Vorlesung „ Methoden des Software Engineering“

Block D „Qualitätssicherung“

Testen objekt-orientierter Programme

Martin Wirsing

Einheit D.3, 23.12.2003

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

Ziele

- Lernen Zustandsautomaten mit Hilfe des Modellprüfungssystems HUGO zu analysieren
- Zusätzliche Probleme beim Testen objekt-orientierter Programm verstehen und Strategien zum Testen solcher Programme kennen lernen

Validieren und Testen objekt-orientierter Systeme

- Das Validieren objekt-orientierter Systeme beginnt mit Analyse und Konsistenzprüfung von Anforderungs- und Entwurfsmodellen
- Die Teststrategie ändert sich
  - Auf Grund der Kapselung erweitert sich das Konzept der "Einheit" (Unit) von Programmfragmenten auf Klassen
  - Integration geschieht auf der Ebene von Klassen oder Komponenten und deren Ausführung anhand von Anwendungsfällen
  - Validierung benützt konventionelle Black Box Techniken
- Validierung und Testen basiert auf UML Spezifikationen
  - Anwendungsfälle
  - Klassendiagramme
  - Zustandsübergangsdiagramme
  - …

Test und Modellprüfung von Zustandsautomaten

- Testobjekte mit interessantem dynamischem Verhalten
- Voraussetzung: Zustandsautomat bzw. Ablaufdiagramm liegt als Spezifikation des Testobjekts vor, beispielsweise
  - Statecharts
  - Objektlebenszyklus aus OOA-Modellierung
  - Aktivitätsdiagramme
- Ein Testfall beschreibt eine **Folge von Ereignissen**, die auf das Testobjekt ausgehend von seinem Anfangszustand angewendet werden.
- Für jeden Schritt dieser Folge wird als Soll der nächste erwartete Zustand und die erwartete Ausgabe festgelegt.
- Bei der Modellprüfung werden zusätzlich Sicherheits- und Lebendigkeitseigenschaften überprüft.

UML Model Checking

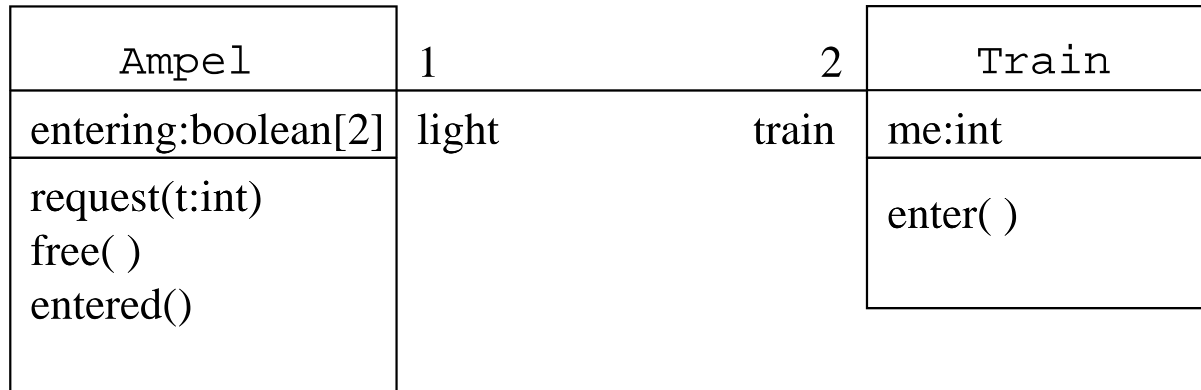
(1) Übersetzung von UML-Diagrammen in vorhandene Model checking-Eingabesprachen (z. B. Vooduu, Hugo/RT)

(2) Model checking auf UML-Diagrammen (z. B. Jack)

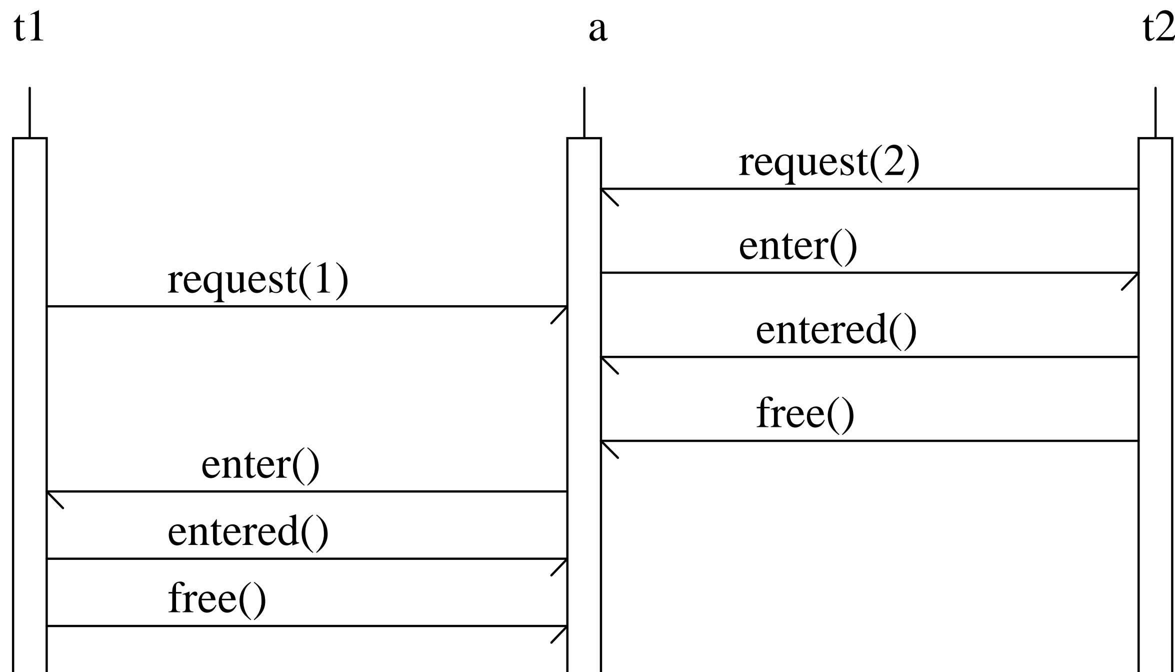
Hier HUGO/RT:

- Übersetzung von UML-Zustandsmaschinen und UML-Kollaborationen in vorhandene Model checking-Eingabesprachen (SPIN, UPPAAL)
- Verwendung einer Zwischenrepräsentation, um UML-Konzepte handhabbar zu machen
- Rückübersetzung der Ergebnisse des Model checkers

Beispiel Eisenbahnsteuerung: Klassendiagramm und Testfall

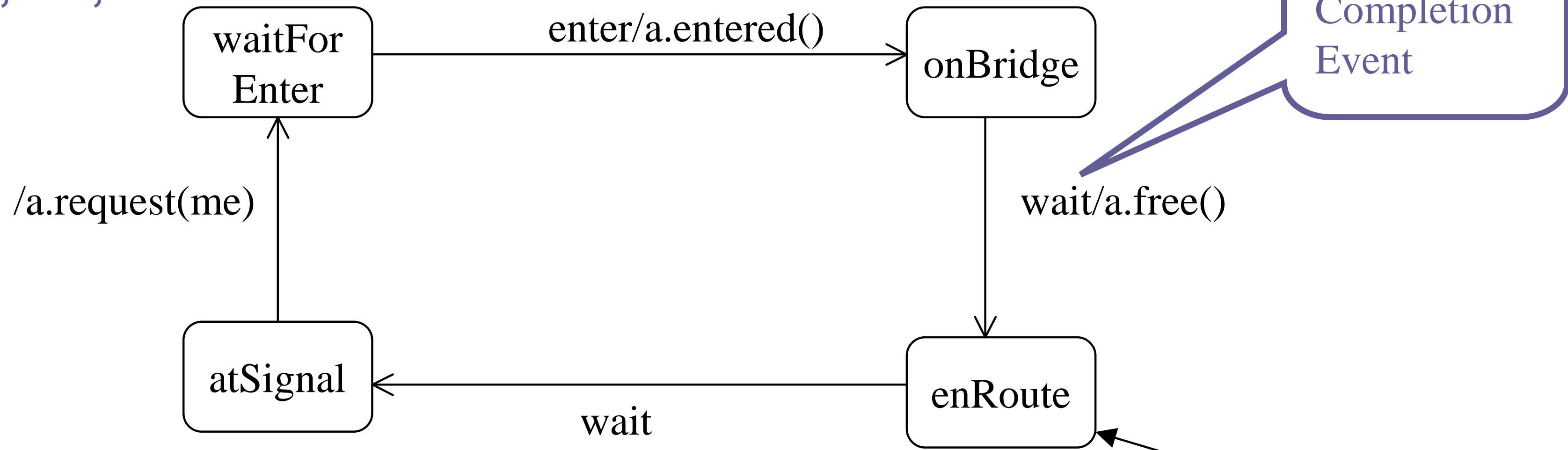


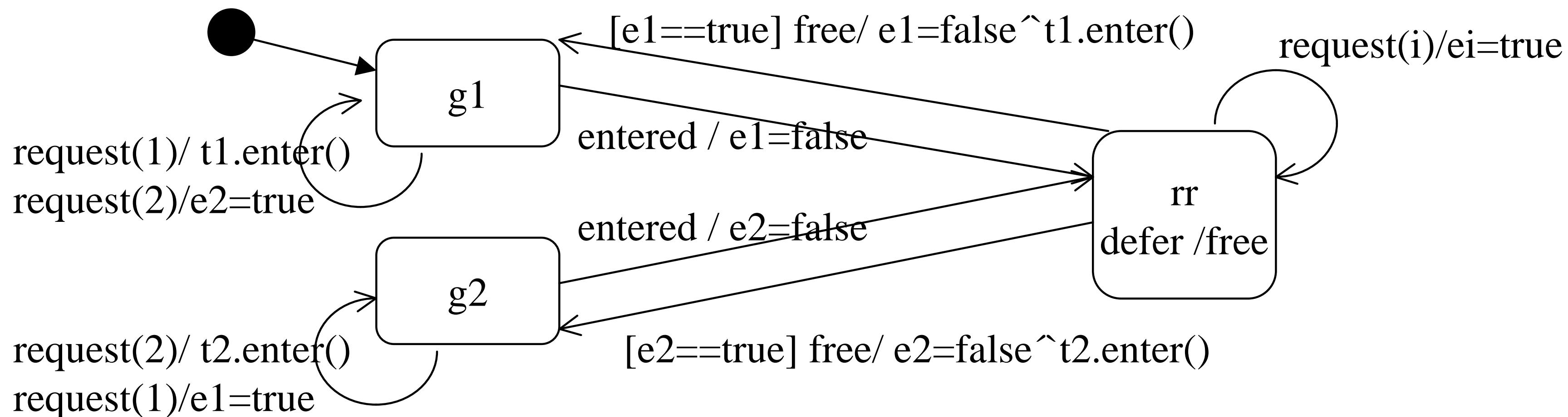
Klassendiagramm:
Zur Vereinfachung genau zwei Züge



Beispiel für Testfall:
Sequenzdiagramm

Beispiel Eisenbahnsteuerung: Statechart





Hugo/RT: Spezifikation von Eigenschaften

- Constraints in temporaler Erweiterung der OCL (CTL, LTL)
- Spezielle nützliche Prädikate:
  - o.inState(s): Ist Objekt o in Zustand s?
  - deadlock:

    Kein Fortschritt mehr möglich, aber kein Queue-Überlauf, kein Netzwerk-Überlauf, mindestens eine Zustandsmaschine nicht im Final state.
  - Boolesche Konnektoren

Mehr Informationen unter:

http://www.pst.ifi.lmu.de/projekte/hugo

Beispiel: Eigenschaften zur Modellprüfung

- Zwei Züge sind niemals gleichzeitig auf der Brücke?

```
AG not(t1.inState(onBridge) ∧ t2.inState(OnBridge))
```

- Ist das System ist verklemmungsfrei?

```
AG not deadlock;
```

- Kommen beide Züge irgendwann auf die Brücke?

```
EF t1.inState(OnBridge);
EF t2.inState(OnBridge);
```

Hugo/RT: Übersetzung

- Übersetzung von

  UML-Klassendiagrammen,

  UML-Zustandsmaschinen,

  UML-Interaktionen und

  OCL-Constraints

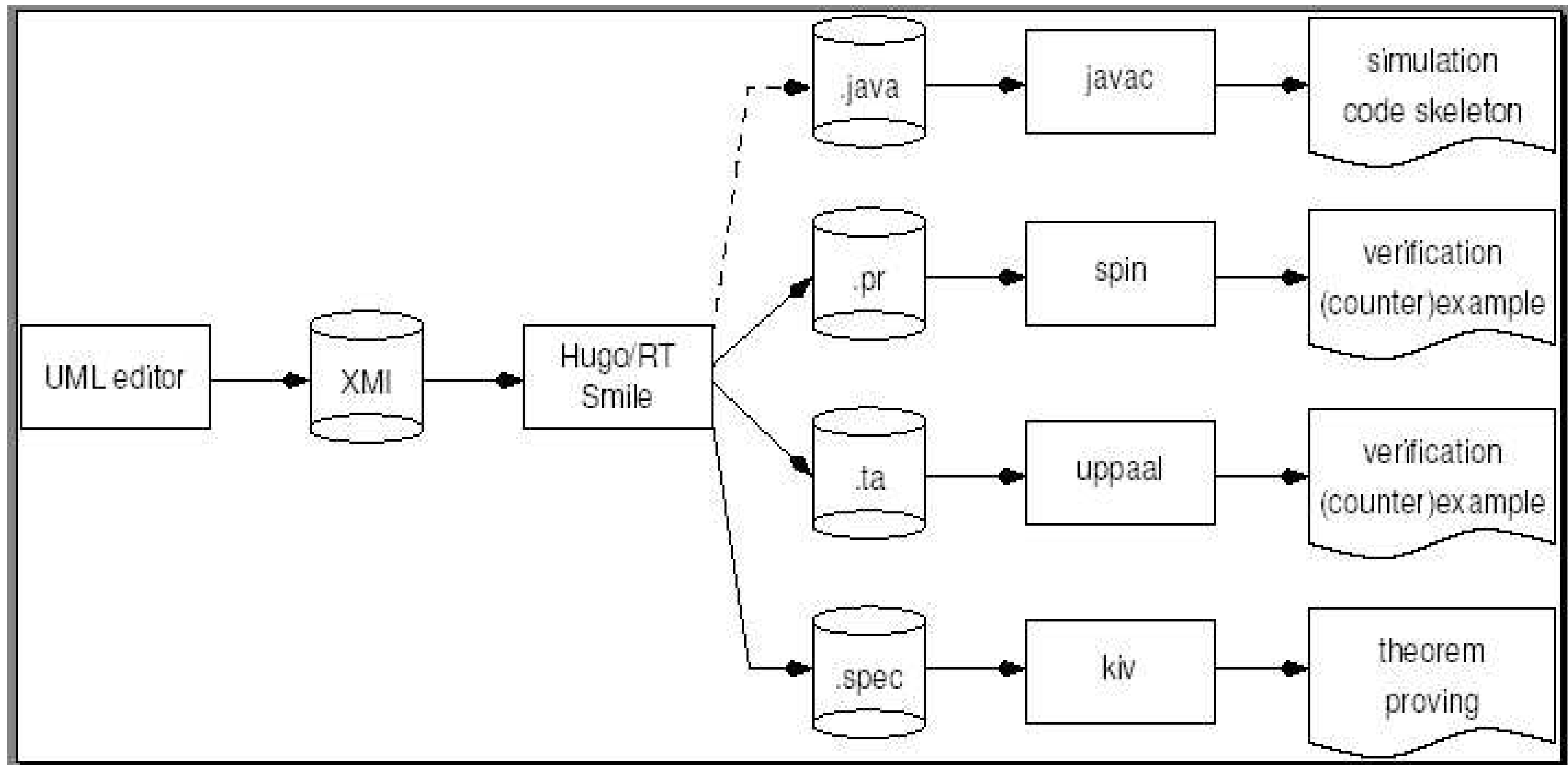
  nach

  Java (Programmiersprache), SPIN, UPPAAL (Model checker) und

  KIV (Theorembeweiser)

- UML-Erweiterungen
  - Assertions (lokale Zusicherungen)
  - Wait-Transitionen (nichtdeterministische Completion events)

Hugo/RT: Übersetzung



Testen objekt-orientierter Programme

- Neue Aspekte
  - Kapselung
  - Vererbung
  - Polymorphie
  - Dynamische Bindung
- (Bemerkung: Noch Forschungsbedarf)

Probleme beim Testen von OO Software (1)

Test von oo-Programmen ist schwierig:

- Der Zustandsraum **eines** Objekts ist meist klar abgegrenzt. Aber durch Beziehungen zu anderen Objekten (**Verzeigerung**) wird der Zustandsraum praktisch **schwer überschaubar**.
- Die Methoden einer Klasse enthalten in der Regel nur wenige Parameter. Die Klasse einer Methode ist wg. der Polymorphie häufig unklar.
- In den Methoden wird von anderen Objekten häufig Gebrauch gemacht. Wg. Polymorphie ist unklar, **welches** Objekt **welcher** Klasse aufgerufen wird.
- Für das Testen ist es wichtig, dass Spezifikationen des Verhaltens vorliegen. Ob dies geschieht, hängt vom Entwicklungsprozess ab und ist in der aktuellen Praxis der OO-SW-Entwicklung eher selten.

Probleme beim Testen von OO Software (2)

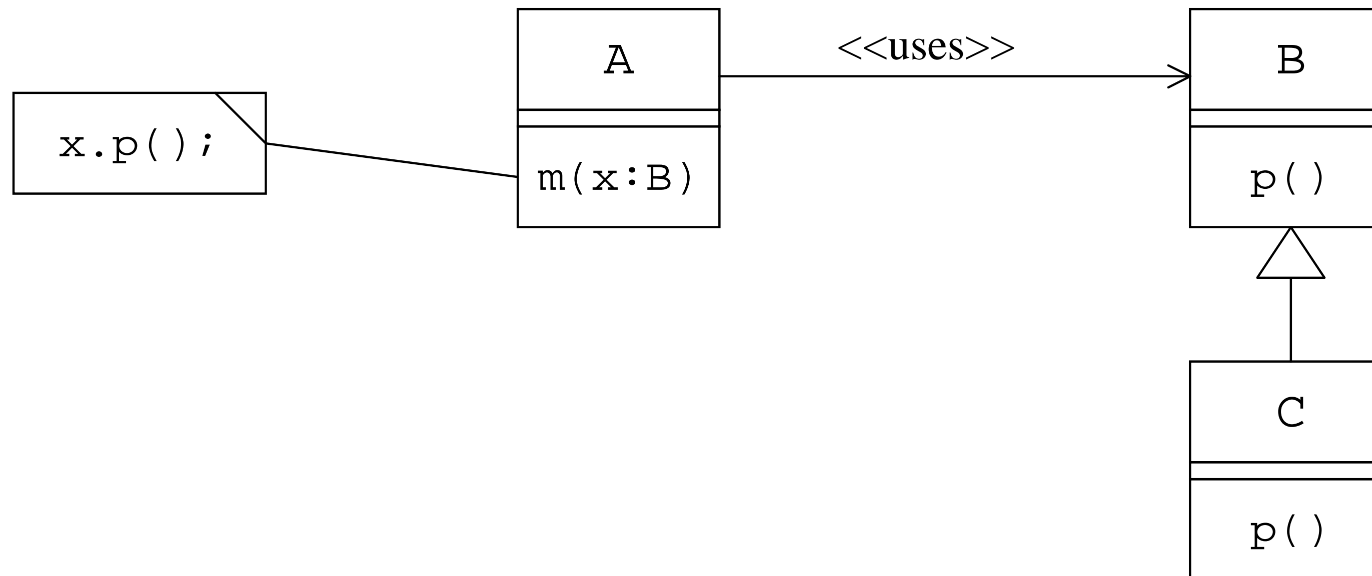
- Kapselung:

  Der interne Zustand von Objekten ist nicht direkt zugreifbar, sondern nur über öffentliche Operationen

  ⇨ interner Zustand kann nicht direkt getestet werden
- Vererbung:
  - Testfälle von Oberklassen müssen für Erbenklassen angepasst werden.
  - Vererbung definiert neuen Kontext für Methoden:

    Das Verhalten von Methoden kann sich ändern, da durch dynamische Bindung redefinierte Methoden aufgerufen werden können.

Beispiel



- Die Methode p wird in C redefiniert.
- Auch das Verhalten m wird dadurch geändert:
  Beim Aufruf o.m(c) mit c:C wird die redefinierte Variante von p aus C ausgeführt.

Strategie zum Testen objekt-orientierter Systeme

- Klassen bilden die kleinste zu testende Einheit.
- Objekte besitzen lokale Zustände und interagieren miteinander: Tests müssen dies berücksichtigen.

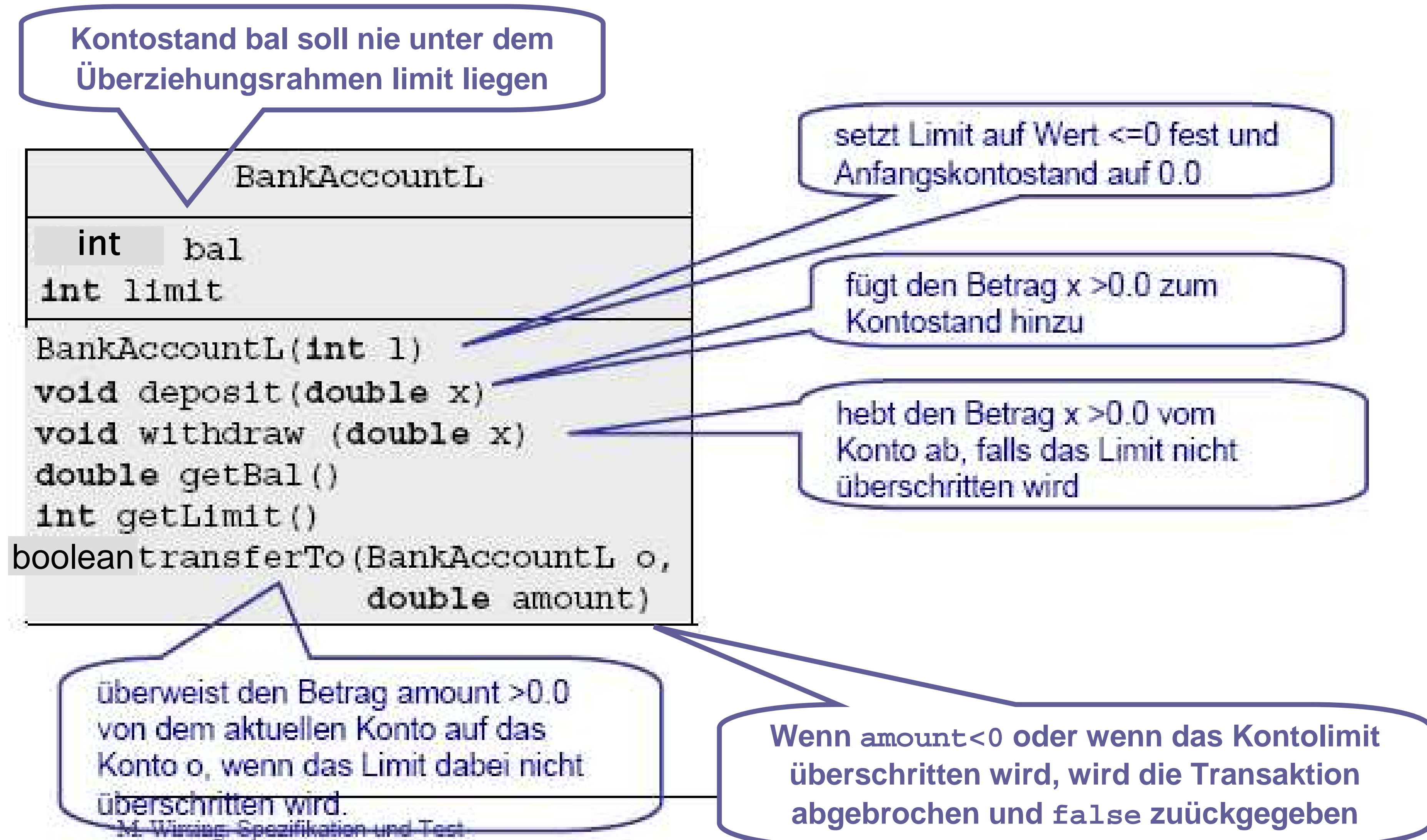
  ⇨ Das Testen von Klassen ist äquivalent zum Testen von Einheiten (Units):

  Die Operationen der Klasse werden getestet.

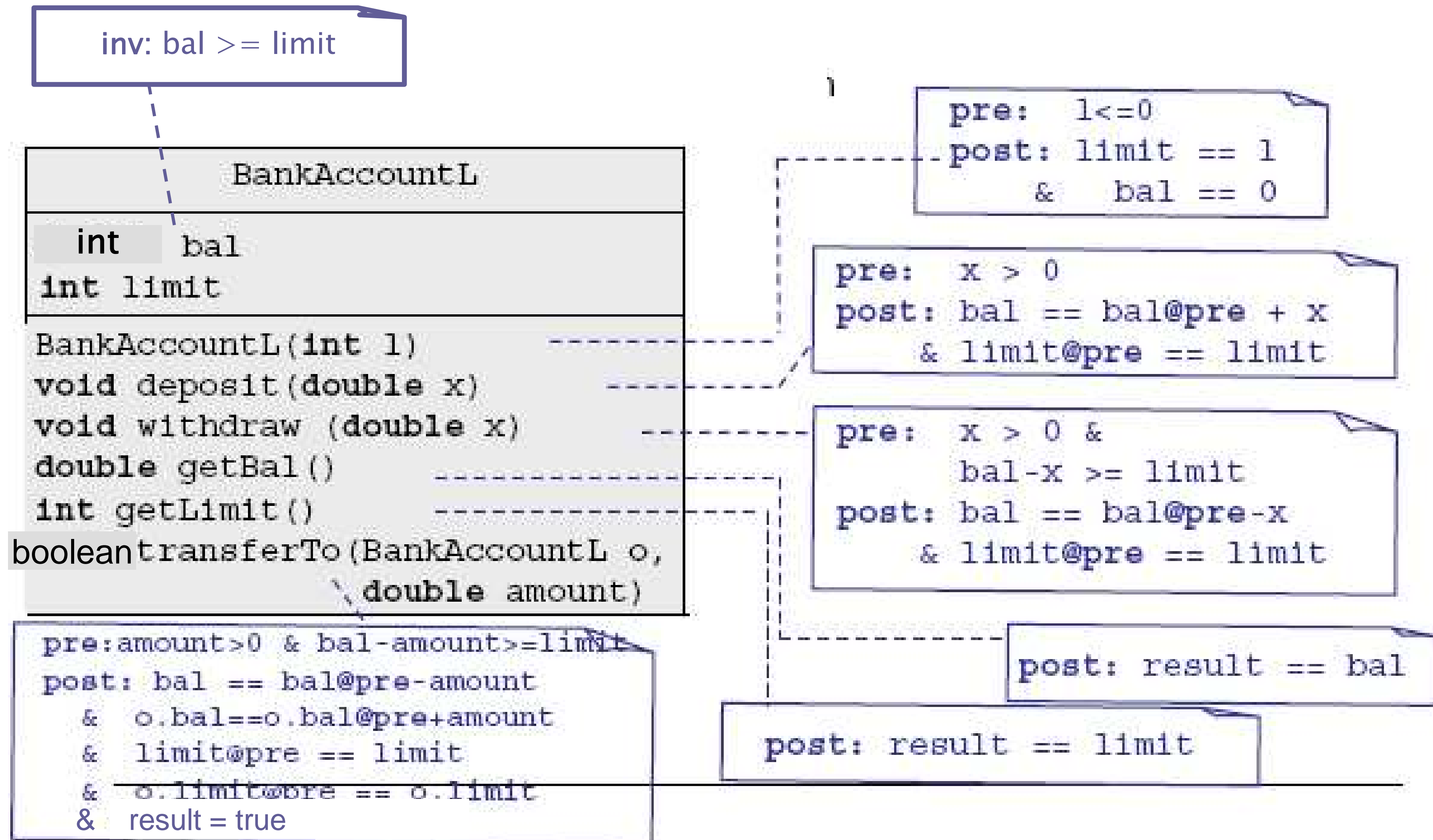
  Das Verhalten der Objekte wird zustandsabhängig getestet.
- Beim OO Testen werden vor allem die Zustände der Objekte und die Interaktionen zwischen Zuständen betrachtet.
- Wo immer möglich sollten Zusicherungen zur Spezifikation von Tests verwendet werden. Programme können mit ausführbaren Zusicherungen instrumentiert werden, die zur Testzeit automatisch überprüft werden.

  Diese Technik wurde u.a. zum Test der Microsoft Foundation Classes verwendet.

Beispiel: Bankkonto mit Überziehungsrahmen



Beispiel: Zusicherungen für das Bankkonto



Eine einfache Implementierung von Bankkonto (1)

```
public class BankAccount
{       private int balance;
        private int limit;

        public BankAccount(int l)
        {       balance = 0; limit = l;
        }

        public BankAccount(int initialBalance, int l)
        {       balance = initialBalance; limit = l;
        }

        public int getBalance()   {   return balance;       }

        public int getLimit()   {    return limit;    }

        public void deposit(int amount)
        {       assert amount > 0;
                balance = balance + amount;
        }
```

assert eingefügt zur Illustration der Instrumentierung durch Zusicherungen für Testzwecke; i.a. kein guter Prog.stil

Eine einfache Implementierung von Bankkonto (2)

```
. . .
   public void withdraw(int amount)
   {    if (!(amount > 0))
           throw new IllegalArgumentException("Negativer Betrag");
        if (!(balance - amount >= limit))
           throw new IllegalArgumentException(„Limit ueberschritten");
        balance = balance - amount;
        }

   public boolean transferTo(BankAccount other, int amount)
   {    try
        {   withdraw(amount);
            other.deposit(amount);
        } catch (IllegalArgumentException e)
        {   return false;
        }
        return true;
   }
}
```

Entwicklung von Testfällen aus der Spezifikation

Für jede Methode Äquivalenzklassen für gültige und ungültige Eingabewerte:

	Aufruf	Eingabebedingung	Resultat
T1	deposit b1.deposit(x)	x >0	b1.bal = b1.bal@pre + x
T2	depositNegAmount b1.deposit(x)	x <=0	Fehler
T3	withdraw b1.withdraw(x)	x >0 b1.bal-x >= limit	b1.bal = b1.bal@pre - x b1.bal >= limit
T4	withdrawNegAmount b1. withdraw(x)	x <=0	Fehler
T5	withdrawOverLimit b1. withdraw(x)	x >0 b1.bal-x < limit	Fehler
T6	transfer b1.transferTo(b2,x)	x >0 b1.bal-x>=limit	b1.bal = b1.bal@pre - x b2.bal = b2.bal@pre + x b2.bal >= limit ...
T7	transferNegAmount b1.transferTo(b2,x)	x <=0	false (Ausnahme soll in Implementierung abgefangen werden)

Entwicklung von Testfällen aus der Spezifikation

Durch Spezialisierung der Eingaben erhält man die Testfälle:

Für alle Testfälle definieren wir b1 und b2 mit Kontostand 100.

	Aufruf	x	Eingabebedingung	Resultat
T1	deposit b1.deposit(100)	100	x >0	b1.bal = 200
T2	depositNegAmount b1.deposit(x)	-100	x <=0	Fehler
T3	withdraw b1.withdraw(x)	50	x >0 b1.bal-x>=limit	b1.bal = 50 b1.bal >= limit
T4	withdrawNegAmount b1. withdraw(x)	-100	x <=0	Fehler
T5	withdrawOverLimit b1. withdraw(x)	150	x >0 b1.bal-x < limit	Fehler
T6	transfer b1.transferTo(b2,x)	50	x >0 b1.bal-x>=limit	b1.bal = 50 b2.bal = 150 ...
T7	transferNegAmount b1.transferTo(b2,x)	-30	x <=0	false (Ausnahme soll in Implementierung abgefangen werden)
..	...		...	...

Integrationstest

Beim Integrationstest werden Interaktionen zwischen Klassen getestet:

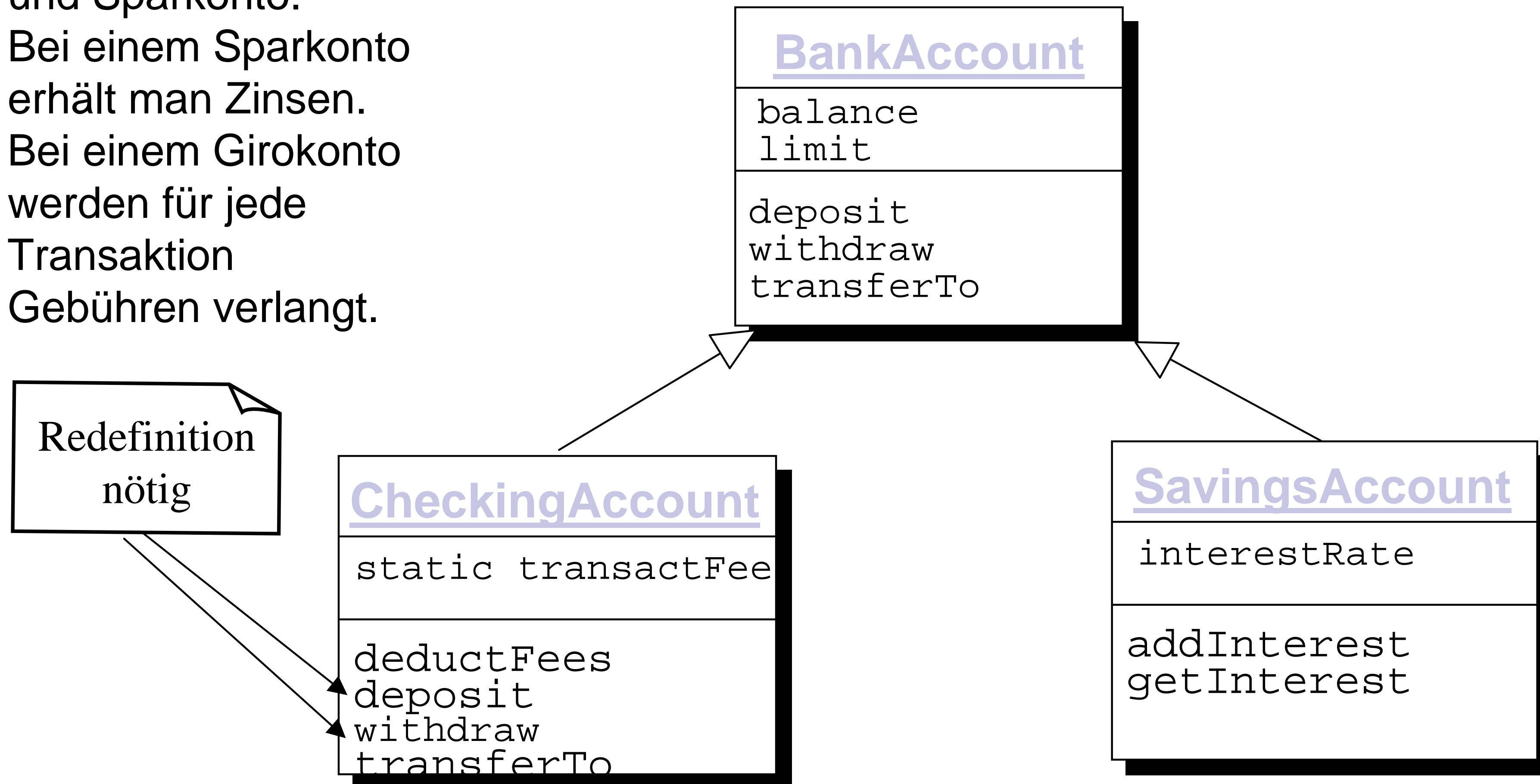
- Nutzungsbasiertes Testen
  - Beginne mit dem Testen/Integrieren unabhängiger Klassen (d.h. Klassen, die keine oder nur wenige andere Klassen benützen).
  - Teste die nächste Schicht von (abhängigen) Klassen, die die unabhängigen Klassen nutzen.
  - Wiederhole das Hinzufügen und Testen der nächsten Schicht von abhängigen Klassen, bis das gesamte System konstruiert ist.
- Thread-basiertes Testen
  - Integration der Menge von Klassen, die auf eine Eingabe oder ein Ereignis des Systems reagieren (müssen).

Wie testet man Vererbung?

- Testen von Vererbung ist Teil des Integrationstests
- In der Regel sind für die Erbenklasse die Testfälle der Oberklasse zu wiederholen (als Regressionstests).
- Neue Testfälle werden benötigt für Instanzen mit zusätzlichen Attributen, für neue Operationen und für redefinierte Operationen, sofern sich die Spezifikation geändert hat (falls sich nur die Implementierung geändert hat, genügen die "alten" Testfälle).
- Außerdem müssen die Vorbedingungen von redefinierten Operationen berücksichtigt werden:
  - Falls die Vorbedingung beim Erben spezieller ist, muss eine neue Zusicherung als Filter definiert werden, der diese Situation abfängt; dafür sind neue Testfälle zu entwickeln.
  - Falls die Vorbedingung beim Erben allgemeiner ist, genügt es, die alten Testfälle für den Erben zu wiederholen.

Beispiel Vererbung

Beispiel: Girokonto und Sparkonto.
Bei einem Sparkonto erhält man Zinsen.
Bei einem Girokonto werden für jede Transaktion Gebühren verlangt.



Testfälle für Unterklassen

- `SavingsAccount`

  Erbt die Methoden von `BankAccount` und hat zwei zusätzliche Methoden zur Behandlung von Zinsen.

  - Testfälle von BankAccount werden übernommen und neue Testfälle für `addInterest()` und `getInterest()` entwickelt.

- `CheckingAccount`

  redefiniert die Methoden und `BankAccount` und hat eine zusätzliche Methode zum Einziehen der Gebühren. Bei Überweisungen werden keine Gebühren vom Empfängerkonto verlangt.

  - Testfälle von `BankAccount` werden übernommen und neue Testfälle für `deductFee()` entwickelt.
  - Ausserdem muss auch `BankAccount` neu getestet werden; aufgrund der dynamischen Bindung wäre es möglich, dass bei einer Überweisung auf ein Girokonto beim Empfängerkonto berechnet werden. Dies ergibt neue Testfälle für `transferTo` (z.B. `a.transferTo(c ))` mit `a:BankAccount, c: CheckingAccount`)

Integrationstest der BankAccount-Hierarchie

- **1. Schritt**: Unit Test der Basisklasse

  Test von BankAccount wie vorher besprochen

- **2. Schritt**: Test von `SavingsAccount` und `CheckingAccount`

  Dies umfasst

  - nochmaliges Testen von `BankAccount` für `BankAccount`- Instanzen sowie
  - das Testen von `SavingsAccount` und `CheckingAccount` für Instanzen dieser Klassen sowie für Instanzen der Oberklasse `BankAccount` soweit anwendbar.

Zusammenfassung

- Modellprüfung von Zustandsautomaten unterstützt die analyse und das Testen von dynamischem Verhalten; HUGO/RT ist ein System zur Validierung von UML-Diagrammen, das Simulation,Modellprüfung und interaktives Beweisen unterstützt.
- Testen objekt-orientierter Programme ist schwierig, da Methoden stark interagieren und Kapselung und Vererbung neue Probleme aufwerfen.
- Unit-Tests beziehen sich auf einzelne Klassen, Integrationstests auf die Zusammenführung von Klassen. Insbesondere wird Vererbung im Rahmen von Integrationstests geprüft.

Literatur

- Johannes Link: Unit Tests mit Java. dpunkt.verlag, 2002.
- Robert V. Binder: Testing Object-Oriented Systems. Addison-Wesley 1999.

Appendix: Automatisierter Unit-Test mit JUnit

- JUnit ist ein „Open Source Framework“ zur Automatisierung von Unit-Tests für Java.
- Entwickelt (um 1998) von Kent Beck und Erich Gamma auf der Basis von SUnit (Beck,1994) zum Testen von Smalltalk-Programmen.
- Zum Herunterladen unter `http://download.sourceforge.net/junit/`

Entwicklung eines Testfalls in JUnit

Zur Entwicklung eines Testfalls geht man in folgenden Schritten vor:

1. Deklariere eine Unterklasse von TestCase.
2. Redefiniere die setUp() Methode, um die Testobjekte zu initialisieren .
3. Redefiniere die tearDown() Methode, um die Testobjekte zu löschen.
4. Deklariere eine oder mehrere public testXXX() Methoden, die die Testobjekte aufrufen und die erwarteten Resultate zusichern.
5. (Optional) Definiere eine main() Methode, die den Testfall startet. (Nicht nötig im Rahmen von Entwicklungsumgebungen wie Eclipse)

Beispiel: Einfacher Testfall für BankAccount

	Aufruf x	Eingabebedingung	Resultat
T1	deposit 100 b1.deposit(100)	x >=0	200

Außerdem sei b1 mit Kontostand 100 und Limit 0 gegeben.

Implementierung (mit Überprüfung der Invariante):

```
public void testDeposit()
  {     BankAccount b1 = new BankAccount(100);
        b1.deposit(100);
        assertEquals(200, b1.getBalance());
        assertTrue(b1.getBalance() >= b1.getLimit());
  }
```

Noch ein BankAccount-Testfall

	Aufruf	x	Eingabebedingung	Resultat
T3	withdraw b1.withdraw(x)	50	x >=0 b1.bal-x>=limit	b1.bal = 50 b1.bal >= limit

Außerdem sei b1 mit Kontostand 100 und Limit 0 gegeben.

Implementierung (mit Überprüfung der Invariante):

```
public void testWithdraw()
  {     BankAccount b1 = new BankAccount(100);

        b1.withdraw(50);
        assertEquals(50, b1.getBalance());
        assertTrue(b1.getBalance() >= b1.getLimit());
  }
```

Testklasse für BankAccount (1)

```
import junit.framework.*;
public class BankAccountTest extends TestCase
{ private BankAccount b1;
  private BankAccount b2;

  public BankAccountTest(String arg0)
  {      super(arg0);
  }
  public void setUp()
  {      b1 = new BankAccount(100, 0);
         b2 = new BankAccount(100, 0);
  }
  public void tearDown()        //ohne Effekt bei BankAccount,
  {      b1 = null;      //da immer neue Testobjekte erzeugt werden
         b2 = null;
  }
```

Testklasse für BankAccount (2)

Fortsetzung

```
  public static void main(String[] args)
  {     junit.swingui.TestRunner.run(BankAccountTest.class);
  }

}
```

Übersetzen und Ausführen von Tests

§ **Übersetzung der Testklasse:**

unter Windows:

```
javac -source 1.4 -classpath .;junit.jar
 accountInt\BankAccountTest.java
```

unter UNIX:

```
javac -source 1.4 -classpath .:junit.jar
 accountInt/BankAccountTest.java
```

§ **Ausführen der Testklasse**

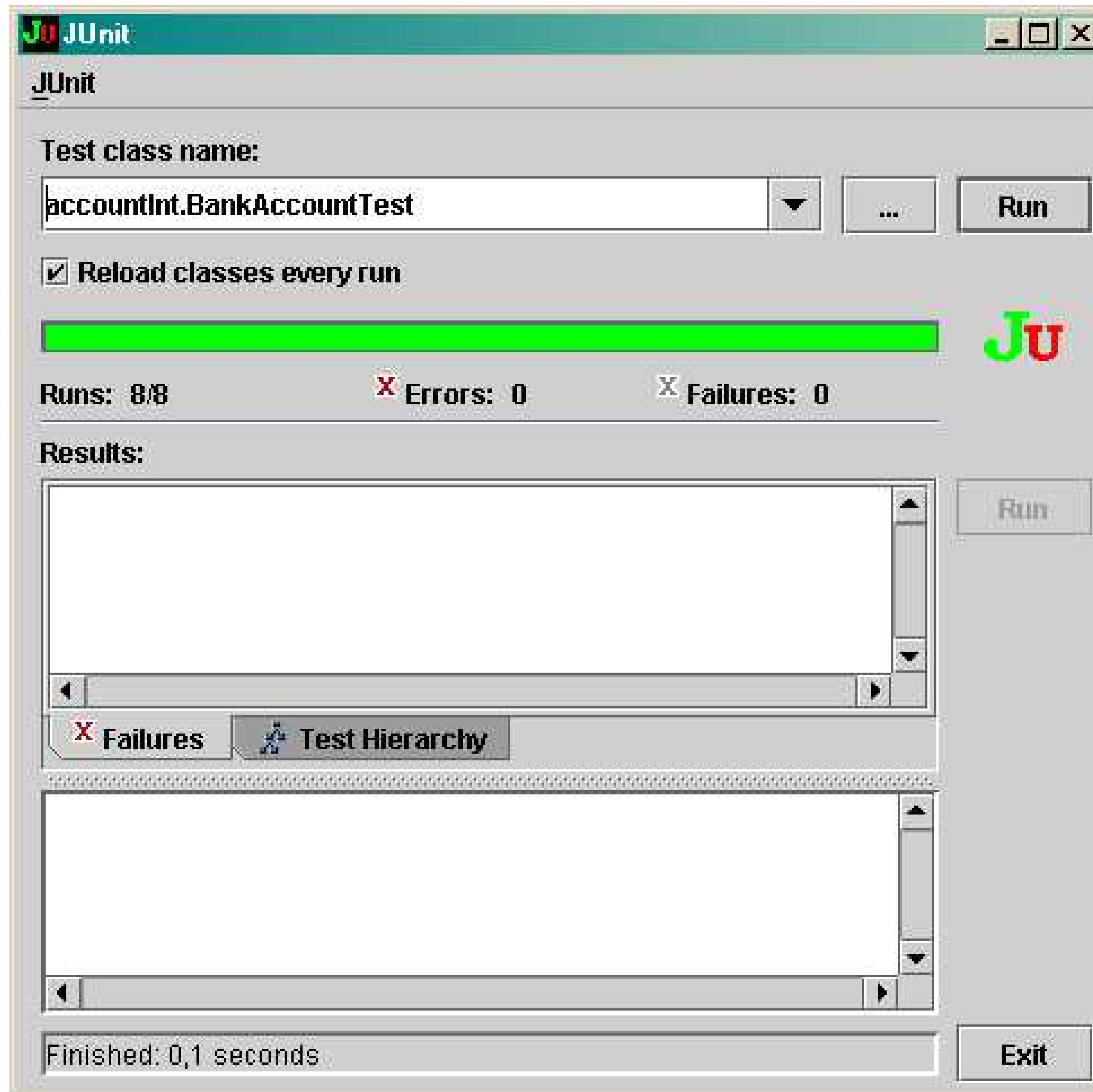
unter Windows:

```
>java -ea -classpath .;junit.jar accountInt.BankAccountTest
```

unter UNIX:

```
>java -ea -classpath .:junit.jar accountInt.BankAccountTest
```

Ausgabefenster von JUnit



- Erfolgreicher Testlauf von BankAccount: Kein Fehlgeschlagener Test gefunden!

Schema für eine Testklasse (1)

```
import junit.framework.*;

public class ZZZTest extends TestCase
  {   private ZZZ z;

  public ZZZTest(String name)
  {   super(name);
  }

  protected void setUp()
  {   z = new ZZZ();
  }

  protected void tearDown()
  {   z = null;
  }
```

Schema für eine Testklasse (2)

Fortsetzung:

```
public void testZZZMethod()
    {    assertTrue(<Boole'scher Wert>);
         assertEquals(expected, actual);
    }

public static void main(String args[])
    {    junit.swingui.TestRunner.run(ZZZTest.class);
    }
}
```

Testoperationen

Die Klasse **Assert** bietet folgende Methoden:

Zeigt keinen Fehler an, wenn gilt, dass

- `static void assertTrue(boolean b)` b wahr ist
- `static void assertFalse(boolean b)` b falsch ist
- `static void assertEquals(Object expected, Object actual)`

  expected und actual gleiche Werte haben (falls expected, actual prim. Daten)

  `expected.equals(actual) == true` (falls expected, actual Objekte)
- `static void assertEquals(double expected, double actual, double upTo)`

  `|expected - actual| <= upTo`
- `static void assertNotNull(Object actual)` `actual != null`
- ...

Testfixture

- Eine Fixture (Junit-Jargon) ist eine Menge von Objekten, die den gemeinsamen Ausgangszustand für die Testfälle einer Testklasse darstellt.
- Durch eine Testfixture vermeidet man Codeduplikation der gemeinsamen Testobjekte mehrerer Testmethoden einer Testklasse.
- Tests können die Objekte einer Testfixture gemeinsam benutzen, wobei jeder Test unterschiedliche Methoden aufrufen und unterschiedliche Resultate erwarten kann. Jeder Test einer Testklasse verwendet seine eigene Fixture, um die Tests von den Änderungen anderer Tests zu isolieren. Deshalb können die Tests einer Testklasse in jeder Reihenfolge ausgeführt werden.
- Eine Testfixture wird durch die setUp() Methode erzeugt; durch die tearDown() Methode werden diese Objekte wieder beseitigt. JUnit ruft die setUp-Methode automatisch vor jedem Test und die tearDown-Methode automatisch nach jedem Test auf.

Testschema für Ausnahmen

Ein Ausnahme im getesteten Code wird folgendermaßen in der Testklasse abgefangen und behandelt

```
public void testForException()
{ try
        {   YYY o = z.m();
            fail("Should raise an ZZZException");
        } catch (ZZZException success)
        { <assert über den Zustand vor dem Auslösen der Ausnahme
                in  z.m()>
        }
}
```

Signalisiert den Fehlschlag des Tests, d.h. dass KEINE Ausnahme ausgelöst wurde

Beispiel: Test auf überzogenes Kontolimit

	Aufruf	x	Eingabebedingung	Resultat
T5	withdrawOverLimit b1. withdraw(x)	150	x >=0 b1.bal-x < limit	Ausnahme

Eine nichterfüllte Vorbedingung von withdraw in BankAccount wird in BankAccountTest folgendermaßen abgefangen und behandelt:

```
public void testWithdrawOverLimit()
{       int balance = b1.getBalance();
        try
        {       b1.withdraw(150);
                fail("IllegalArgumentException expected");
        } catch (IllegalArgumentException e)
        {       assertEquals(balance, b1.getBalance());
        }
}
```

Signalisiert, dass KEINE Ausnahme ausgelöst wurde, obwohl das Kontolimit überzogen ist

Testen von behandelten Ausnahmen

	Aufruf	x	Eingabebedingung	Resultat
T7	transferNegAmount b1.transferTo(b2,x)	-30	x <0	false

Tests für Ausnahmen, die im getesteten Code abgefangen werden, benötigen keine spezielle Form (siehe BankAccountTest):

```
public void testTransferToNegativeAmount()
{     int balance1 = b1.getBalance();
      int balance2 = b2.getBalance();
      assertFalse(b1.transferTo(b2,-30));
      assertEquals(balance1,b1.getBalance());
      assertEquals(balance2,b2.getBalance());
}
```

Aufruf von transferTo führt zu Behandlung der Ausnahme in transferTo mit Resultat `false` . Siehe BankAccount

Testsuite

- Eine Testsuite ist ist eine Menge von Testfällen, die gemeinsam ausgeführt und betrachtet werden.
- Typischerweise testet man in einer Testsuite mehrere Klassen oder ein gesamtes Package.
- Wichtige Operationen der Klasse TestSuite:

  `TestSuite(ZZZTest.class)` Konstruktor konvertiert Testklasse in Testsuite

  `static Test suite()` gibt eine Instanz von TestSuite oder von TestCase zurück

  `addTestSuite(ZZZTest.class)` fügt eine Testklasse zu einer Suite hinzu

Testsuite für BankAccount und SavingsAccount

Die Klasse AllTests konstruiert eine Testsuite aus den Testklassen für BankAccount und SavingsAccount und führt alle diese Tests aus:

```
import junit.framework.*;
public class AllTests
{   public static Test suite()
    {    TestSuite suite = new TestSuite(BankAccountTest.class);
         suite.addTestSuite(SavingsAccountTest.class)
         return suite;
    }
    public static void main(String args[])
    {    junit.swingui.TestRunner.run(AllTests.class);
    }
}
```

Test-gesteuerter Entwurf

- Neue Software-Entwurfstechniken stellen das Testen vor das Implementieren des Programms:

  Exteme Programming, Test-first Programming, Agile Software Development

- Schritte des Test-gesteuerten Entwurfs:

  1. Erstelle UML-Diagramm
  2. Entwerfe einen Test für eine Methode
  3. Schreibe möglichst einfachen Code, bis der Test nicht mehr fehlschlägt
  4. Wiederhole 2. und 3. bis alle Methoden des Klassendiagramms implementiert sind.

Kurze Iterationen, in denen abwechselnd Code und Test geschrieben wird.

- Dabei wird häufig der Code (und manchmal der Test) restrukturiert („Refactoring“). Jedes Mal werden alle Tests durchgeführt, um sicher zu stellen, dass die Coderestrukturierung nicht zu Fehlern im „alten“ Code geführt hat.