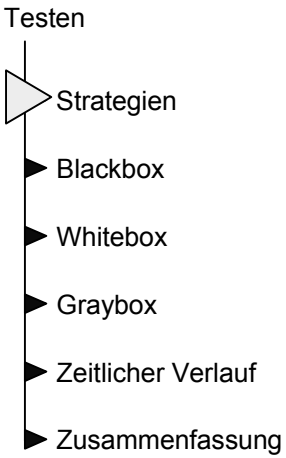
Test-Strategien

- Grundsätzliches
- Blackbox-Testen
- Whitebox-Testen
- Graybox-Testen
- Ablauf von Tests
- Zusammenfassung

Grundsätzliche Test-Strategien

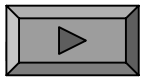


- Blackbox-Tests
  - Äquivalenzklassenbildung
  - Randwertanalyse
- Whitebox-Tests
  Man unterscheidet verschiedene Überdeckungsgrade:
  - Statement Coverage
  - Decision Coverage, Branch Coverage
  - Condition Coverage
  - Decision / Condition Coverage
  - Multiple Condition Coverage
- Graybox-Tests
  - Mischung aus Black- und Whitebox-Tests
- Test First
  - Whitebox-Test mit Regressionstest-Charakter (s. JUnit)

Blackbox-Tests: Allgemeines

- Andere Bezeichnungen
  Data-driven Tests, I/O-driven Tests, funktionales Testen
- Der Tester sieht das Programm als Black Box, d. h. die innere Struktur des Programms interessiert nicht
  Testfälle (engl: test cases) werden ausschließlich aus der Spezifikation abgeleitet
- Der Tester ist interessiert am Finden von Umständen (Testfälle), in denen das Programm nicht mit seiner Spezifikation übereinstimmt

- Diskutieren Sie mit einem Partner:
  - Welche und wie viele Fälle muss man bei einem Blackbox-Programm testen, um sicher zu sein, dass es korrekt ist?
  - Woher weiß man, wann es soweit ist, d.h. wann man genug getestet hat?
- Dauer: 3 Minuten



Beispiel für erschöpfendes Testen

- Ein Sortierprogramm verwendet ShakerSort für Zahlenfolgen mit bis zu 15 Elementen, bei mehr als 15 Elementen wird QuickSort verwendet
  Die “interne Grenze” 15/16 hat sich der Implementierer ausgedacht (vielleicht gemessen), sie tritt in der Spezifikation nicht auf, ist also für den Blackbox-Tester völlig unvorhersehbar

```
void sort ( Object [ ] sortArray ) {
    if ( sortArray.length <=15 )
        shakerSort( sortArray );
    else
        quickSort( sortArray );
}
```

- Die Chance, diesen (nicht böswilligen!) Fehler mittels Blackbox-Tests zu entdecken, ist ziemlich gering
- Folge:
  Vermutlich wird die sort-Methode nie mit mehr als 15 Objekten getestet und der QuickSort-Algorithmus bleibt völlig ungetestet

Testen aller möglichen Eingaben

- Erschöpfendes Testen bedeutet, jede mögliche Eingabe wird als Testfall vorgesehen und das Programm damit getestet

→ Das bedeutet eine unendliche Menge an Testeingaben

→ Erschöpfende Blackbox-Tests sind nicht möglich

Die genannten Beispiele sind Mini-Probleme!

Blackbox-Tests: Persistenz

- Das Problem ist noch größer, wenn Programme mit persistenten Daten arbeiten
  Ihr Verhalten ist dann nicht nur von den aktuellen Eingabedaten abhängig, sondern von ihrer Geschichte (besonders Reihenfolge!)

  Beispiel
  - In einem Programm mit einer Kunden-Datenbank können aus der Datenbank heraus automatisch Rechnungen erstellt werden
  - Bezahlte Beträge werden in der Datenbank markiert
  - Rechnungen werden nur erstellt, wenn der Betrag noch nicht als "bezahlt" gekennzeichnet ist
  - Die Reihenfolge
    Rechnung drucken $\leftrightarrow$ Betrag als "bezahlt" markieren
    beeinflusst den Zustand, damit den Programmablauf und damit die Tests, die zustandsabhängig erfolgen müssen

  $\rightarrow$Es müssen alle Reihenfolgen aller möglichen Eingaben getestet werden

Blackbox-Tests: Folgerungen

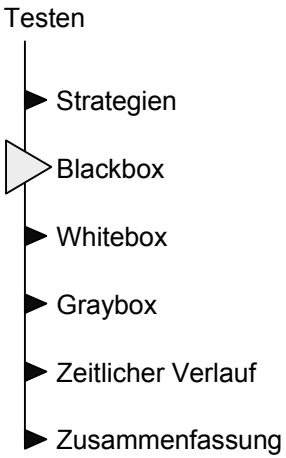
- Blackbox-Tests sind nützlich und sehr verbreitet

Aber

- Blackbox-Tests können keine Fehlerfreiheit garantieren
- Blackbox-Tests sind unökonomisch:
  Es sind (auch) Informationen über die Programmstruktur notwendig, um sinnvolle Testfälle zu finden

→ Es gibt Möglichkeiten, ihre Effizienz deutlich zu verbessern

Blackbox-Tests: Testklassen-Bildung



- Eine allgemeine Regel sollte beim Erstellen von Testfällen beachtet werden:
  Ein guter Testfall reduziert die Zahl von notwendigen weiteren Tests soweit wie möglich, um ein vordefiniertes Ziel für "vernünftiges" Testen zu erreichen
- Das impliziert, dass es günstig ist, den Eingaberaum in Äquivalenzklassen aufzuteilen
- Aus diesen Überlegungen folgt eine Technik namens Equivalence Partitioning, deutsch: Testklassen- oder Äquivalenzklassen-Bildung

Äquivalenzklassen-Bildung 1/3

- Es gibt 2 Arten von Äquivalenzklassen:
  - Gültige Äquivalenzklassen repräsentieren gültige Eingaben für ein Programm
  - Ungültige Äquivalenzklassen repräsentieren alle anderen Eingaben
- Heuristische Regeln zur Identifizierung von Äquivalenzklassen:
  - Wenn eine Eingabebedingung einen Bereich oder eine Anzahl von Werten (z. B. "gültige Eingaben sind 1 .. 999") angibt, identifiziere eine gültige Äquivalenzklasse (test1=500) und zwei ungültige Äquivalenzklassen (test2=-100 und test3="100)
  - Wenn eine Eingabebedingung eine Menge von Werten spezifiziert (z. B. "Ampelfarben sind rot, gelb und grün"), dann identifiziere eine gültige Äquivalenzklasse für jedes Mengenelement (test1=rot, test2=gelb, test3=grün) und eine ungültige Äquivalenzklasse für ein Nicht-Element (test4=blau)

Äquivalenzklassen-Bildung 2/3

- Weitere heuristische Regeln:
  - Wenn eine Eingabebedingung eine Muss-Situation kennzeichnet (z. B. "ein Bezeichner in Java muss mit einem Buchstaben beginnen"), identifiziere eine gültige Äquivalenzklasse (test1="einDatum": Bezeichner beginnt mit einem Buchstaben) und eine ungültige Äquivalenzklasse (test2="2Daten": Bezeichner beginnt mit einer Zahl)
  - Wenn es einen Grund gibt, anzunehmen, dass Elemente einer Äquivalenzklasse nicht einheitlich behandelt werden, splitte diese Äquivalenzklasse in verschiedene Unterklassen

Äquivalenzklassen-Bildung 3/3

Wenn alle Äquivalenzklassen identifiziert sind, werden daraus wie folgt Testfälle abgeleitet:

1. Vergib eine eindeutige Nummer für jede Äquivalenzklasse
2. Bis alle gültigen Äquivalenzklassen abgedeckt sind, entwirf einen neuen Testfall, der so viele wie möglich von bisher nicht abgedeckten gültigen Klassen abdeckt
   (damit möglichst wenige Testfälle gebraucht werden)
3. Bis alle ungültigen Äquivalenzklassen abgedeckt sind, entwirf einen neuen Testfall, der genau eine der bisher nicht abgedeckten ungültigen Klassen abdeckt

Randwertanalyse 1/2

Die Boundary Value Analysis (Randwertanalyse) unterscheidet sich in zwei Punkten von der Äquivalenzklassen-Bildung:

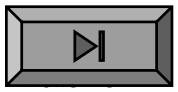
- Anstatt aus einer Äquivalenzklasse einen beliebigen Wert als Repräsentanten für einen Testfall zu wählen, verlangt die Randwertanalyse, dass ein oder mehrere Werte so gewählt werden, dass die Enden der Wertebereiche jeder Äquivalenzklasse überprüft werden
- Anstatt sich nur auf den Eingaberaum zu konzentrieren, wird auch der Ergebnisraum bei der Bildung der Testfälle beachtet

Randwertanalyse 2/2

Heuristisch gute Regeln zur Erstellung von Testfällen:

- Wenn eine Eingabebedingung einen Bereich oder eine Anzahl von Werten (z. B. 1 bis 999) angibt, entwirf Testfälle für die Enden des gültigen Wertebereiches (1 und 999) und für Werte, die direkt außerhalb liegen (0 und 1000)
- Wende diese Regel auf die Ausgabebedingungen an, d. h. entwirf Testfälle so, dass die Enden der gültigen Ausgabebereiche erreicht werden und versuche Testfälle so zu wählen, dass die Ergebniswerte gerade außerhalb des gültigen Wertebereichs liegen würden; Beispiel:
  Das Ergebnis einer Sinusfunktion sollte +1.0 und -1.0 erreichen (für die Eingabewerte $^1/_2\ \pi$ und $^3/_2\ \pi$)
- Wenn eine Eingabebedingung eine geordnete Liste von Werten spezifiziert, entwirf Testfälle, die sich auf das erste und das letzte Element der Menge konzentrieren

Fragen?



Zeitlicher Verlauf

- Unit-Test, Modultest, Komponententest (auch: Klassentest)
  - Fokus: Funktionalität, Sonderfälle, Performanz etc.
- Integrationstest
  - Inkrementell vs. "alles auf einmal"
  - Fokus: Schnittstellen, Kommunikation
- Systemtest
  - In der realen Umgebung, ohne Auftraggeber
  - Fokus: Vollständigkeit, Konfiguration, Interoperabilität, Performanz etc.
- Abnahmetest
  - In der realen Umgebung durch den Auftraggeber oder
  - In Form von Alpha- und Beta-Tests für einen anonymen Markt

Regressionstests

Regressionstests dienen der Überprüfung bereits erreichter Testergebnisse nach einer Änderung

- Testfälle und –Ergebnisse werden protokolliert (z.B. in einer Testdatenbank)
- Nach einer Änderung werden die Testfälle (automatisch) wieder durchgespielt
- Die Ergebnisse können mit Sollergebnissen und den in der Vergangenheit erreichten Ergebnissen verglichen werden
- (Nur) Abweichungen werden gemeldet
  → Hohe Effektivität

Testen

- Strategien
- Blackbox
- Whitebox
- Graybox
- Zeitlicher Verlauf
- Zusammenfassung

Fragen?

Zusammenfassung

- Testen kann nur die Anwesenheit von Fehlern aufzeigen, niemals deren Abwesenheit
- Testen ist nur auf ausführbarem Code, also erst am Ende des Entwicklungsprozesses möglich;
  daher sind spät (!) entdeckte Fehler oft teuer zu beheben
- Es ist (normalerweise) nicht möglich, ein Programm erschöpfend zu testen: das gilt für Blackbox- wie für Whitebox-Tests
- Man unterscheidet
  - Blackbox, Whitebox- und Graybox-Tests
  - Im zeitlichen Verlauf:
    Unit-/Modul-/Komponenten-Tests, Integrationstests, Systemtest, Abnahmetest
  - Bei wiederholtem Testen nach Änderungen: Regressionstests
- Die Kombination verschiedener Verfahren liefert eine gute Überdeckung
- Es gibt deutlich effizientere Maßnahmen, z. B. Inspektionen