
Statisches Programmverständnis

Verifikation einer Klasse

- für jede Methode und jeden Konstruktor:
 - Annahme: Vorbedingungen und Invarianten erfüllt
 - Nachbedingungen und Invarianten ableiten aus Rumpf, Vorbedingungen und Invarianten
 - Zusicherungen im Rumpf als Zwischenschritte
- für jedes Nachrichtensenden (= Methodenaufruf):
 - Vorbedingungen aufgerufener Methode ableiten
 - 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 Überprüfungen durch Programmierer

Schleifeninvarianten

- Schleifen schwierig zu überprüfen wenn Anzahl der Iterationen unbekannt
- Schleifeninvariante = Bedingung, die in jeder Iteration gilt:

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

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

Rekursive Aufrufe

- Vor- und Nachbedingungen statt Schleifeninvarianten
- rekursive Aufrufe nicht nur am Ende des Rumpfes
- Vor- und Nachbedingungen meist voneinander verschieden
- daher bietet Rekursion mehr Freiheiten beim Programmieren und bei der Dokumentation als Schleifen
- Vorgehen bei der Konstruktion rekursiver Methoden aber ähnlich wie bei der Konstruktion von Schleifen

Beweisen = verständlich machen

- statisches Programmverstehen \approx Korrektheit beweisen
- Ziel ist es, komplizierte Aussagen so weit zu vereinfachen, dass sie leicht verständlich sind (z.B. Kommentare)
- Ziel ist Einfachheit, NICHT Verkomplizierung (z.B. durch oft unnötige Formalisierung)
- dahinter steckt Denkweise, nicht reine Mathematik

Termination

Korrektheit und Termination

- Schleifeninvarianten \Rightarrow Korrektheit jeder Iteration
- Obergrenze für Anzahl der Iterationen \Rightarrow Termination
- Notwendig: jede Iteration bringt uns näher zum Ziel
- Berechnung einer Obergrenze für Iterationen
- ähnelt Aufwandsabschätzung, aber Unterschiede im Detail (Subtraktion gut, Division schlecht)

Termination / Aufwandsabschätzung

- binäre Suche: Abbruch wenn $low > high$; pro Iteration:
 $low = middle + 1$ oder $high = middle - 1$
wobei $middle = (low + high) / 2$
- Termination: vor Änderung stets $low \leq middle \leq high$
Iteration verringert $high - low$ um mind. 1
Obergrenze: $high - low = elems.length$
- Aufwand: jede Iteration halbiert $high - low$ (ungefähr)
daher logarithmischer Aufwand
wenn $high == low$ ändert Halbierung nichts
Halbierung für Termination nicht ausreichend

Fortschritt und Termination

- jede Iteration entspricht einer Zahl in einer Reihe
(z.B. $j - i$ wobei i bzw. j sich in jeder Iteration ändert)
- es muss eine Grenze geben (z.B. $j - i \geq 0$)
- jede Iteration muss Zahl näher an die Grenze bringen
(z.B. $j - i$ wird in jeder Iteration um mind. 1 kleiner)
- der Fortschritt muss ungefähr konstant sein
(darf sich nicht pro Schritt verringern, z.B. halbieren)
- Überlegungen zur Termination beim Programmieren

Automatisches Beweisen

- große Fortschritte (z.B. Model Checking)
- nur für klare Fragestellungen
- erwünschte/unerwünschte Eigenschaften nahe beisammen
Beispiel: einfache, rasche Bedienung erwünscht
führt leicht zu Denial-of-Service-Attacken
Lösung (z.B. Passwort) verhindert einfache Bedienung
- kein Ersatz für statisches Programmverstehen
- sehr sinnvoll: Code Review

Testen

Testen und Softwarequalität (1)

- Testfälle können nicht alles abdecken
- gefundene Fehler \Leftrightarrow vorhandene Fehler (Beurteilung)
- 1 Fehler ausgebessert \Rightarrow 10 Fehler eingebaut
- Fehler in jeder Softwareentwicklungsphase
- nicht alle Fehler werden sichtbar (aber für wen?)
- seltene Fehler sind großes Sicherheitsrisiko (Eindringen)
- „Fehler“ können Absicht sein (Eindringen ermöglichen)

Testen und Softwarequalität (2)

- Testen kann Problemstellen aufdecken
- aber Qualitätssteigerung nur durch statisches Verstehen (z.B. Code Reviews)
- Testfälle zur Spezifikation verwendbar

Teststufen

- Unittest
- Integrationstest
- Systemtest
- Abnahmetest

Testmethoden

- Black-Box-Test (Testen am Ende)
- White-Box-Test (Programm und Testfälle gleichzeitig)
- Grey-Box-Test (Testfälle zuerst)

Testarten

- Funktionaler Test
- Nichtfunktionaler Test
- Schnittstellentest
- Oberflächentest
- Stresstest (z.B. Crasch- oder Lasttest)
- Sicherheitstest
- Regressionstest