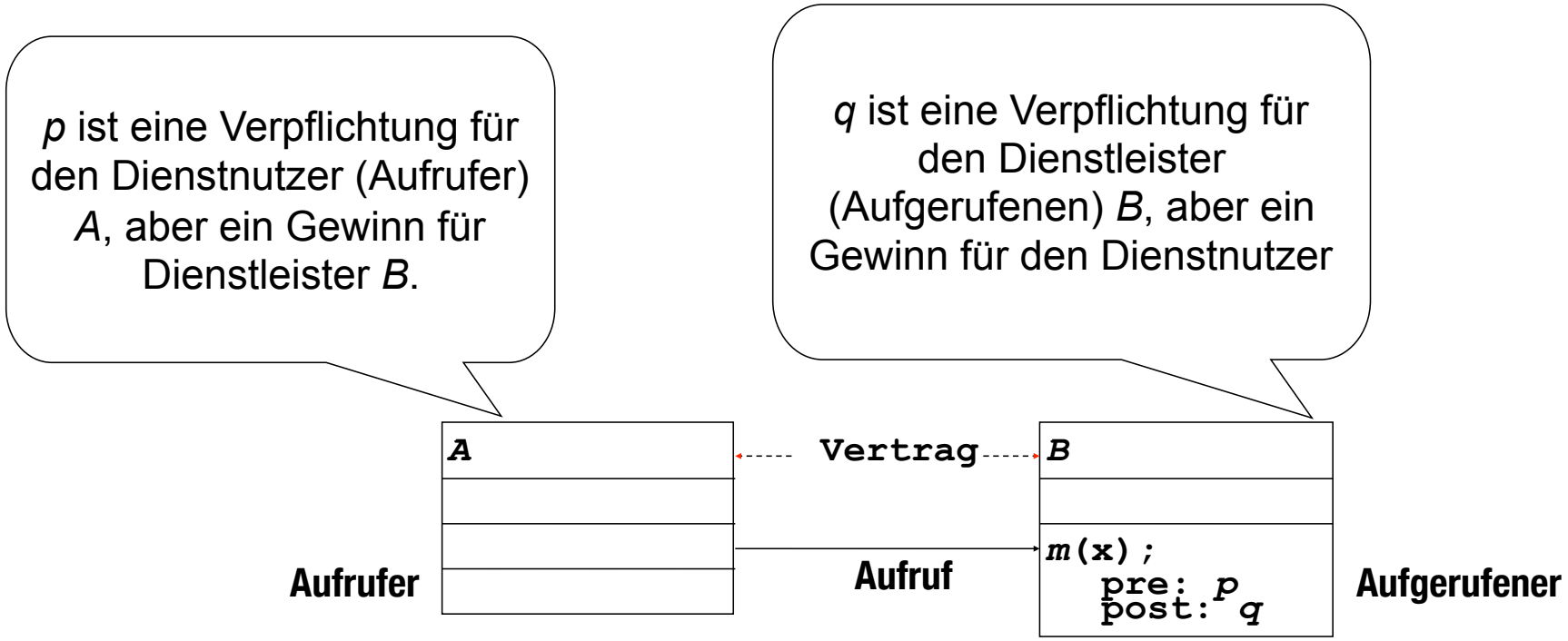
Objektorientierte Verträge

OOPM, Ralf Lämmel

Verallgemeinerung von Vor- und Nachbedingungen für imperative Programme: OO-Verträge zwischen Dienstleister & -nutzer



Klasse (Datentyp) *B* sagt zu *A* (und allen anderen Anwendern): "Wenn Du *m* aufrufst so dass *p* gilt, erbringe ich einen finalen Zustand, in welchem *q* gilt."

Vor- und Nachbedingungen
für Methoden

Push-Operation eines Kellers mit “Vertrag”

```
public void push(int item) {
   // Housekeeping for state
   int oldSize = size();

   // Precondition
   assert true;

  ... IMPLEMENTATION ELIDED ...

   // Postcondition
   assert    !isEmpty()
        &&   top() == item
        &&   size() - 1 == oldSize;
}
```

Nachbedingungen sind Bedingungen die nach der Methodenausführung gelten sollen. Nachbedingungen müssen immer durch den Methodenkörper erbracht werden---die zugehörige Vorbedingung zu Beginn vorausgesetzt.

Pop-Operation eines Kellers mit “Vertrag”

```
public void pop() {
   // Housekeeping for state
   int oldSize = size();

   // Precondition
   assert !isEmpty();

   ... IMPLEMENTATION ELIDED ...

   // Postcondition
   assert size() + 1 == oldSize;
}
```

Vorbedingungen beschreiben, wann eine Methode anwendbar ist und sinnvoll funktioniert. Vorbedingungen sind verbindlich für alle Methodenaufrufe.

Sichtbare Größen in Vor- und Nachbedingungen

* Argumente der Methoden
* Lesender Zugriff auf öffentlichen Zustand (Getter)
* Andere nichtmutierende (öffentliche) Methoden
* Vorbedingungen sehen nur initialen Zustand.
* Nachbedingungen sehen (auch) resultierenden Zustand.

In Java muss man den initialen Zustand für die Nachbedingungen sichern.

Klassen-Invarianten

Invarianten sind Bedingungen, die
“immer” gelten sollen.

Beispiele für Invarianten

- *Datentyp **Konto***
  - Der Kontostand ist immer nicht-negativ.
- *Datentyp **FloatRange***
  - Untere und obere Schranke sind verträglich.

Ein Konto mit Invariante

```
public class Account {
   private float balance;
   private void invariant() {
      assert balance >= 0;
   }
   // Return balance in account
   public float getBalance() {
      invariant();
      return balance;
   }
  ...
}
```

Programmierung der Invariante als (private) Methode

Überprüfung der Invariante

Einzahlen

```
public void deposit(float amount) {
    invariant();
    // Precondition
    assert amount >= 0;
    // Housekeeping
    float oldBalance = balance;


    balance += amount;


    invariant();
    // Postcondition
    assert balance >= oldBalance
        && oldBalance + amount == balance;
}
```

Brauchen wir diese Überprüfung wirklich?

Überprüfung der Invariante

Auszahlen

```
public void withdraw(float amount) {
   // Precondition
   assert amount >= 0 && balance >= amount;
   // Housekeeping
   float oldBalance = balance;

   balance -= amount;

   invariant();
   // Postcondition
   assert balance <= oldBalance
       && oldBalance - amount == balance;
}
```

Überprüfung der Invariante

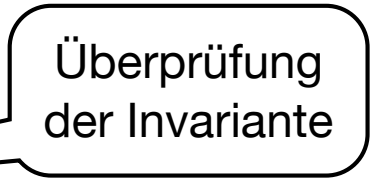
Die Attribute der Klasse sind privat. Damit können nur lokale Methoden die Attribute ändern. Damit genügt das Überprüfen der Invariante am Ende einer jeden Methode.

Ein Datentyp FloatRange I/III

```
public class FloatRange {
   private Float min, max;
   private void invariant() {
     assert this.min <= this.max;
   }
   public FloatRange(Float min, Float max) {
      // Precondition
      assert min < max;

      this.min = min;
      this.max = max;

      invariant();
      // Postcondition
      assert getMin() == min && getMax() == max;
   }
  ...
}
```



Ein Datentyp FloatRange II/III

```
public Float getMin() {
   return min;
}
public void setMin(Float min) {
   // Precondition
   assert min <= getMax();

   this.min = min;

   invariant();
   // Postcondition
   assert getMin() == min;
}
```

Überprüfung der Invariante

Ein Datentyp FloatRange III/III

```
public Float getMax() {
   return max;
}
public void setMax(Float max) {
   // Precondition
   assert getMin() <= max;

   this.max = max;

   invariant();
   // Postcondition
   assert getMax() == max;
}
```

Überprüfung der Invariante

Eine Klasseninvariante gilt für alle Instanzen der Klasse.

Alle Methoden einer Klasse müssen deren Invariante einhalten.

- Verlangte Gültigkeit von Vorbedingungen
  - Zu Beginn von Methoden
- Verlangte Gültigkeit von Nachbedingungen
  - Zum Ende von Methoden
- **Verlangte Gültigkeit von Invarianten:**
  - **Am Ende von Konstruktoren**
  - **Zu Beginn und zum Ende von Methoden**

Behandlung von Verträgen in Java
Zusammenfassung

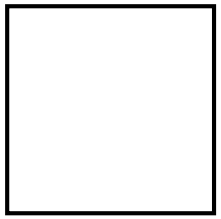
- Vorbedingungen:
  - assert bei Methodeneintritt
- Nachbedingungen:
  - assert bei Methodenaustritt
- **Sicherung von initialem Zustand in Variablen**
- Invarianten:
  - Extra “private” Methode mit assert
  - Invariante wird von allen Methoden aufgerufen:
    - vor/nach dem Test der Nachbedingung und
    - (eventuell) vor/nach dem Test der Vorbedingung.

Zu teuer für “Produktion” aber gut für Debugging.

“Verfolgungswahn”

Vererbung und Verträge

- Ein Szenario:
  - Gegeben seien Klassen *Rectangle* und *Square*.
  - Wir könnten *Square* als Unterklasse von *Rectangle* betrachten.
  - Eine Instanz von *Square* hat gleiche Seitenlängen.
- Wie verhält sich dieses Szenario mit Vererbung bezüglich von Verträgen?
- Könnten wir auch *Rectangle* als Unterklasse von *Square* betrachten?



Rechtecke ohne Verträge

```
public class Rectangle {
   private int width;
   private int height;
   public Rectangle(int width, int heigth) {
      setWidth(width);
      setHeight(height);
   }
   public int getWidth() { return width; }
   public int getHeight() { return height; }
   public void setWidth(int width) { this.width = width; }
   public void setHeight(int height) { this.height = height; }
   public int getPerimeter() { return (width+height)*2; }
}
```

Was ist ein sinnvoller Vertrag für Rechtecke?

Rechtecke mit Vertrag I/II

```
package oo.dbc.square;

public class Rectangle {

   private int width;
   private int height;

   private boolean invariant() {
      return getWidth() > 0 && getHeight() > 0;
   }
   public Rectangle(int width, int heigth) {
      assert width > 0 && height > 0; // Precondition
      this.width = width;
      this.height = heigth;
      assert invariant();
   }
   public int getWidth() { return width; }
   public int getHeight() { return height; }
   ...
```

Warum verwenden wir hier nicht die Setter?

Rechtecke mit Vertrag II/II

```
   ...
   public void setWidth(int width) {
      assert width > 0; // Precondition
      this.width = width;
      assert invariant();
   }

   public void setHeight(int height) {
      assert height > 0; // Precondition
      this.height = height;
      assert invariant();
   }

   public int getPerimeter() {
      return (width+height)*2;
   }
}
```

Wollen wir hier eine Nachbedingung?

Nachbedingung mit Bezug auf Resultat

```
public int getPerimeter() {
   return (width+height)*2;
}
```

```
public int getPerimeter() {
  int result = (width+height)*2;
  // Postcondition
  assert result == (width+height)*2;
  return result;
}
```

Explizites Vormerken des Ergebnisses

Nachbedingung kann von Resultat sprechen

Implementation wäre unmittelbar aus der Nachbedingung abzulesen

Die Unterklasse für Quadrate

```
public class Square extends Rectangle {
   protected boolean invariant() {
      return super.invariant() && getWidth() == getHeight();
   }
   public Square(int length) { super(length, length); }
   public int getLength() { return getWidth(); }
   public void setWidth(int width) { setLength(width); }
   public void setHeight(int height) { setLength(height); }
   public void setLength(int length) {
      assert length > 0; // Precondition
      this.width = length;
      this.height = length;
      assert invariant();
   }
}
```

Wir müssen *Rectangle* offen lassen (siehe nächste Folie) so dass wir auf Invariante und Attribute zugreifen können.

Angepasste Klasse für Rechtecke

```
package oo.dbc.square;

public class Rectangle {

  protected int width;
  protected int height;

  protected boolean invariant() {
     return getWidth() > 0 && getHeight() > 0;
  }
  public Rectangle(int width, int heigth) {
     assert width > 0 && height > 0; // Precondition
     this.width = width;
     this.height = heigth;
     assert invariant();
  }
  public int getWidth() { return width; }
  public int getHeight() { return height; }
  ...
```

"protected" stellt sicher, dass Zugriff nur in Unterklassen zulässig ist.

Das Substitutionsprinzip

- Einfache Formulierung: Ein Untertyp “bewahrt” (in einem abstrakten Sinne) das Verhalten des Obertyps. Deswegen darf man eine Instanz des Untertyps anstelle einer Instanz des Obertyps verwenden.
- Veerbung **CreditAccount extends ZeroCreditAccount**
  - getBalance() -- Verhalten bewahrt.
  - deposit(...) -- Verhalten bewahrt.
  - statement() -- Ausgabe „erweitert“
  - withdraw(...) -- Auszahlung bewahrt bei „geeigneter Invariante“

Was wäre mit der Invariante dass der Kontostand nicht negativ ist?

Erfinder: Barbara Liskov
Empfänger des Turing Award 2008
http://www.infoq.com/presentations/liskov-power-of-abstraction

Substituierbarkeit in Bezug auf Verträge

- Unterklassen müssen bezüglich der Oberklassen folgende Regeln befolgen:
  - gleiche oder stärkere Invariante,
  - gleiche oder schwächere Vorbedingungen,
  - gleiche oder stärkere Nachbedingungen.
- Die obigen Forderungen können konstruktiv wie folgt garantiert werden:
  - Die Vorbedingung wird abgeschwächt durch Disjunktion.
  - Die Nachbedingung wird verstärkt durch Konjunktion.
  - Die Invariante wird auch verstärkt durch Konjunktion.

Invarianten im Konto-Beispiel

- ZeroCreditAccount
  - balance >= 0
- CreditAccount extends ZeroCreditAccount
  - balance >= 0 && ?

Wir dürfen die Invariante nur verstärken, aber selbst die ererbte Invariante ist schon nicht realsierbar in der Unterklasse. **Frage**: Wie können Sie die Invarianten bzw. die Klassen ändern so dass das richtige Verhältnis der Invarianten besteht, aber trotzdem zwei unterschiedliche Klassen ZeroCreditAccount und CreditAccount in Vererbung stehen?

Praktische Substituierbarkeit im Sinne von Zuweisungs- oder Typkompatibilität

* Kontenbeispiel:

  ```
  Account a = new CreditAccount();
  ```

* Formenbeispiel:

  ```
  Shape s = new Circle(...);
  ```

Diese Kompatibilität ist auch zulässig bei aktuellen Argumenten von Untertypen der formalen Argumenttypen.

Man darf auch Instanzen in einen Container einordnen, dessen Elementstyp ein Obertyp des Instanztyps ist.

Liskovsche Substituierbarkeit ist eine semantische Eigenschaft, aber wir beschränken uns häufig auf die Komponente der Typsicherheit davon.

Beispiel für einen schlechten Vertrag

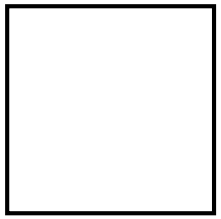
- Nachbedingung für **setWidth**
  - *Rectangle*
    - **getWidth()==width && oldHeight == getHeight()**
  - *Square*
    - **getWidth()==width && getHeight() == width**

Nicht wahr für *Square!*

Diese Nachbedingung ist wahr für *Square*. Es ist aber keine Verstärkung der Nachbedingung von *Rectangle!*

Typen vs. Verträge

- Ein Vergleich:
  - Die Vorbedingung zu einer Methode beschränkt die Methodenargumente.
  - Die Argumenttypen beschränken auch die Methodenargumente.
- Eine Idee:
  - Typen und Verträge muss man vielleicht gar nicht absolut trennen.
  - Beides kann man übrigens statisch und dynamisch prüfen.



“Java ist nicht Alles!”

- Java’s assert bietet nur *naive* Laufzeittests.
  - Zugriff auf initialen Zustand ist schlecht unterstützt.
- Andere Technologien beinhalten auch ***Verifikation***.
  - JML (Java-Plattform)
  - Spec# (.NET-Plattform)

Einschub: JML

Kommentar für Java, Spezifikation für JML.

```
//@ requires amount >= 0;
public void deposit(float amount) {
....
}
```

Es gibt entsprechende Verifikationstechnologie so dass JML-Spezifikationen ***statisch*** und unter Umständen automatisch als korrekt bewiesen werden können.

JML-Erweiterungen zu Java-Ausdrücken

Syntax	Meaning
`\result`	result of method call
a `==>` b	a implies b
a `<==` b	a follows from b (i.e., b implies a)
a `<==>` b	a if and only if b
a `<=!=>` b	not (a if and only if b)
`\old(`E`)`	value of E in pre-state

http://www.eecs.ucf.edu/~leavens/JML/

- **Zusammenfassung**
  - Verträge = Vor- / Nachbedingungen + Invarianten
  - Substitutionsprinzip als Anforderung an Untertypen
  - Aufrufer sind Dienstnutzer / Aufgerufene sind Dienstleister.
- **Ausblick**
  - Syntax von Softwaresprachen
  - Ein- und Ausgabe in OO-Programmen
  - Prüfungsvorbereitung