UNIT-TESTEN

Studientag 01853

Moderne Programmiertechniken und -methoden

UNZULÄNGLICHKEITEN IM DBC

- nicht alle Fälle getestet
- keine Garantie, dass ein bestimmter Programmpfad ausgeführt wurde
- beschränkte Ausdrucksstärke

TESTFALL

- Definition: Spezifikation einer Eingabe und der dazu erwarteten Ausgabe
  - ist also formulierte partielle Spezifikation
- Vorteil: kann bestimmten Ausführungspfad erzwingen
- testet eine Einheit eines Programms: Methode, Klasse, Paket

TESTEN

- Regressionstests: wiederholte Ausführung
  - stellen automatisiert sicher, dass Änderungen am Code keine Fehler hervorrufen
- Test-Frameworks: für Programme, die Programme testen
- Test-Suite: umfasst mehrere unabhängige (Reihenfolge ist für Erfolg irrelevant) Testfälle

JUNIT 3

- Success, Failure (unerfüllte Ergebnisvorgabe) und Error (z.B. unerwartete Exception)
- Tests sind Subklassen von TestCase, für jede Test-Methode (Name beginnt mit test) eine Instanz
- TestCase ist Subklasse von Assert
- Fixtures mittels setUp, tearDown
- Ausführungssteuerung mittels TestResult

JUNIT 4

- Tests durch @Test markiert
- Fixtures mittels @Before, @After (bzw. @BeforeClass, @AfterClass)
- Assertions als Statische Methoden aus Assert

JUNIT 4.4

- assertThat und Hamcrest
- Vorbedingungen an Tests: assume
- allquantifizierte Theorien: @Theory
  - benötigte Parameter über @DataPoint, bei mehreren Parametern permutiert

VERERBUNG VON TESTFÄLLEN

- nicht immer sinnvoll, da
  - Testfälle nach Fixtures organisiert sind und ggf. mehrere Klassen in einem Testfall getestet werden können
    - parallele Hierarchie von Tests und zu testender Klassen nicht unbedingt gegeben
- Test von Superklassen aufgrund veränderten Verhaltens in Subklassen fehlschlagen können

TESTEN VON INTERFACES

- Testfälle können programmiert, aber ohne Implementierung nicht getestet werden

- Testfälle gehören logisch gesehen zum Interface, müssen aber im Rahmen der Implementierung ausgeführt werden

- Lösung: abstrakte Test-Klasse mit abstrakter Factory-Methode für ein das Interface implementierendes Objekt

  - Nachteil: erhöhtes Wachstum an Test-Klassen

MOCK-OBJEKTE

- stellen beispielhafte Implementierungen von Klassenabhängigkeiten dar
- überprüfen, ob Methoden entsprechend der Erwartungen aufgerufen werden (Anzahl und ggf. Reihenfolge)
- Mock-Objekt muss zuweisungskompatibel zum Zielobjekt sein
- Erzeugung von Mock-Objekten zur Laufzeit durch Meta-Programmierung

EINBINDEN VON MOCK-OBJEKTEN

- Dependency Injection
- globaler Austausch der Klasse von realem und Mock-Objekt
  - Problem: auf beide muss ggf. zugegriffen werden können, aber beide können nicht gleichzeitig in der Hierarchie vorhanden sein
- über AOP den Konstruktoraufruf des Zielobjekts abfangen und Mock-Objekt zurückgeben

NUTZUNGSSZENARIEN

- i.d.R. Mock-Objekte bei anbietenden Interfaces (Client/Server)
- aber auch bei ermöglichenden
  - Mock-Objekt kann als Testorakel unerwartetes Verhalten provozieren
  - oder es prüft die erwartete Aufruf-Reihenfolge

GRENZEN DER EINSETZBARKEIT

- Klassenmethoden in Java (kein dynamisches Binden)
- finale Methoden (können nicht überschrieben werden)
- erforderliche Seiteneffekte, die durch Mock-Objekte nicht erfüllt werden

VON DER FAILURE ZUM FEHLER

- manuelles Binden von Testfällen an Programmeinheiten (z.B. mit Annotationen)
  - viel Wartungsaufwand
- erstellen eines Aufrufgraphen (statische Programmanalyse)
  - zu umfangreich und enthält viele Pfade, die nicht durchschritten werden
  - keine Erfassung von Aufrufen über Reflection

VON DER FAILURE ZUM FEHLER II

- Tracen des Programmablaufs
- Fehlerlokatoren
  - abdeckungsbasiert: Wahrscheinlichkeit eines Fehlers ([0;1] oder {0;1})
  - modellbasiert: anhand von Klauselmengen und logischer Betrachtung
  - historische Betrachtung: welche Änderungen kamen hinzu

TESTS TESTEN

- Problem: falsch negative Ergebnisse
  - Test passiert trotz Fehler in getesteter Einheit
- Error/Defect-Seeding: bewusster Einbau von Fehlern um Testfälle scheitern zu lassen (schwer automatisierbar)
- Mutation Testing: zufälliges Ändern des Programms (gut automatisierbar)

FAZIT

- DbC und Unit-Tests haben das gleiche Ziel: die Überprüfung der Einhaltung der Spezifikation
  - Ansatz: Korrektheit des Systems ergibt sich aus der Korrektheit der Komponenten
- statische Typprüfung ist quasi Verified DbC
  - kleinere Fehlerklasse, dafür formaler Beweis

FAZIT II

- Unit-Tests: prüft korrekte Abbildung von Ein- auf Ausgaben für genau spezifizierte Fälle
  - aber nur Fallweise
- DbC: prüft Eingaben auf Zulässigkeit und Ausgaben auf Korrektheit
  - umfassend, weil zur Laufzeit jede Paarung geprüft wird
  - aber nur Fallweise

FAZIT III

- Verified DbC: formaler Beweis, dass die Einhaltung der Nachbedingungen aus der Einhaltung der Vorbedingungen folgt
  - nicht umfassend umgesetzt