Vorlesung vom 06.06.06: The Java Modelling Language

Wo sind wir?

- Hoare-Kalkül
- Jenseits von Hoare:
  - *Einführung*
  - Reasoning in JML
  - Werkzeugunterstützung
- Korrekte Programmentwicklung

Was ist JML?

The Java Modeling Language (JML) is a behavioral interface specification language that can be used to specify the behavior of Java modules. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus.

Von `http://www.jmlspecs.org/`

Was ist JML?

- JML *spezifiziert Verhalten*
- Baut auf *Java* auf:
  - *Strukturierung* durch Java (Klassen)
  - Syntaktisch: *Annotation* von *Programmen* als *Kommentar*
- Methodologie: *Design by Contract*
- Sprachmittel:
  - *Vor-* und *Nachbedingungen*, *Invarianten*
  - *Ausnahmeverhalten* (exceptional behaviour)

Design-by-Contract

- *Spezifikation* definiert *Vertrag* zwischen
  - den *Benutzern* der Klasse
  - und den *Implementieren*
- *Benutzer* der Klasse
  - müssen *Vorbedingung garantieren*
  - dürfen *auf Nachbedingung vertrauen*
- *Implementierer* der Klassse
  - müssen *Nachbedingung garantieren*
  - dürfen *Vorbedingung annehmen*
- JML-Spezfikation macht Vertrag *explizit*

Vor- und Nachbedingungen

```
/*@ requires amount >= 0;
    ensures balance == \old(balance- amount) &&
            \result == balance; @*/
int debit(int amount)
{ ... }
```

- `\old` für vorherigen Wert
- `\result` für Resultat
- Beliebige *Boolsche Ausdrücke*

Ausnahmeverhalten (Exceptional Behaviour)

```
/* @ requires amount >= 0;
     ensures true;
     signals (BankException e)
             amount > balance &&
             balance == \old(balance) &&
             e.getReason().equals("Insufficient funds")  @*/
int debit(int amount) throws BankException;
```

- Spezifiziert Nachbedingung bei Ausnahme
  `signals (Exception) (Condition)`
- Abwesenheit von Ausnahmen:
  `signals (Exception) false`
  `normal_behaviour`

Klasseninvarianten

```
public class Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
  /*@ invariant 0 <= balance &&
                balance <= MAX_BAL;
    @*/
```

- *Alle* Methoden müssen *Invariante* erhalten
  - Invariante wird implizit Teil der Vor/Nachbedingung aller Methoden
- . . . auch unter *Ausnahmen*!

Klasseninvarianten

- Invarianten dokumentieren *Entwurfsentscheidungen*:

```
public class Directory {
 private File[] files;
 /*@ invariant files != null &&
     (\forall int i; 0<= i && i< files.length;
                     files[i] != null &&
                     files[i].getParent() == this);
  @*/
```


non_null

- Nützliche Abkürzung: `non_null`
- Verwendung in Vor/Nachbedingung oder Invariante

```
public class Directory {
  private /*@ non_null @*/File[] files;

  void createSubdir(/*@ non_null @*/String name) {...}
  ...
  Directory /*@ non_null @*/ getParent(){...}
```


Zusicherungen

- `assert` sichert *Bedingung* zu:

```
if (i<= 0 || j < 0) { ...
  } else if (j< 5) {
   //@ assert i>0 && 0<j && j< 5;
   ...
  } else {
   //@ assert i>0 && j> 5
   ...
  }
```


Zusicherungen

- `assert` Java-Schlüsselwort (seit Java1.4)
- JML: *ausdrucksstärker*, muss *nicht ausführbar* sein:

```
...
for (n= 0; n< a.length();n ++)
 if (a[n] == null) break;
 /@* assert (\forall int i; 0<= i && i< n; a[i] != null);
  @*/
```


Frame properties

- *Frame properties* schränken Seiteneffekte ein:

```
/*@ requires amount >= 0;
    assignable balance
    ensures balance == \old(balance- amount) &&
            \result == balance; @*/
int debit(int amount)
{ ... }
```

  - `debit` kann *nur* `balance` ändern
- Default: `assignable \everything`

Die Reine Lehre

- Methode ohne Seiteneffekte ist `pure`

```
public /*@ pure @*/ int getBalance(){...}

Directory /*@ pure non_null @*/getParent()...}
```

- Reine Methoden: implizit `assignable \nothing`
- (Ausschließlich) reine Methoden in Spezifikation verwendbar

```
/*@ invariant getBalance() && getBalance()<= MAX_BAL @*/
```


Noch ein Beispiel

```
import org.jmlspecs.models.JMLDouble;
public class SqrtExample {

 public final static double eps = 0.0001;

 //@ requires x >= 0.0;
 //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \resu
 public static double sqrt(double x) {
    return Math.sqrt(x);
  }
}
```


Ein längeres Beispiel: `PriorityQueues`

Spezifikation in drei Schritten:

- Signatur: `PriorityQueueUser.java-refined`
- Verhaltensspezifikation: `PriorityQueueUser.jml-refined`
- Entwurfsspezifikation `PriorityQueueUser.java`

Ein längeres Beispiel: `PriorityQueueUser.java-refined`

```
public interface PriorityQueueUser {
 /*@ pure @*/ boolean contains(Object argObj);
 /*@ pure @*/ Object next() throws PQException;
 void remove(Object argObj);
 /*@ pure @*/ boolean isEmpty();
}
```


Ein längeres Beispiel: `PriorityQueueUser.jml-refined`

```
//@ refine "PriorityQueueUser.java-refined";
//@ model import org.jmlspecs.models.*;

public interface PriorityQueueUser {
```


Ein längeres Beispiel: `PriorityQueueUser.jml-refined`

```
 /*@ public model instance JMLValueSet entries;
   @ public initially entries != null && entries.isEmpty();
   @*/
 /*@ public instance invariant entries != null
   @    && (\forall JMLType e; entries.has(e);
   @                          e instanceof QueueEntry);
   @ public instance invariant
   @   (\forall QueueEntry e1; entries.has(e1);
   @       (\forall QueueEntry e2;
   @         entries.has(e2) && !(e1.equals(e2));
   @           e2.obj != e1.obj
   @           && e2.timeStamp != e1.timeStamp ) );
   @*/ }
```


Ein längeres Beispiel: `PriorityQueues`

Spezifikation in drei Schritten:

- Signatur: `PriorityQueueUser.java-refined`
- Verhaltensspezifikation: `PriorityQueueUser.jml-refined`
- Entwurfsspezifikation: `PriorityQueueUser.java`
- Implementation: `PriorityQueue.java`
- `QueueEntry.jml-refined`, `QueueEntry.java`.

Ein längeres Beispiel: `PriorityQueueUser.java`

```
  package org.jmlspecs.samples.jmlkluwer;
//@ refine "PriorityQueueUser.jml-refined";

public interface PriorityQueueUser {

 /*@ also
   @   public normal_behavior
   @     ensures \result <==>
   @          argObj != null
   @          && (\exists QueueEntry e; entries.has(e);
   @                      e.obj == argObj);
   @*/
 /*@ pure @*/ boolean contains(Object argObj);
```



```
 /*@ also
   @   public normal_behavior
   @     requires !entries.isEmpty();
   @     ensures
   @       (\exists QueueEntry r;
   @           entries.has(r) && \result == r.obj;
   @           (\forall QueueEntry o;
   @               entries.has(o) && !(r.equals(o));
   @               r.priorityLevel >= o.priorityLevel
   @               && (r.priorityLevel == o.priorityLevel
   @                   ==> r.timeStamp < o.timeStamp) ) );
```



```
   @ also
   @   public exceptional_behavior
   @     requires entries.isEmpty();
   @     signals_only PQException;
   @*/
 /*@ pure @*/ Object next() throws PQException;
```



```
 /*@ also
   @   public normal_behavior
   @     requires argObj != null && contains(argObj);
   @     assignable entries;
   @     ensures (\exists QueueEntry e;
   @         \old(entries.has(e)) && e.obj == argObj;
   @         entries.equals(\old(entries.remove(e))));
   @ also
   @   public normal_behavior
   @     requires argObj == null || !contains(argObj);
   @     assignable \nothing;
   @     ensures \not_modified(entries);
   @*/
```

```
 void remove(Object argObj);

 /*@ also
   @   public normal_behavior
   @     ensures \result <==> entries.isEmpty();
   @*/
 /*@ pure @*/ boolean isEmpty();
}
```


Zusammenfassung: JML

- *Verhaltensorientierte* Spezifikationssprache
- Java-Erweiterung: *Annotierung* von Java-Programmen
  - Vor/Nachbedingungen mit `requires` und `ensures`
  - Ausnahmeverhalten mit `signals`
  - Klasseninvarianten mit `invariant`
  - Zusicherungen mit `assert`
  - Seiteneffekte mit `assignable` oder `pure`
- *Nächste Woche*: Semantik von JML, Beweisen mit JML, Werkzeuge