Zusicherungen (Fortsetzung)

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 sich nicht um Sortieren kümmern

Konstruktor von `SortedIntArray` ist auch in Unterklassen auszuführen:

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

Überprüfung von Zusicherungen auf eine Klasse beschränkt

$\longrightarrow$ wenig fehleranfällig

Statisches Programmverständnis

Verifikation einer Klasse

für jede Methode und jeden Konstruktor:

- Annahme: Vorbedingungen und Invarianten erfüllt
- müssen Nachbedingungen und Invarianten ableiten aus Rumpf, Vorbedingungen und Invarianten
- verwenden Zusicherungen im Rumpf als Zwischenschritte

für jedes Nachrichtensenden (= Methodenaufruf):

- müssen Vorbedingungen aufgerufener Methode ableiten
- müssen Invarianten des Aufrufers ableiten,
  außer wenn `this` in aufgerufener Methode sicher nicht sichtbar

assert-Anweisungen

Überprüfungen zur Laufzeit oft sehr aufwendig,
daher Überprüfungen normalerweise ausgeschaltet

einschalten mit: `java -ea Programm`

es kommt nicht auf Überprüfungen zur Laufzeit an,
sondern auf statische Überprüfungen durch Programmierer

Schleifeninvarianten

Schleifen schwierig zu überprüfen wenn Anzahl der Iterationen unbekannt

Schleifeninvariante = Bedingung, die in jeder Iteration gilt:

```
assert Schleifeninvariante;
while (!Abbruchbedingung) {
    assert Schleifeninvariante;
    Schleifenrumpf;
    assert Schleifeninvariante;
}
assert Schleifeninvariante && Abbruchbedingung;
```

Schleifeninvarianten bereits bei Programmkonstruktion festlegen
(im Nachhinein viel schwieriger zu finden)

Aufgabe: Überprüfen von Zusicherungen

Suchen Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Frage:

Warum kommt es bei Zusicherungen auf die statische Überprüfung beim Programmieren an, nicht auf Überprüfungen zur Laufzeit?

Zeit: 2 Minuten

Beweisen = verständlich machen

statisches Programmverstehen $\approx$ Korrektheit beweisen

Ziel ist es, komplizierte Aussagen so weit zu vereinfachen, dass sie leicht verständlich sind (z.B. über Kommentare)

Ziel ist Einfachheit, NICHT Verkomplizierung
(z.B. durch oft unnötige Formalisierung)

dahinter steckt Denkweise, nicht reine Mathematik

Automatisches Beweisen

große Fortschritte (z.B. Model Checking),
aber trotzdem nur für klare und eher einfache Fragestellungen

erwünschte und unerwünschte Eigenschaften liegen oft nahe beisammen

Beispiel: einfache und rasche Bedienung einer Webseite erwünscht
→ führt leicht zu Denial-of-Service-Attacken
→ Lösung (z.B. Passworteingabe) verhindert einfache Bedienung

Automatisches Beweisen kein Ersatz für statisches Programmverstehen

sehr sinnvoll: **Code-Review**