Das Vertragsmodell

Grundlagen erarbeiten für die
arbeitsteilige Programmierung
und für die
spätere Weiterentwicklung

- **ähnlich für modular und objektorientiert**
- **Vertragsmodell**
- **Fehler- und Ausnahmebehandlung**

Sicherheitsaspekte bei der Softwareentwicklung

Ziel: Es sollen nicht nur **wiederverwendbare**, **erweiterbare** und **änderbare** Klassen, sondern auch

korrekte und robuste

Klassen entwickelt werden können.

1.) Das Vertragsmodell (Programming by contracting)

2.) Vererbung und Zusicherungen (Programming by sub-contracting)

3.) Disziplinierte Ausnahmebehandlung

Motivation des Vertragsmodells

- **Wiederverwendbarkeit** hängt ab**:**
  - **von** Verständlichkeit,
    (Sinn und Zweck einer Klasse müssen verstanden werden),
  - **sichere** Vewendbarkeit,
    (die Klasse muß als korrekt und robust eingeschätzt werden).
- Das **"reine" objektorientierte Modell** liefert**:**
  - **den** Klassennamen,
  - **die** Signaturen **von Routinen.**
- Es **fehlt:**
  - **Spezifikation der Semantik**

Wir erinnern uns:

- Der Kunde einer Klasse sieht **nur** die Schnittstelle. Diese läßt sich genauer durch die Signaturen der Merkmale beschreiben.
- Die Signatur eines Merkmals **f** ist ein Paar **(Argumenttyp, Ergebnistyp)**

Motivation

- Eine Klasse, die eine andere **benutzt** (Benutzt-Beziehung), verwendet i.d.R. die Operationen der anderen Klasse. Sie ist also von den **Diensten** dieser anderen Klasse **abhängig**.
- Die Benutzt-Beziehung zwischen Klassen wird bei Verwendung des Vertragsmodells als **Verhältnis zwischen Klient und Lieferant** interpretiert, die für das Erbringen einer Leistung einen **Vertrag geschlossen** haben. Daraus erwachsen beiden Seiten **Rechte** und **Pflichten**.

Klient

```
public class Zinsberechner
{
   public Zinsberechner(){..};

   public void jahresabschluss()
   {
     int i = 0;
     while (i < _sparbuecher.anzahl()){
       _sparbuecher[i].berechneZinsen();
     }
   }
      ...

   private Sparbuch[] _sparbuecher;
}
```

Dienst

Lieferant

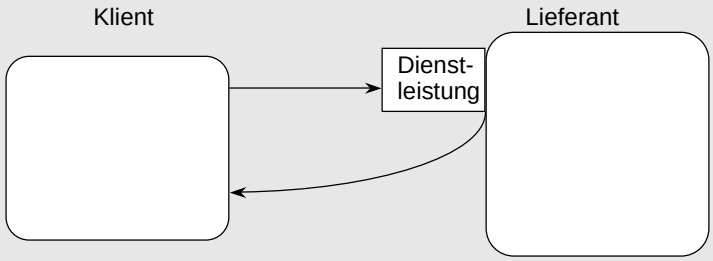
```
public class Sparbuch
{
  public Sparbuch(Zinssatz z) { ..};

  public void berechneZinsen()
  {
    _zinsen = _saldo*_zinssatz.gibWert()
                /(100*365);
  }

  public Zinssatz gibZinssatz()
  {
    return _zinssatz;
  }

  private Zinssatz _zinssatz;
}
```

Das Vertragsmodell - Begriffe



Begriffe

Klient (client)	**Lieferant, Anbieter (**supplier**)**
anfordern (buy)	**anbieten** (sell)
benutzen (use)	**liefern** (supply)
Vertragsnehmer	**Vertragsgeber**

Vertrag (contract)
Dienstleistung (service)

Vertragsmodell: Anwendungsbereich

```
class Printable
  feature
    start_lines is …
    next_line is …
    end_of_lines : BOOLEAN is …

    .
end -- class Printable
```

```
class Printer
  .
  .
  print( d: Printable ) is
    .
    .
    until d.end_of_lines
    do
      d.next_line
      ...
    end
```

```
class Letter
  inherit Printable
    .
    .
end -- class Letter
```

Use

Inherit

- Das **Vertragsmodell**
  (client-supplier model, contracting for reliability)
  - eine (einfache) Interpretation der *Use* -Beziehung,
  - Basis einer Konstruktionstechnik des objektorientierten Entwurfs auch für die *Inherit*- Beziehung,
  - eine Umsetzung der Prinzipien des Entwurfs mit Abstrakten Datentypen.

Vertragsmodell: Angebot und Nachfrage

- Die **Metapher:**
  - **die** Benutzt-Beziehung **zwischen Klassen wird als Verhältnis zwischen** Klient **(Client) und** Lieferant **(Supplier) interpretiert:**
    - **Eine Klasse bietet als Lieferant eine (Dienst-)** Leistung **an, die eine andere Klasse als Klient gebrauchen kann.**
- **Client** und **Supplier:**
  - **Class A ist** client **von Class B und B** supplier **von A, wenn gilt**:

```
Class A
  feature
    an_entity : B;
  ...
    a_procedure (var_name : B)
```

Vertragsmodell und Zusicherungen

- **Der Vertrag:**
  - **Das Erbringen einer Leistung wird vertraglich geregelt. Daraus erwachsen beiden Seiten Rechte und Pflichten.**
  - **Zwischen Klassen wird festgelegt, unter welchen Bedingungen welche Leistung erbracht wird.**
  - **Die Vertragsbedingungen werden mit Hilfe von Zusicherungen festgelegt.**
- **Zusicherungen:**
  - **sind Eigenschaften der Werte einer Größe,**
  - **sind boolesche Ausdrücke,**
  - **schaffen die Verbindung zwischen Klassen und ADTs.**
- **Zusicherungen in Eiffel** (für das Vertragsmodell)**:**
  - **Vor- und Nachbedingungen von Routinen,**
  - **Klasseninvarianten.**

Vertragsmodell: Beispiel

```
class Stack
 feature

   pop is
      require
         not empty
      do
         number_of_elements := number_of_elements - 1;
      ensure
         not full
      end;
      •••
 invariant
   empty implies (number_of_elements = 0)
end -- class Stack
```

Vorbedingung (d.h. Voraussetzungen)

Nachbedingung (d.h. Verpflichtungen)

Invariante (d.h. Randbedingung)

Verträge unterstützen Software-Zuverlässigkeit

Klasseninvariante des **Lieferanten**:

- *Randbedingung* für Verträge zwischen Klient- und Lieferant-Objekt.

Vor- und **Nachbedingungen** einer **Routine**:

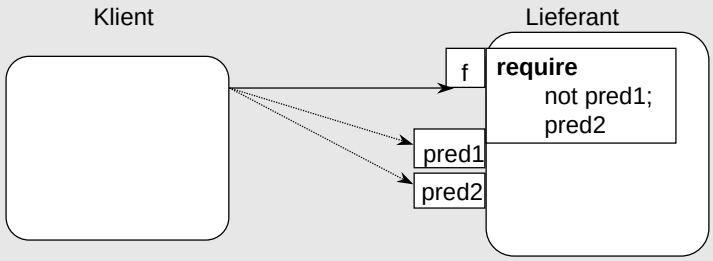
- *Vertrag* (für die Benutzung der Routine), der Klient- und Lieferant-Objekt bindet (bzw. ProgrammiererInnen).
- Ein Vertrag gilt pro Aufruf einer Operation.

Verpflichtungen **Rechte**

Wenn
der **Klient** die **Vorbedingungen** erfüllt,

dann
garantiert der **Lieferant** die **Nachbedingungen**.

Verträge müssen prüfbar sein



Vertragsmodell und Zusicherungen:

- Damit ein Vertrag sinnvoll ist, muß der Klient seine Verpflichtungen überprüfen können. Daher:
  - Alle in einer Vorbedingung verwendeten Merkmale müssen exportiert werden.

Verträge müssen prüfbar sein

- Damit ein Vertrag sinnvoll ist, muß der Klient seine **Verpflichtungen überprüfen** können.
- Alle in einer **Vorbedingung** verwendeten Merkmale müssen **an der Schnittstelle als eigenständige Operationen** vorhanden sein.

Klient

```
Sparbuch einSparbuch = new Sparbuch(3.0);

einSparbuch.einzahlen(100);


einSparbuch.auszahlen(50);  ✓

if (einSparbuch.istBetragGedeckt(200))
{
  einSparbuch.auszahlen(200);  ✓
}
```

Lieferant

```
public class Sparbuch
{
//Vorbedingung: istBetragGedeckt(b)
  public void auszahlen(float b)
  {...};
//Nachbedingung: saldo == old.saldo - b

  public boolean istBetragGedeckt(float b)
  {...};

  private float _saldo;
  private Zinssatz _zinssatz;
};
```

Rechte und Pflichten des Klienten

- Der Klient hat die Pflicht, die Operation nur aufzurufen, wenn der **Zustand** des Lieferanten es erlaubt.
- Dann hat der Klient aber auch das Recht, die **ordnungsgemäße Erfüllung** des Vertrags zu erwarten.

→ Einhalten der Vorbedingungen

→ Erfüllung der Nachbedingungen

Vor- und Nachbedingungen werden an der **Schnittstelle** des Lieferanten spezifiziert.

Lieferant

```
public class Sparbuch
{
//Vorbedingung: istBetragGedeckt(b)
  public void auszahlen(float b)
  {...};
//Nachbedingung: saldo == old.saldo - b

  public boolean istBetragGedeckt(float b)
  {...};

  private float _saldo;
  private Zinssatz _zinssatz;
};
```

Rechte und Pflichten des Lieferanten

- Der Lieferant hat die Pflicht die **versprochene Leistung** zu erbringen.
- Er hat aber gleichzeitig das Recht, bei **Nichteinhaltung des Vertrags** von Seiten des Klienten, die Operation nicht auszuführen.

→ Erfüllung der Nachbedingungen

→ Einhalten der Vorbedingungen

Klient

```
Sparbuch einSparbuch = new Sparbuch(3.0);

einSparbuch.einzahlen(100);

einSparbuch.auszahlen(50);   ✓

einSparbuch.auszahlen(200);  ❍
```

Lieferant

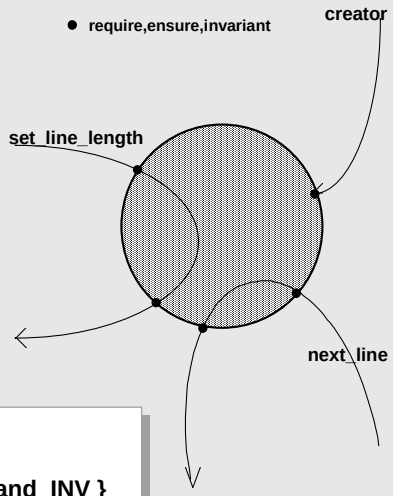
```
public class Sparbuch
{
//Vorbedingung: istBetragGedeckt(b)
  public void auszahlen(float b)
  {...};
//Nachbedingung: saldo == old.saldo - b

  public boolean istBetragGedeckt(float b)
  {...};

  private float _saldo;
  private Zinssatz _zinssatz;
};
```

Dynamik von Zusicherungen in Eiffel

```
class Letter
  ...
  feature
    set_line_length (l : INTEGER) is …
    start_line is …
    ...
    next_line: String is
      require  not end_of_lines
      ...
      ensure  Result.count <= max_length
      end;
      ...
  invariant
    max_length <= 80
end -- class Letter
```



$\{ pre_{Creator} \}\ do_{Creator}\ \{ INV \}$

$\{ pre_f \text{ and } \underline{INV} \}\ do_f\ \{ post_f \text{ and } INV \}$

für alle exportierten features f

Vor- und Nachbedingungen in Java

- Vor- und Nachbedingungen sind nicht Bestandteil der Java Programmiersprache. Sie müssen mit in Java vorhandenen Mittel nachgebildet werden.
- Üblicherweise stellt man hierfür Klassenmethoden zur Verfügung:

- `Contract.require(Bedingung, Objekt, "Fehler Text")` für Vorbedingungen
- `Contract.ensure(Bedingung, Objekt, "Fehler Text")` für Nachbedingungen
- `Contract.check(Bedingung, Objekt, "Fehler Text"` für Bedingungen

Invarianten werden nachgebildet, indem die Klasse eine Methode mit dem Namen "`invariant()`" implementiert. Sie müssen besonders gut dokumentiert werden. Es sind allerdings keine besonderen Sprachmittel wie *`old`* oder nochange vorhanden.

REQUIRE (in Java)

```
import wam.base.contract.Contract
...

public Element gibAktuellesBehälterElement() {
   Contract.require( !istBehälterLeer(), this, “Behälter ist
     leer” );
   ...
 }

public void sortiereElementEin( Element ) {
   Contract.require( Element != null, this, “Element ist
     ungültig” );

   if ( !istBehälterLeer() ) {
      if ( Element == gibAktuellesBehälterElement() )

      ...
}
```

- Die Funktion **`gibAktuellesBehälterElement`** hat als Vorbedingung 'Behälter darf nicht leer sein'. Wird diese Funktion von einer anderen Funktion, z.B. **`sortiereElementEin`** aufgerufen, ist die aufrufende Funktion verpflichtet, die Vorbedingung der aufgerufenen Funktion zu überprüfen.

ENSURE (in Java)

- **`Ensure:`** Ensure ist eine Ausgangsbedingung einer Funktion, d.h. diese Bedingung muß am Ende der Funktion erfüllt sein.
- **`Ensure`** ist die letzte Anweisung in einer Funktion vor dem 'return'. **`Ensure`** dient der Überprüfung des Ergebnisses einer Funktion.

```
public void löscheListe(void) {
   ...
   for ( ... ){
      removeElement(AktuellesElement);
   }

   Contract.ensure( istListeLeer() )
}
```

- Die Funktion **`löscheListe`** hat die Aufgabe alle Elemente einer Liste zu löschen. Ist die Funktion beendet, muß die Liste also leer sein.

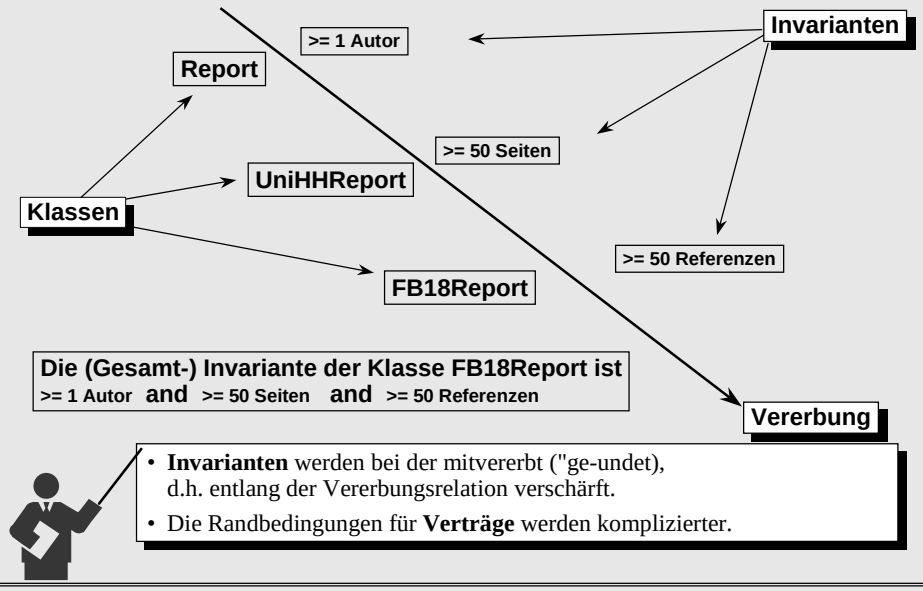
Zusicherungen und Vererbung

Die Metapher:

- Verträge können an Zulieferer weitergegeben werden.
- Ein Zulieferer muß mindestens die Leistung erbringen, die der Lieferant versprochen hat.
- Ein Zulieferer darf höchstens die Voraussetzung fordern, die der Lieferant gefordert hat.

- **Vor- und Nachbedingungen:**
  - Eine Vorbedingung kann in Unterklassen nur durch eine oder-Verknüpfung erweitert werden - die Bedingung wird abgeschwächt.
  - Eine Nachbedingung kann in Unterklassen nur durch eine und-Verknüpfung erweitert werden - die Bedingung wird verschärft.
- **Invarianten:**
  - Eine Invariante kann in Unterklassen nur durch eine und-Verknüpfung ergänzt werden - die Bedingung wird verschärft.

Invarianten und Vererbung in Eiffel



- **Invarianten** werden bei der mitvererbt ("ge-undet), d.h. entlang der Vererbungsrelation verschärft.
- Die Randbedingungen für **Verträge** werden komplizierter.

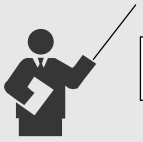
Umsetzen von Vorbedingungen mit der Ausnahmebehandlung

- Ist die Vorbedingung `!isEmpty()` bei Aufruf der Operation `pop()` nicht erfüllt,
  - so scheitert die Operation,
  - aber das Objekt selbst bleibt in einem "stabilen" Zustand, d.h. `_noOfElem >= 0`.

```
class PreconditionException extends Exception
{
  public PreconditionException () {super();}
  public PreconditionException (String s)
  {
    super(s);
  }
}
```

```
public class Stack
{
  // kann nur aufgerufen werden, wenn der
  // Stack nicht leer ist.
  // Vorbedingung !isEmpty()
  public void pop() throws PreconditionException
  {
    try
    {
      if (isEmpty())
      {
        throw new PreconditionException ("..");
      }
      _array.remove(_noOfElem);
      _noOfElem = _noOfElem - 1;
    }

    finally
    {
      _noOfElem = 0;
    }
  }
}
```



Exceptions können auch ohne `try`-Block geworfen werden. Der `try`-Block ist hier wegen des Aufrufs von `_array.remove(_noOfElem)` eingefügt.

"Exception Handling considered harmful ?"

- Ausnahmebehandlung ermöglicht - ähnlich dem Goto - eine Abweichung vom lokalen Kontrollfluß und ist daher diszipliniert zu handhaben.
- Im Sinne des Vertragsmodells stellen wir fest:
  - **Operationen** sind **entweder erfolgreich**, d.h. die Nachbedingungen werden erfüllt,
  - **oder** sie **schlagen fehl**, d.h. die Vorbedingungen sind nicht eingehalten worden.
- Für eine **disziplinierte Ausnahmebehandlung**, die softwaretechnisch vertretbar ist, folgt daraus:
  - Das Fehlschlagen einer Operation führt grundsätzlich zu einer Ausnahme. Im Fehlerhandler wird das Objekt in einen "stabilen" Zustand überführt. Anschließend wird eine für den Kontext aussagekräftige Ausnahme an den rufenden Kontext weitergeleitet ("**Organisierte Panik oder Vertragsmodell**").
- Fehlerhandler sollen **keine alternative Implementierung** für die fehlgeschlagene Operation liefern.

Ein Beispiel für das Wiederaufsetzen

```
public class Stack
{

  public void push (int i)
  {
     try
     {
        _array.put(i, _noOfElem + 1);
        _noOfElem = _noOfElem + 1;
     }

     catch( RangeException e )
     {
        // internes Array vergrößern und
        // den Operationsrumpf von Push()
        // erneut zur Ausführung bringen.
        push(i);
     }
```

Wieder aufsetzende Fehlerhandler sind kein Ersatz für das Beheben von Entwurfs- und Programmierfehlern. Sie lassen sich jedoch sinnvoll für die Behandlung solcher Fehler einsetzen, die “außerhalb” des Programms, also durch die Hardware oder das Betriebssystem, ausgelöst werden.

Vertragsmodell mit JWAM-Contract

- Das JWAM-Rahmenwerk stellt mit der Klasse `Contract` einen Mechanismus zur Verfügung, um **Verträge zu prüfen**.
- Die Klasse `Contract` bietet **statische Operationen** an, mit denen Vorbedingungen (`require`) und Nachbedingungen (`ensure`) überprüft werden können.

```
import de.jwamcontract.*;

public class Stack
{
  /**
   * kann nur aufgerufen werden, wenn der Stack nicht leer ist.
   * @require !isEmpty()
   */
  public void pop()
  {
    Contract.require(!isEmpty(), this, „not empty“);

     _array.remove(_noOfElem);
     _noOfElem = _noOfElem - 1;
  }
}
```