Spezifikation und Test: Zusicherungen in Klassendiagrammen

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

05/03

Ziele

- Lernen in UML Klasseninvarianten und das Verhalten von Methoden zu spezifizieren
- Lernen Unit Tests zu schreiben und Testen und Debuggen mit Junit durchzuführen

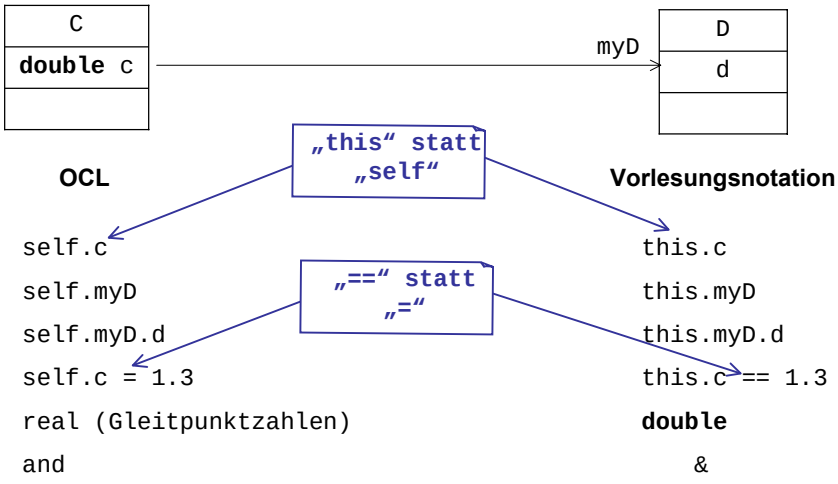
Spezifikation im UML

- Klassendiagramme legen nur die Signatur der Attribute und Methoden fest. Das reicht **NICHT** aus, um das Verhalten des Systems präzise und eindeutig zu spezifizieren.
- Die Sprache OCL („Object Constraint Language“) wird in UML verwendet, um Klasseninvarianten und Vor- und Nachbedingungen zu spezifizieren.
- OCL wurde 1997 von Jos Warmer et al. entwickelt.
- Wir benutzen eine Java-ähnliche Syntax für OCL.

OCL

OCL ist eine Teilsprache der Logik 1. Stufe mit Ausdrücken zur Bezeichnung von Instanzvariablen (und Zugriffspfaden auf Instanzvariablen) und vordefinierten Datentypen.

OCL vs Vorlesungsnotation



Spezifikation von Invarianten

Beispiel: Bankkonto mit Überziehungsrahmen

Textuell:

```
context   BankAccountL
inv: bal >= limit
```

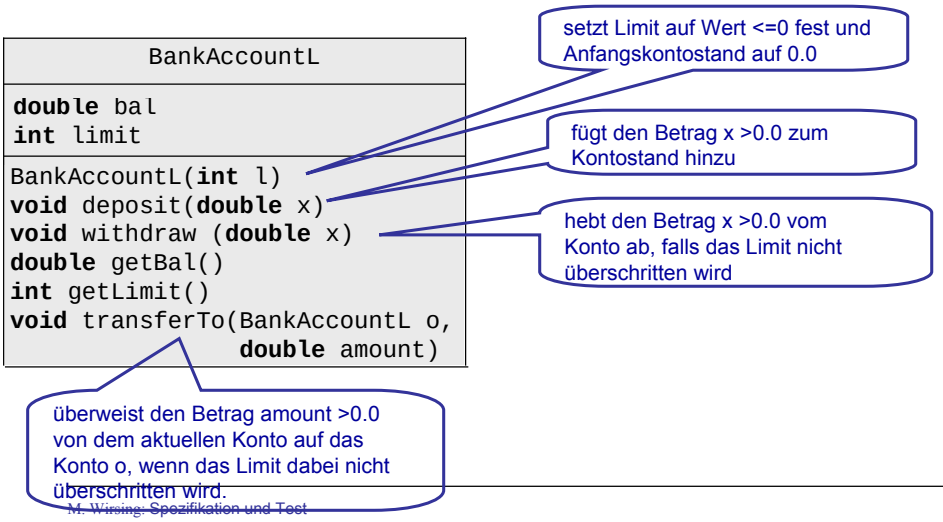
Spezifikation von Invarianten

- Eine Klasseninvariante I der Klasse C ist eine Eigenschaft der Attribute von C, die für jede Instanz von C in jedem Zustand gilt.
- Eine Klasseninvariante `I` wird ausgedrückt als Notiz (in geschweiften Klammern)

oder textuell in der Form

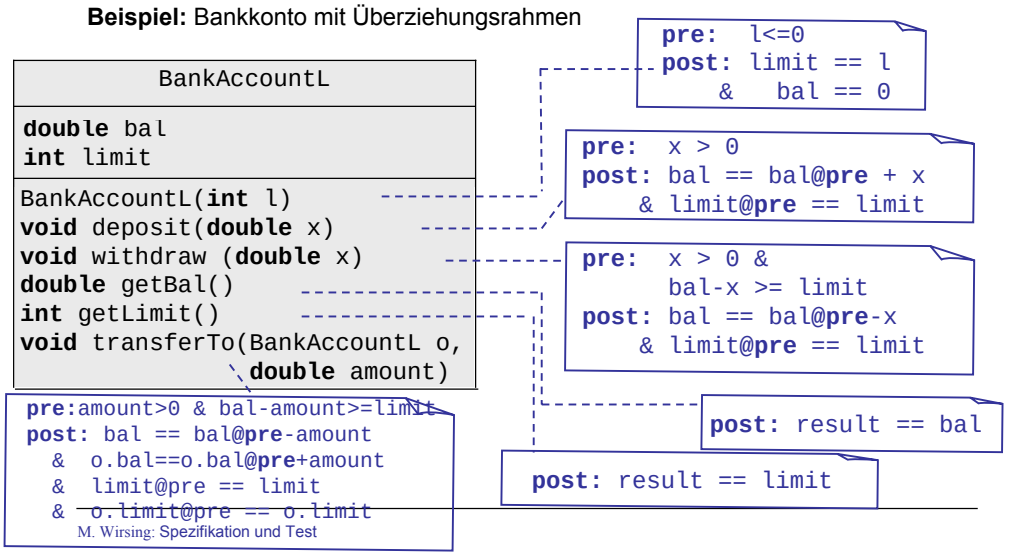
```
context C
inv: I
```


Beispiel: Bankkonto mit Überziehungsrahmen



Spezifikation von Vor- und Nachbedingungen

Beispiel: Bankkonto mit Überziehungsrahmen



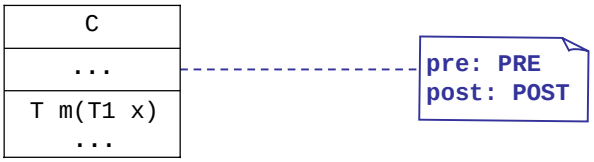
Spezifikation von Vor- und Nachbedingungen

```
context BankAccountL:: BankAccountL(int l)
pre:  l<=0
post: limit == l  &   bal == 0

context BankAccountL:: deposit(double x)
pre:  x > 0
post: bal == bal@pre + x  &  limit@pre == limit

context BankAccountL:: withdraw (double x)
pre:  x > 0  &  bal -x >= limit
post: bal == bal@pre-x  &  limit@pre == limit

context BankAccountL:: getBal()
post: result == bal

context BankAccountL:: getLimit()
post: result == limit

context BankAccountL:: transferTo (BankAccountL o, double amount)
pre: amount > 0 & bal-amount >= limit
post: bal==bal@pre-amount  & limit@pre==limit
  & o.bal==o.bal@pre+amount  & o.limit@pre==o.limit
```

Spezifikation von Vor- und Nachbedingungen

Vor und Nachbedingungen von Methoden werden ausgedrückt

- als Notiz mit Schlüsselwörtern ***pre, post***:



wobei PRE und POST OCL-Formeln sind

- oder textuell

```
context C :: T m(T1 x)
pre: PRE
post: POST
```


Spezifikation von Vor- und Nachbedingungen

- PRE und POST dürfen als Variablen nur enthalten `self`, den formalen Parameter `x` und die Attribute von `C`;
- In POST kommt außerdem vor

`a@pre` (für jedes Attribut a von `C`)

`result` (zur Bezeichnung des Resultats)

Wert von a im Vorzustand (`a@pre` ersetzt den Gebrauch der logischen Variablen)

Vordefinierte lokale Variable zur Bezeichnung des Resultats

Partielle Korrektheit

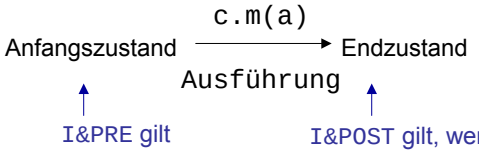
- Eine Methode der Klasse C

```
T m(T1 x) {body}
```

heißt **partiell korrekt** bzgl. der Vorbedingung PRE, der Nachbedingung POST und der Invariante I, wenn

m partiell korrekt bzgl. PRE und POST ist und die Invariante I erhält;

d.h. für **alle** Instanzen c von C und **alle** aktuellen Parameter a,



wenn I&PRE im Anfangszustand von c.m(a) gilt und,

wenn c.m(a) terminiert, dann gilt I&POST nach Ausführung von c.m(a)

Totale Korrektheit

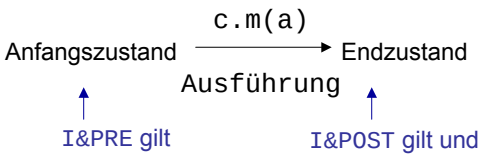
- Eine Methode der Klasse C

```
T m(T1 x) {body}
```

heißt **total korrekt** bzgl. PRE, POST und der Invariante I,
wenn

m total korrekt bzgl. PRE und POST ist und die Invariante I erhält;

wenn für alle Instanzen c von C und **alle** aktuellen Parameter a,



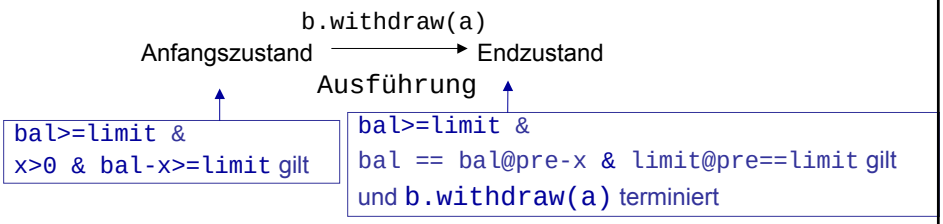
wenn I&PRE im Anfangszustand von c.m(a) gilt,

dann terminiert c.m(a) und I&POST gilt nach Ausführung von c.m(a)

Beispiel Totale Korrektheit

- Die Methode `withdraw` der Klasse `BankAccountL` ist total korrekt bzgl
  - der Invariante `bal >= limit`
  - der Vorbedingung `x > 0 & bal -x >= limit`
  - der Nachbedingung `bal == bal@pre-x & limit@pre == limit`

d.h. für **alle** Instanzen `b` von `BankAccountL` und **alle** aktuellen Parameter `a` gilt:



Korrekte Implementierung

- Eine Javaklasse `C` heißt **partiell / total korrekte Implementierung** einer UML Spezifikation von `C`, wenn
  alle Methoden von `C` partiell / total korrekt bzgl. ihrer Vor- und Nachbedingung, der Invarianten von C und
  der Invarianten der Klassen der formalen Parameter sind.

Bemerkung:
Dabei wird vorausgesetzt, dass durch Methodenaufrufe nur die explizit in Vor- und Nachbedingung erwähnten Attribute geändert werden.

Korrekte Implementierung von `BankAccountL`: 1. Möglichkeit

Die folgende Implementierung von `BankAccount` ist total korrekt bzgl. der UML Spezifikation (aber problematisch beim Produktionslauf, siehe später):

```
public class BankAccountL
 { private double bal;
   private int limit;

   public BankAccount(int l)
        {       assert l<=0; bal = 0.0; limit = l;
        }
   public void deposit(double amount)
   {        assert amount>0;
            bal = bal + amount;
   }
   public void withdraw(double amount)
   {        assert (amount>0 & bal-amount >= limit);
            bal = bal - amount;
   }
```

setzt Limit auf Wert <=0 fest und Anfangskontostand auf 0.0

Problematisch, da nicht geprüft bei Produktionslauf

fügt den Betrag amount >0.0 zum Kontostand hinzu

hebt den Betrag amount >0.0 vom Konto ab, falls das Limit nicht überschritten wird

Korrekte Implementierung von Spezifikationen

Da bei der Auslieferung (Produktionslauf) eines Programms die Zusicherungen ausgeblendet werden, ist es besser,

Vorbedingungen
durch kontrollierte Ausnahmen

zu prüfen.

Das entspricht dem Grundsatz der **defensiven Programmierung!**

Beispiel `BankAccountL`:

```
public void transferTo(BankAccount other, double amount)
{       if (!(amount>0)) throw new
              IllegalArgumentException(„amount ist nicht positiv");
        if (!(bal-amount >= limit)) throw new
              IllegalArgumentException(„Kontolimit ueberschritten");
        if (!(other.getBalance()>= other.getLimit())) throw new
             IllegalArgumentException(other + „verletzt Invariante");
        withdraw(amount);
        other.deposit(amount);
}
```

Ausnahme bei Verletzung der Vorbedingung amount >0.

Ausnahme bei Verletzung der Vorbedingung bal-amount >= limit.

Ausnahme bei Verletzung der Invariante für other .

Korrekte Implementierung von `BankAccountL` mit Ausnahmen

Eine defensive Implementierung von `BankAccountL` mit Ausnahmen:

```
public class BankAccountL
 { private double bal;
   private int limit;

   public BankAccount(int l)
   {    if (l>0) throw new
            IllegalArgumentException(„limit ist nicht negativ“);
        bal = 0.0; limit = l;
   }

    public void deposit(double amount)
    {      if(amount<=0) throw new
             IllegalArgumentException(„amount ist nicht positiv“);
           bal = bal + amount;
    }
```

Ausnahme, falls limit nicht negativ

Ausnahme, falls amount nicht positiv

Nachweis der Korrektheit

Beispiel `BankAccountL`

Zum Beweis der totalen Korrektheit von `BankAccountL` ist zu zeigen, dass

1. alle Methoden von `BankAccountL` terminieren und dass
2. unter der Annahme der Invariante `bal>=limit,` der Invarianten der Klassen der formalen Parameter und der Vorbedingung
   nach Ausführung jeder Methode
   die Nachbedingung, die Invariante `bal>=limit` und die Invarianten der Klassen der formalen Parameter gelten.

Nachweis der Korrektheit

- Zum **Beweis der Korrektheit** kann man den **Hoare-Kalkül** verwenden (mit zusätzlichen Regeln für Instanzvariablen, **new**, Methodenaufruf)

⇒ Dies werden wir hier **nicht** untersuchen.

Siehe Vorlesung „Formale Objekt-orientierte Software-Entwicklung“

- Unabhängig von dem Beweis der Korrektheit sind **Spezifikationen** von Invarianten, Vor- und Nachbedingungen aber **wichtige Hilfsmittel**, die es erlauben, die **Qualität von Java-Programmen zu verbessern**.

Test der Korrektheit

- Man kann die Sprache JML verwenden, die es ermöglicht,

Invarianten, Vor- und Nachbedingungen als formale Kommentare

zum Programm hinzuzufügen und **automatisch zu testen**.

- In Java können wir **Assertions** verwenden, um die **Gültigkeit der Invarianten und Nachbedingungen zur Laufzeit zu überprüfen.**

! Beide Möglichkeiten sind aber **keine Beweise**, sondern nur Tests an endlich vielen Beispielen.

Nachbedingunengen als „asserts"

- Um Nachbedingungen als „asserts" darstellen zu können, müssen die @pre-Variablen durch logische Variablen ersetzt werden:

  Für jedes `c.a@pre` vom Typ `C` in einer Nachbedingung wird eine logische Variable `A` mittels

  ```
  final C A = c.a;
  ```

  als (neue) lokale Konstante eingeführt und mit dem Wert von `c.a` im Vorzustand initialisiert.

Korrektheitszusicherungen in Java

Beispiel `transferTo`

```
public void transferTo(BankAccount other, double amount)
{       final double THISBAL = bal;
        final double OTHERBAL = other.getBalance();
        final int THISLIMIT = limit;
        final int OTHERLIMIT = other.getLimit();

        if (!(amount>0)) throw new
        IllegalArgumentException(„amount ist nicht positiv");
        if (!(bal-amount >= limit)) throw new
       IllegalArgumentException(„Kontolimit ueberschritten");
        if (!(other.getBalance()>= other.getLimit())) throw new
        IllegalArgumentException(other + „verletzt Invariante");

        withdraw(amount);
        other.deposit(amount);
```

Logische Variablen zur Kodierung der „@pre-Werte"

Vorbedingungen von transferTo

Invarianten für other; Invar. für this schon geprüft in Vor-bedingung.

Korrektheitszusicherungen in Java

Fortsetzung `transferTo`:

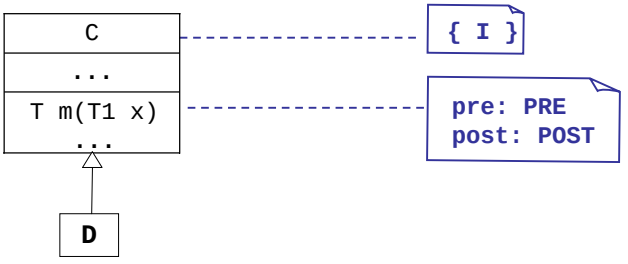
```
            assert (bal>=limit &
                   other.getBalance() >= other.getLimit());
            assert (bal == THISBAL-amount &
                   other.getBalance() == OTHERBAL+amount &
                   limit == THISLIMIT &
                   other.getLimit()== OTHERLIMIT);
    }
```

Invariante für this und other

Nachbedingung von transferTo

Vererbung und Spezifikation

Die Vererbungsbeziehung erhält auch die spezifischen Eigenschaften einer Oberklasse. Man nennt dies „behavioral subtyping“.



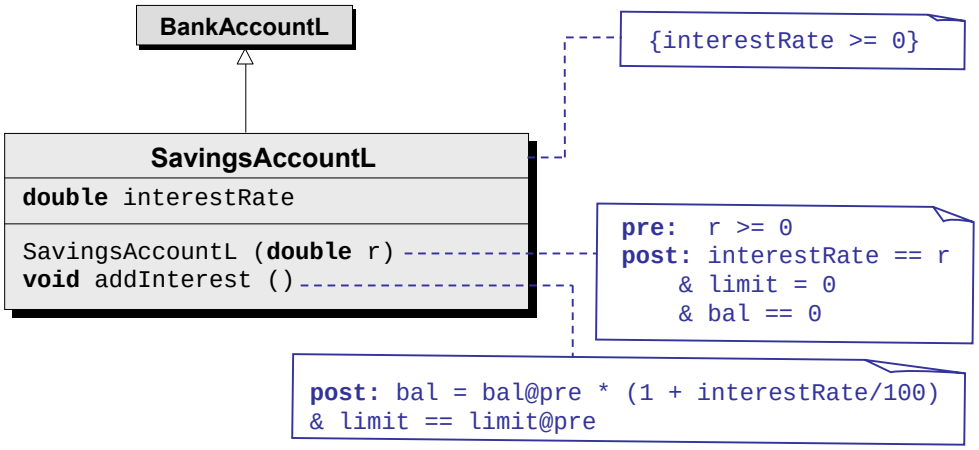
Ist `D` Erbe von `C`, so gilt

- die Invariante `I` von `C` gilt auch für **`D`**
- jede Methode `T m(T1 x)` von `C` vererbt die Vor- und Nachbedingungen an `D`

```
context D :: T m(T1 x)
pre: PRE
post: POST
```

Vererbung und Spezifikation

Beispiel: Sparkonto



Vererbung von Invarianten und Vor-/Nachbedingungen

Beispiel: Sparkonto

- Ein Sparkonto erfüllt alle eigenen Invarianten, Vor- und Nachbedingungen und die der Oberklasse `BankAccountL` z.B.

- Die Invariante von `SavingsAccountL` lautet

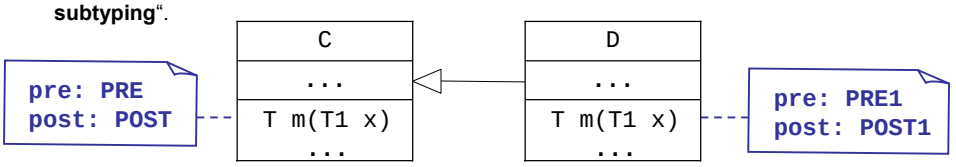
```
context SavingsAccountL
inv: bal >= limit & interestRate >= 0
```

- Die ererbte Vor- und Nachbedingung von `deposit` lautet

```
context SavingsAccountL:: deposit(double x)
    pre:  x > 0
    post: bal == bal@pre + x  &  limit@pre == limit
```

Spezifikation von Redefinition: „behavioral subtyping"

Wird eine vererbte Methode m in D redefiniert, so muss die Erbenmethode das Verhalten der „Vatermethode" spezialisieren und an jeder Stelle aufgerufen werden können, an der die „Vatermethode" aufgerufen werden kann („Substitutionsprinzip"). Man nennt dies „**behavioral subtyping**".



Aufgrund des Substitutionsprinzips muss gelten:

- PRE implies PRE1
- POST1 implies POST

Spezifikation von Redefinition: Substitutionsprinzip

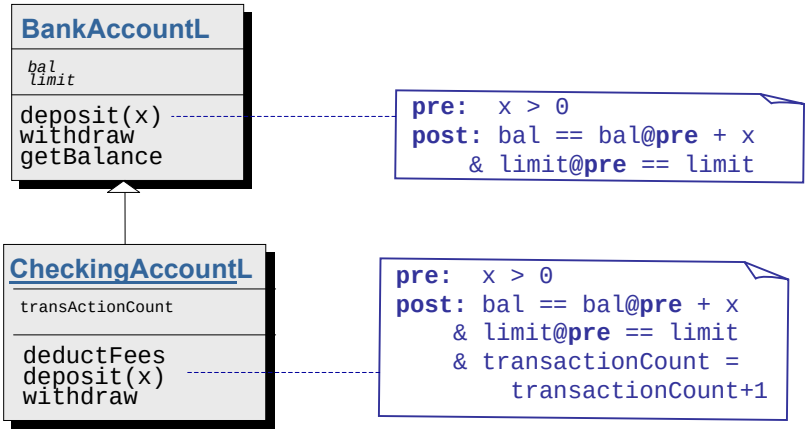
- Das „Substitutionsprinzip" bei Vererbung verlangt, dass eine Erbenmethode an jeder Stelle aufgerufen werden kann, an der die „Vatermethode" aufgerufen werden kann. Deshalb muss die Vorbedingung PRE1 der Erbenmethode schwächer sein als die Vorbedingung PRE der Vatermethode:

  PRE implies PRE1

- Die Nachbedingung POST1der Erbenmethode muss spezieller (stärker) sein als die Nachbedingung POST der Vatermethode:

  POST1 implies POST

Spezifikation von Redefinition: Beispiel CheckingAccount



Es gilt: Pre(BankAccountL) implies Pre(CheckingAccountL) und Post(CheckingAccountL) implies Post(BankAccountL)

Zusammenfassung

- In UML werden Zusicherungen in der Sprache OCL spezifiziert. In der Vorlesung verwenden wir aber eine Java-artige Variante von OCL.
- Ein Invariante beschreibt eine Eigenschaft der Attribute (und Assoziationen). Die Invarianten sollen vor und nach jedem Aufruf einer (public) Methode gelten.
- Vor- und Nachbedingungen spezifizieren das Verhalten von Methoden.
- Erbenklassen erhalten das spezifizierte Verhalten der Oberklasse („**behavioral subtyping**“): Invarianten und Vor- und Nachbedingungen werden vererbt. Auch bei Redefinition gilt das **Substitutionsprinzip**, aus dem folgt: Die Vorbedingung der Vaterklasse impliziert die Vorbedingung des Erben, die Nachbedingung des Erben impliziert die Nachbedingung der Vaterklasse.

Zusammenfassung

- Eine Java-Implementierung einer Methode ist partiell/total korrekt bzgl. ihrer Spezifikation in UML, wenn sie partiell/total korrekt ist bzgl der Vor- und Nachbedingungen und wenn sie alle Invarianten erhält.
- OCL-Vorbedingunngen und Klasseninvariantenwerden in Java durch kontrollierte Ausnahmen implementiert; interne Invarianten (z.B. Schleifeninvarianten) und Nachbedingungen werden mittels „assert“ implementiert und getestet.