

---

# Vorgefertigte Teile – Hindernisse

Hinderung an der Verwendung fertiger Teile durch:

- Unkenntnis
- unterschiedliche Modelle
- mangelndes Vertrauen
- „Ich kann es besser“

Verwendung zahlt sich aus, ist aber mit Arbeit verbunden

---

# Top Down

```
public static void main(String[] args) {
    List<Integer> nums = readNums();
    Collections.sort(nums);
    print(nums);
}

private static List<Integer> readNums () {
    Scanner sc = new Scanner(System.in);
    List<Integer> nums = new LinkedList<Integer>();
    while (sc.hasNextInt())
        nums.add(sc.nextInt());
    return nums;
}

private static void print(List<Integer> nums) {
    for (int i: nums)
        System.out.println(i);
}
```

---

# Bottom Up

```
public class SortedNums {
    private GenTree<Integer> nums = new GenTree<Integer>();
    public void readFrom(Scanner in) {
        while (in.hasNextInt())
            nums.add(in.nextInt());
    }
    public void printTo(PrintStream out) {
        for (int i: nums)
            out.println(i);
    }
    public static void main (String[] args) {
        SortedNums nums = new SortedNums();
        nums.readFrom(new Scanner(System.in));
        nums.printTo(System.out);
    }
}
```

---

# Qualitätssicherung

---

# Qualitätssicherung – Themenübersicht

- Spezifikation ..... festlegen was zu erreichen ist
- Programmverständnis ..... verstehen ohne probieren
- Testen ..... praktisch ausprobieren
- Ablauf nachvollziehen ..... interne Details nachprüfen
- Ausnahmebehandlung ..... was tun im Fehlerfall?
- Validierung ..... Daten, Programme brauchbar?

---

# Spezifikationen

---

# Was ist eine Spezifikation?

- mehr oder weniger rigorose Beschreibung eines Systems
- Anforderungsdokumentation
- wird im Laufe der Entwicklung genauer (Zusicherungen)
- spätestens Implementierung macht Spezifikation formal
- Spezifikation ist Basis für
  - statisches Verstehen
  - Verifikation
  - Testen

---

# Design by Contract (DbC)

- Spezifikation als Vertrag zwischen Server und Client
- Objekt = Server: bietet Dienste (Methoden) an
- Objekt = Client: nutzt Dienste eines Servers
- Bestandteile des Software-Vertrags sind
  - Methodensignaturen in Interfaces und Klassen
  - durch Namen suggerierte Eigenschaften
  - Zusicherungen auf Schnittstellen (auch Kommentare)
  - Usancen = übliche Gepflogenheiten (stillschweigend)



---

# Typischer Software-Vertrag

Client kann Nachricht an Server senden wenn:

- entsprechende Methode in Server sichtbar
- Argumenttypen Untertypen der formalen Parametertypen
- alle Vorbedingungen der Methode erfüllt

Server garantiert nach Ausführung der Methode:

- Objekt des Ergebnistyps zurückgegeben
- Nachbedingungen der Methode erfüllt
- Invarianten und History-Constraints des Servers erfüllt  
(Achtung: Invariante  $\neq$  Schleifeninvariante)

---

# Zusicherungen auf Objektschnittstellen

Arten aus fortlaufendem Text herauslesen:

**Vorbedingung:** Einschränkung auf Parameter typisch

- alles, worum sich Aufrufer kümmern muss
- Bedingung, die vor Methodenausführung erfüllt sein muss

**Nachbedingung:** was Methode tun und zurückgeben soll

- Großteil der Zusicherungen auf Methodenschnittstelle

**Invariante:** unveränderliche Eigenschaft des Objekts

- bezieht sich auf ganzes Objekt (nicht einzelne Methode)

**History-Constraint:** Änderbarkeit des Objektzustands

- bezieht sich auf Objekt oder Gruppe von Methoden

---

# Zusicherungen und Untertypen

## Vorbedingung:

- im Untertyp schwächer als im Obertyp
- im Untertyp Verknüpfung mit ODER
- Bsp.: Obertyp:  $x > 0$  bzw.  $y \neq \text{null}$   
Untertyp:  $x \geq 0$  bzw. jedes  $y$  OK

## Nachbedingung, Invariante, History-Constraint:

- im Untertyp stärker als im Obertyp
- im Untertyp Verknüpfung mit UND
- Bsp.: Obertyp:  $x \geq 0$  bzw. jedes  $y$  OK  
Untertyp:  $x > 0$  bzw.  $y \neq \text{null}$

---

# Usancen

- nicht ausdrücklich angeschriebene übliche Zusicherungen
  - gelten dennoch, weil man sich darauf verlässt
- wichtig: *Objektzustand darf sich nicht unerwartet ändern*
- Änderung des Objektzustands verboten, wenn nicht durch Methodennamen oder Zusicherungen ausdrücklich erlaubt
- Änderungen, die kein Client sehen kann, sind aber erlaubt
  - z.B. Teilergebnisse speichern statt immer neu berechnen
- durch Namen suggerierte Eigenschaften müssen gelten

---

# Zusicherungen sparsam einsetzen

- Zusicherungen erhöhen Programmkomplexität
- je weniger explizite Zusicherungen nötig, desto besser
- aber weglassen nötiger Zusicherungen sehr gefährlich
- einfachen Code mit komplexen Zusicherungen vermeiden:

```
// gib mittlere Zahl in nums zurück
// nums sortiert (Bedingung leicht zu übersehen)
public static int median (int[] nums) {
    return nums[nums.length / 2];
}
```

- Abstraktion bietet Ausweg

---

# Abstraktion kapselt Zusicherung

```
import java.util.Arrays
public class SortedIntArray {
    private int[] elems;    // elems bleibt stets sortiert
    public SortedIntArray(int[] e) {
        elems = Arrays.copyOf(e, e.length);
        Arrays.sort(elems);
    }
    public int median() { // gib mittlere Zahl zurück
        return elems[elems.length / 2];
    }
    public boolean member(int x) { // x enthalten?
        ... /* binäre Suche */
    }
}
```

---

# Abstraktion entlastet Client

- Anwender von `SortedIntArray` braucht nicht zu wissen, dass `elems` sortiert sein muss
- Konstruktor in `SortedIntArray` muss auch bei Instanziierung von Unterklassen ausgeführt werden:

```
public class Sub extends SortedIntArray {  
    public Sub(int[] e) {  
        super(e);  
    }  
}
```

- Überprüfung von Zusicherungen auf eine Klasse beschränkt und daher wenig fehleranfällig

---

# Arten der Typäquivalenz

**Namensgleichheit** in statischen Sprachen wie Java

- explizite Spezifikation der Untertypbeziehungen
- Abstraktion und Untertypbeziehungen gekoppelt
- umständlicher, aber intuitiver und sicherer

**Strukturgleichheit** in dynamischen Sprachen

- Untertypbeziehungen ergeben sich automatisch
- Abstraktion von Untertypbeziehungen entkoppelt
- erfordert viel Programmierdisziplin