Schlägt alle 2 Jahre einmal fehl...
   }
   public void ueberweise(String p1, String p2, double euro) {
        einzahlen(p1, -euro);
        einzahlen(p2, euro);
   }
}
```

- Wie kann man testen, dass nie Geld verloren geht?
- Man müsste gezielt Rückgabewerte von Framework-Methoden oder privaten Methoden austauschen – aber wie?





Unit-Testing

TELEMATIK

- Reflections
- Assertions/Pre-/Postconditions/Invarianten
- Exceptions
- Generizät von Datentypen
 - Grenzen von Typsubstitution
 - Ko- und Kontravarianz
 - Liskovsches Substitutionsprinzip
 - · Generics in Java

11



Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte

MATIK

Reflections

- Normaler Ablauf: Programm schreiben, compilieren, ausführen
- Aber was wenn ich ein Programm zur Laufzeit inspizieren oder verändern möchte?
- Unterschiedliche Gründe
 - Testen (um Fehler zu injizieren!)
 - Nachladen von Plugins zur Modularisierung von Programmen
 - Serialisierung/Deserialisierung von Code
 - · Patchen zur Laufzeit
 - Debugging
- Benötigt die Fähigkeit im Programm Codestruktur zu analysieren und ggf. zu verändern
- Typisch: Abruf Klassenhierarchie, Auflisten von Methoden und Parametern, Austausch von Klassen und Methoden
- Teil von Java, Python, ...



Reflections – in Java



- API verstreut über verschiedene Packages, z.B. java.lang.Class, java.lang.instrument, java.lang.reflect
- Einfache Beispiele:

```
Class cls = "test".getClass();
System.out.println("Die Klasse heisst " + cls.getName());
// Die Klasse heisst java.lang.String
```

```
// import java.lang.reflect.Method;
Method[] methods = cls.getMethods();
for (Method m : methods)
    System.out.println(m.getName());
```

Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte

13



Reflections - in Java

Ein komplexes Beispiel:

```
static class Foo {
    private String h = "Hallo";
    public void greet() { System.out.println(h); }
}

public static void main(String[] args) {
    Foo foo = new Foo();
    foo.greet();
    try {
        Field f = foo.getClass().getDeclaredField("h");
        f.setAccessible(true);
        f.set(foo, "Moin");
    } catch (Exception e) {
    }
    foo.greet();
}
```



Reflections – Annotations



- Annotationen erlauben Anmerkungen an Klassen & Methoden
- Beginnen mit @
- Einige wenige vordefinierte z.B. @Override
- Aber auch eigene; u.A. durch Reflections abrufbar
- Häufig genutzt, wenn zusätzlicher Code geladen wird (Java EE!)

```
@WebServlet(urlPatterns = {"/account", "/password"})
public class MySlet extends
javax.servlet.http.HttpServlet { ...
```

Oder um Unit-Tests zu markieren...

```
class MultiTest {
   @org.junit.jupiter.api.Test
   void mul() {
     ...
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

15





Reflections – Zugriff auf Annotations





Reflektionen über Reflections

- Reflections sind ein sehr m\u00e4chtiges Werkzeug, aber Einsatz sollte wohldosiert erfolgen
- Nachteile:
 - Geringe Geschwindigkeit weil Zugriff über Programmcode erfolgt
 - Kapselung kann umgangen werden
 - o private, protected und final können entfernt werden
 - Aufruf/Veränderung interner Methoden & Auslesen/Veränderung interner Variablen
 - Synchronisation zwischen externen und internen Komponenten bei Weiterentwicklung?
 - Debugging veränderter Programme?
 - · Sicherheit?!
 - Verwandte Techniken:
 - Monkey Patching (JavaScript-Umfeld)
 - Method Swizzling (Swift/Objective-C-Umfeld)

17



Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte

Überblick

- Unit-Testing
- Reflections
- Assertions/Pre-/Postconditions/Invarianten
- Exceptions
- Generizät von Datentypen
 - Grenzen von Typsubstitution
 - Ko- und Kontravarianz
 - Liskovsches Substitutionsprinzip
 - Generics in Java



Assertions



- Kann man interne Zustände testen, ohne invasive Techniken wie Reflections?
- Einfache Möglichkeit: An sinnvollen Stellen im Programmcode testen, ob Annahmen/Zusicherungen (Assertions) stimmen...
- Tests, die nie falsch sein sollten
 - Erlauben gezielten Programmabbruch, um Folgefehler zu vermeiden
 - Erlauben gezieltes Beheben von Fehlern
 - Gemeinsames Entwickeln von Annahmen und Code

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

19





Assertions

- Aber: Ausführungsgeschwindigkeit niedriger
- Verluste stark abhängig von Programm/Programmiersprache
- Verbreitetes Vorgehen:
 - · Aktivieren der Tests in UnitTests und Debug-Versionen
 - · Deaktivieren in Releases
- → Benötigt spezielle "if"-Bedingung: assert

```
class Stack {
    public void push(Object o) {
        ...
        assert empty() == false
    }
    ...
}
```

Aktivierung der Tests über Start mit java -ea



Pre- & Postconditions



- An welchen Stellen ist es sinnvoll Annahmen zu prüfen?
- Einfache Antwort: an so vielen Stellen wie möglich
- Komplexere Antwort: Design by contract, ursprünglich Eiffel
- Methoden/Programmabschnitte testen Bedingung vor und nach Ausführung

```
class Stack {
    public void push(Object o) {
        assert o != null // precondition
        ...
        assert empty() == false // postcondition
    }
    ...
}
```

- Einige Sprachen bieten spezialisierte Befehle: requires und ensures
- → Ziel mancher Sprachen: Formale Aussagen über Korrektheit

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

21





Klasseninvarianten

- Bei OO-Programmierung sind Vor- und Nachbedingungen nur eingeschränkt sinnvoll
- Bedingungen oft besser auf Objekt-Ebene → interner Zustand
- Invarianten spezifizieren Prüfbedingungen
- In Java nicht nativ unterstützt:
 - · Erweiterungen, wie Java Modeling Language
 - Simulation:

```
class Stack {
    void isValid() {
        for(Object o : _objs)
            assert o != null
    }
    public void push(Object o) {
        isValid() // always call invariant
            ...
        isValid() // always call invariant
    }
}
```





- TELEMATIK Rechnernetze
 - Unit-Testing
 - Reflections
 - Assertions/Pre-/Postconditions/Invarianten
 - Exceptions
 - Generizät von Datentypen
 - Grenzen von Typsubstitution
 - Ko- und Kontravarianz
 - Liskovsches Substitutionsprinzip
 - Generics in Java

23



Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte



Exceptions

- Signifikantes Element vieler Sprachen: Wie wird mit Fehlern umgegangen?
- Fehler selbst unterschiedliche Gründe:
 - · Fehler im Programm
 - Ungültige Eingabeparameter
 - · Fehler im Programmablauf
 - Speicherallokation fehlgeschlagen
 - o E/A-Fehler
 - o Bit-Flips
- Erstere in komplexen Programmen nicht auszuschließen
- Letztere nie auszuschließen





Exceptions

- Problem: Fehler können an verschiedenen Stellen im Code auftreten
- → Führt schnell zu Spaghetti-Code, vergessenen Rückgabe-Checks & Fehlernummern (String-Rückgabe ineffizient)

```
class DB { // this is pseudo code!
    byte [] array;
    private int readFile(String f) {
        int errorCode = 0;
        File f = File.open("/tmp/file");
        if(f.canRead() == true) {
            int len = f.length();
            if(len > 0) {
                array = new char[len];
                if(array != null) {
                    bool ok = f.readAllBytes(array);
                    if(ok == false)
                        errorCode = -1;
                } else {
                    errorCode = -2;
                }
            } else {
                errorCode = -3;
            if(f.close() == false)
                errorCode = -4;
        } else {
            errorCode = -5;
        return errorCode;
    }
}
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte







Exceptions

- Besser für Code-Komplexität: Fehlerprüfungen an zentralerer Stelle
 - · Weiterlaufen lassen und später auf irgendeinen Fehler prüfen?
 - Besser: Abbrechen und Programm-Stack "abbauen" bis (zentralere) Fehlerbehandlung greift
 - Dabei Fehler sinnvoll gruppieren
- Java (und viele mehr): try/catch/throw-Konstrukt

```
private void readFile(String f) {
    try {
        Path file = Paths.get("/tmp/file");
        if(Files.exists(file) == false)
            throw new IOException("No such dir");
        array = Files.readAllBytes(file);
    } catch(IOException e) {
        // do something about it
    }
}
```

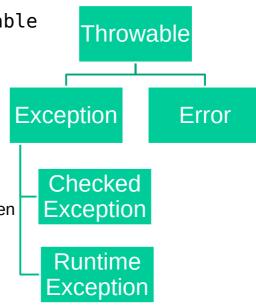


Exceptions - in Java



throw übergibt ein Objekt vom Typ Throwable an Handler, dabei zwei Unterarten:

- Error:
 - Sollte nicht abgefangen werden
 - z.B. Fehler im Byte-Code, Fehlgeschlagene Assertions
- Exceptions:
 - Checked Exception: Programm muss Exception fangen oder in Methode vermerken
 - Runtime Exceptions: Müssen nicht (aber sollten) explizit behandelt werden, bspw. ArithmeticException oder IndexOutOfBoundsException



Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte







Exceptions – Checked Exceptions

Deklaration einer überprüften Exception:

```
void dangerousFunction() throws IOException {
    ...
    if(onFire)
        throw IOException("Already burns");
    ...
}
```

Aufrufe ohne try-catch-Block schlagen fehl!

- Sollte man checked oder unchecked Exceptions verwenden?
 - · Checked sind potenziell sicherer
 - Unchecked machen Methoden lesbarer
 - → Faustregel unchecked, wenn immer auftreten können (zu wenig Speicher, Division durch 0)





Exceptions - Syntaktischer Zucker (I)

Fangen mehrerer unterschiedlicher Exceptions:

```
try {
  dangerousFunction();
} catch(IOException i) {
  // handle that nasty error
} catch(Exception e) {
  // handle all other exceptions
}
```

Was passiert wenn beide catch-Blöcke vertauscht werden?

Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte



29



Exceptions – Syntaktischer Zucker (II)

Aufräumen nach einem try-catch-Block:

```
try {
  dangerousFunction();
} catch(Exception e) {
  // handle exceptions
  return;
} finally {
  // release locks etc..
}
```

Anweisungen im finally-Block werden immer ausgeführt, d.h. auch bei return in try- oder catch-Block (oder fehlerloser Ausführung)

- Trick: Man braucht keinen catch-Block!
 - Try-finally-Blöcke eignen sich Locks sicher freizugeben etc.
 - Mehr zu ähnlichen Konzepten später...



Überblick



- Unit-Testing
- Reflections
- Assertions/Pre-/Postconditions/Invarianten
- Exceptions
- Generizät von Datentypen
 - Grenzen von Typsubstitution
 - Ko- und Kontravarianz
 - Liskovsches Substitutionsprinzip
 - · Generics in Java

31



Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



Generizität – Motivation

- (Typ-)Generizität:
 - Anwendung einer Implementierung auf verschiedene Datentypen
 - Parametrisierung eines Software-Elementes (Methode, Datenstruktur, Klasse, ...) durch einen oder mehrere Typen
- Beispiel: Minimum zweier Werte

```
int min(int a, int b) {
   return a < b ? a : b;
}

float min(float a, float b) {
   return a < b ? a : b;
}

String min(String a, String b) { // lexikographisch
   return a.compareTo(b) < 0 ? a : b;
}</pre>
```

• Für jeden Typ? Wie kann sort implementiert werden?







Klassenhierarchie mit zentraler Basisklasse

```
    z.B. java.lang.Object
```

```
void sort(Object[] feld) { ... }
```

alternativ (nutzt intern Object)

```
void sort(java.util.Vector feld) { ... }
```

- Nutzung primitive Datentypen nicht direkt möglich
 - Object[] ≠ int[]
 - erfordert Wrapper-Klassen wie java.lang.Integer

```
Object[] feld = new Object[10];
feld[0] = new Integer(42);
int i = ((Integer) feld[0]).intValue();
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

33





Weiteres Problem: Typsicherheit

Typsicherheit "untergraben"

```
Vector obst = new Vector();
obst.add(new Apfel());
Object birne = obst.firstElement();
```

Vektor könnte unterschiedlichstes Obst enthalten

```
Object einObst = obst.firstElement();
if (einObst instanceof Birne) {
   Birne birne = (Birne) einObst;
   ...
}
```

- Fallunterscheidung bei 40.000 Obstsorten weltweit?
- Dürfen wir Äpfel mit Birnen vergleichen?
- Wann dürfen wir Typen durch allgemeinere ersetzen?







- Kann ein Objekt einer Oberklasse (eines Typs) durch ein Objekt seiner Unterklasse (Subtyps) ersetzt werden?
- Beispiele:

```
• short <u>istSubtyp</u> int <u>istSubtyp</u> long
```



Viele Programmiersprachen ersetzen Typen automatisch, d.h.

```
long min(long a, long b) {
  return a < b ? a : b;
}</pre>
```

wird auch für shorts und ints verwendet

Immer sinnvoll?

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

35





Typ-Substituierbarkeit: Gegenbeispiel

Nach Reussner: Component Based Software Engineering

```
void successor([in] long a, [out] long b){
  if (a < MAX_LONG) b = a + 1; else ...
}</pre>
```

```
int aInt, bInt;
long aLong = MAX_LONG - 1, bLong;

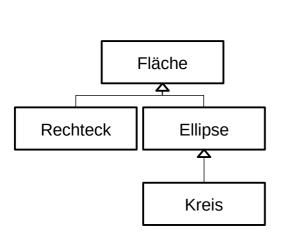
successor(aLong, bLong); // prima (fehlerbehandlung)
successor(aInt, bLong); // prima (immer ok)
successor(aInt, bInt); // overflow oder rücktyp zu klein
successor(aLong, bInt); // rücktyp zu klein
```

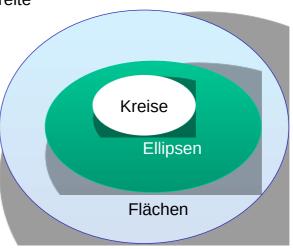


Kreis-Ellipse-Problem



- Modellierung von Vererbungsbeziehungen
 - "Ist ein Kreis eine Ellipse?" "Oder eine Ellipse ein Kreis?"
 - Annahme: Kreis := Ellipse mit H\u00f6he = Breite





Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte







Kreis-Ellipse-Problem /2

Aber ...

```
class Ellipse {
    private int width, height;
    ...
    void skaliereX(int f) { width = width * f; }
    void skaliereY(int f) { height = height * f; }
}
class Circle extends Ellipse { ... }
```

• Nutzung:

```
Circle c = new Circle();
c.skaliereX(2.0);
c.skaliereY(.5);
```









evtl. Reihenfolge in der Klassenhierarchie tauschen?

```
class Circle{
    private int radius;
}

class Ellipse extends Circle{ ... }
Was bedeutet das
für Ellipse?
```

- Verwandte Probleme:
 - · Rechteck-Quadrat
 - Set-Bag

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte







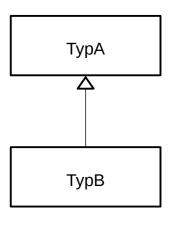
Kovarianz und Kontravarianz

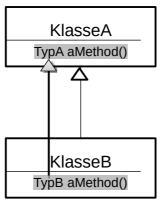
- Geg.: Ordnung von Datentypen von spezifisch → allgemeiner
- Gleichzeitige Betrachtung einer Klassenhierarchie, die Datentypen verwendet
- Kovarianz: Erhaltung der Ordnung der Typen
- Kontravarianz: Umkehrung der Ordnung
- Invarianz: keines von beiden
- Anwendung für
 - Parameter
 - Rückgabetypen
 - Ausnahmetypen
 - Generische Datenstrukturen



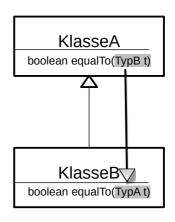








Parametertyp mit der Vererbungshierarchie



Parametertyp entgegen der Vererbungshierarchie

Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte

41





Beispiel

Basierend auf Meyer's SKIER-Szenario

```
class Student {
    String name;
    Student mate;
    void setRoomMate(Student s) { ... }
}
```

- Wie überschreibt man in einer Unterklasse Girl oder Boy die Methode "setRoomMate" in elternfreundlicher Weise?
- Von Eltern sicher gewollt Kovarianz:

```
class Boy extends Student {
    void setRoomMate(Boy b) { ... }
}
class Girl extends Student {
    void setRoomMate(Girl g) { ... }
}
```





Beispiel - Typsicherheit? (I)

Was passiert mit folgendem Code?

```
Boy kevin = new Boy("Kevin");
Girl vivian = new Girl("Vivian");
kevin.setRoomMate(vivian);
```

- Gültig!
 - Verwendet setRoomMate der Basisklasse
 - setRoomMate Methoden der abgeleiteten Klassen überladen nur Spezialfälle
- In C++ und Java keine Einschränkung der Typen zur Compile-Zeit
- Kovarianz so nur in wenigen Sprachen implementiert (z.B. Eiffel über redefine); Überprüfung auch nicht immer statisch!
- Auch bekannt als catcall-Problem

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

43





Beispiel - Typsicherheit? (II)

Ausweg: Laufzeitüberprüfung

```
class Girl extends Student {
    ...
    public void setRoomMate(Student s) {
        if (s instanceof Girl)
            super.setRoomMate(s);
        else
            throw new ParentException("Oh Oh!");
    }
}
```

Nachteile?





Ko- und Kontravarianz für Rückgabewerte

Kovarianz für Rückgabewerte

```
public class KlasseA {
    KlasseA ich() { return this; }
}

public class KlasseB extends KlasseA {
    KlasseB ich() { return this; }
}
```

Kontravarianz?

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



45



Ko- und Kontravarianz – Zusammenfassung

- In objektorientierten Programmiersprachen im Allgemeinen
 - Kontravarianz: für Eingabeparameter
 - Kovarianz: für Rückgabewerte und Ausnahmen
 - Invarianz: für Ein- und Ausgabeparameter





Liskovsches Substitutionsprinzip (LSP)

- Barbara Liskov, 1988 bzw. 1993
 - · definiert stärkere Form der Subtyp-Relation: berücksichtigt Verhalten

Wenn es für jedes Objekt o_1 eines Typs S ein Objekt o_2 des Typs T gibt, so dass für alle Programme P, die mit Operationen von T definiert sind, das Verhalten von P **unverändert** bleibt, wenn o_2 durch o_1 **ersetzt** wird, dann ist S ein Subtyp von T.

- Subtyp darf Funktionalität eines Basistyps nur erweitern, aber nicht einschränken
- Beispiel: Kreis-Ellipse → Kreis als Unterklasse schränkt Funktionalität ein und verletzt damit LSP

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

47





Typsicherheit - Generics in Java

Motivation: Parametrisierung von Kollektionen mit Typen

```
LinkedList<String> liste = new LinkedList<String>();
liste.add("Generics");
liste.add("machen");
liste.add("Spass");
String s = liste.get(0);
```

aber:

```
<del>liste.add(new Integer(42))</del>;
```

oder:

The method ... is not applicable for the arguments ...

```
Integer i = (Integer)liste.get(0);
```





Typsicherheit – Generics in Java

auch für Iteratoren nutzbar

```
Iterator<String> iter = liste.iterator();
while(iter.hasNext()) {
    String s = iter.next();
    ...
}
```

oder mit erweiterter for-Schleife

```
for(String s : liste) {
    System.out.println(s);
}
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte

49



Typsicherheit – Generics in Java

Definition mit Typparameter

```
class GMethod {
    static <T> T thisOrThat(T first, T second) {
        return Math.random() > 0.5 ? first : second;
    }
}
```

- T = Typparameter (oder auch Typvariable) wird wie Typ verwendet, stellt jedoch nur einen Platzhalter dar
- · wird bei Instanziierung (Parametrisierung) durch konkreten Typ "ersetzt"
- nur Referenzdatentypen (Klassennamen), keine primitiven Datentypen





Typsicherheit – Generics in Java

explizite Angabe des Typparameters

```
String s = GMethod.<String>thisOrThat("Java", "C++");
Integer i = GMethod.<Integer>thisOrThat(new Integer(42),
    new Integer(23));
```

automatische Typinferenz durch Compiler

```
String s = GMethod.thisOrThat("Java", "C++");
Integer i = GMethod.thisOrThat(new Integer(42),
    new Integer(23));
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



51



Eingrenzung von Typparametern

Festlegung einer Mindestfunktionalität der einzusetzenden Klasse,
 z.B. durch Angabe einer Basisklasse

```
static<T extends Comparable<T>> T min(T first, T second)
{
   return first.compareTo(second) < 0 ? first : second;
}</pre>
```

- Instanziierung von T muss von Comparable abgeleitet werden: Hier ein Interface, dass wiederum generisch ist, daher Comparable<T>
- Verletzung wird vom Compiler erkannt





Eingrenzung von Typparametern

Angabe des Typparameters bei der Klassendefinition

```
class GArray<T> {
    T[] data;
    int size = 0;

    public GArray(int capacity) { ... }
    public T get(int idx) { return data[idx]; }
    public void add(T obj) { ... }
}
```

- Achtung: new T[n] ist unzulässig!
- Grund in der Implementierung von Generics

Programmierparadigmen (SS 2023): 02 – OO & weitere Konzepte





Einschub: Implementierung von Generics

- Zwei Möglichkeiten der internen Umsetzung generischen Codes:
 - Code-Spezialisierung: jede neue Instanziierung generiert neuen Code
 - \circ Array
String> \rightarrow ArrayString, Array<Integer> \rightarrow ArrayInteger
 - o Problem: Codegröße
 - Code-Sharing: gemeinsamer Code für alle Instanziierungen
 - o Array<String> → Array<Object>, Array<Integer> →
 Array<Object>
 - Probleme: keine Unterstützung primitiver Datentypen & keine Anpassung von Algorithmen an Typ
- Java: Code-Sharing durch Typlöschung (Type Erasure)
- Typen beim Übersetzen geprüft, aber keinen Einfluss auf Code
- sichert auch Kompatibilität zu nicht generischem Code (Java-Version < 1.5)
 - Bsp.: ArrayList vs. ArrayList<E>





Eingrenzung von Typparametern

Reflektion (Metaklassen) zur Erzeugung nutzen:

```
public GArray(Class<T> clazz, int capacity) {
   data = (T[]) Array.newInstance(clazz, capacity);
}
```

Konstruktoraufruf:

```
GArray<String> array =
  new GArray<String>(String.class, 10);
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



55



Kovarianz generischer Typen

einfache Felder in Java sind kovariant

```
Object[] feld = new Object[10];
feld[0] = "String";
feld[1] = new Integer(42);
```

Instanziierungen mit unterschiedliche Typen sind jedoch inkompatibel

```
GArray<String> anArray = new Garray<String>();
GArray<Object> anotherArray = (GArray<Object>) anArray;
```



Wildcards



 Wildcard "?" als Typparameter und abstrakter Supertyp für alle Instanziierungen

```
GArray<?> aRef;
aRef = new GArray<String>();
aRef = new GArray<Integer>();
```

aber nicht:

```
GArray<?> aRef = new GArray<?>();
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



57



Wildcards - Anwendung

hilfreich insbesondere für generische Methoden

```
// dieser Methode ist der genaue Typ egal
static void p0(GArray<?> ia) {
    for(Object o : ia) {
        System.out.print(o);
    }
}

// floats wollen wir mit Genauigkeit 2 haben
static void pF(GArray<Float> ia) {
    for(Float f : ia) {
        System.out.printf("%5.2f\n", f);
    }
}
```







nach "unten" in der Klassenhierarchie → Kovarianz

```
? extends Supertyp
```

 Anwendungsbeispiel: Sortieren eines generischen Feldes erfordert Unterstützung der Comparable-Schnittstelle

```
void sortArray(GArray<? extends Comparable> array) {
    ...
}
```

Programmierparadigmen (SS 2023): 02 - OO & weitere Konzepte



59



Beschränkte Wildcards

nach "oben" in der Klassenhierarchie → Kontravarianz

```
? super Subtyp
```

Anwendungsbeispiel: Feld mit ganzen Zahlen und Objekten

```
GArray<? super Integer> array;
// Zuweisungskompatibel zu ...
array = new GArray<Number>();
array = new GArray<Object>();
array = new GArray<Integer>();
// aber nur erlaubt:
Object obj = array.get(0);
```





Zusammenfassung

- Ziel des Kapitels: Einführen von Mechanismen zur Handhabung von komplexeren Code
- Unterschiedliche Einzelfunktionen:
 - Systematisiertes & schnelles Testen
 - Inspektion/Veränderung von Code zur Laufzeit
 - · Zusichern von Bedingungen
 - Fehlerbehandlung
 - Typsicherheit
 - · Generische und wiederverwendbare Algorithmen

