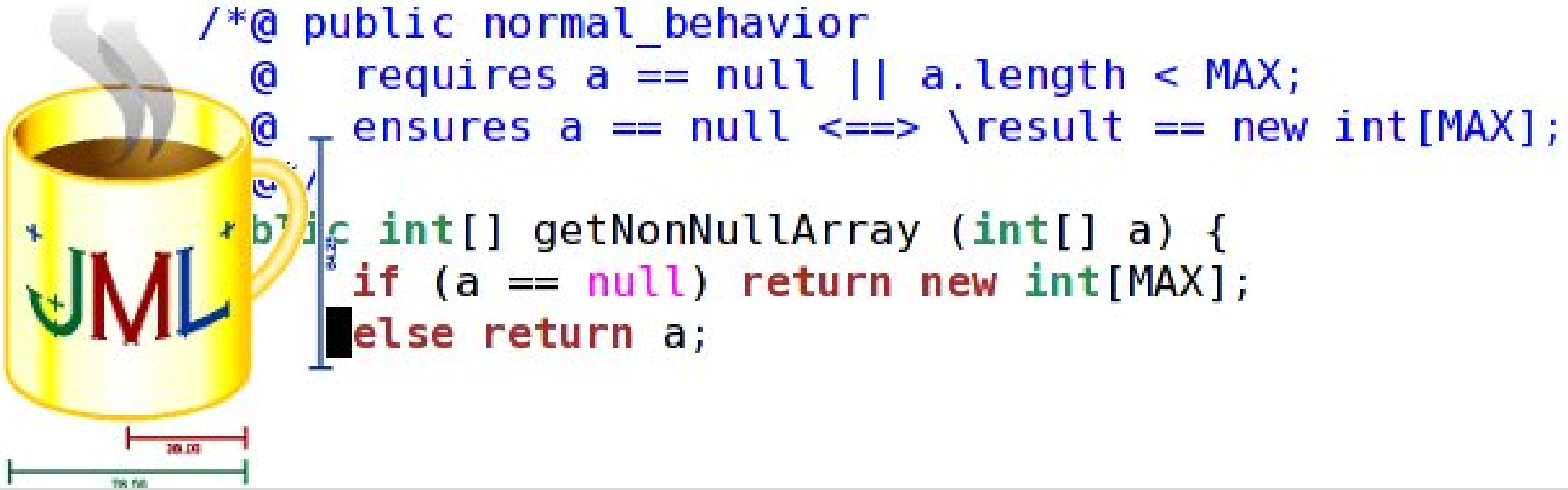
Karlsruher Institut für Technologie

Formale Spezifikation mit Java Modeling Language

Daniel Bruns | Praxis der Software-Entwicklung, 25. November 2010

INSTITUT FÜR THEORETISCHE INFORMATIK



Software ist allgegenwärtig

Software ist allgegenwärtig

Das bedeutet:

Software kann immer und überall Fehler verursachen.

Formale Methoden

- Fehler sollen möglichst früh erkannt – oder vermieden – werden.
- Manche Methoden greifen daher bereits in der Entwurfsphase.
- Alle Methoden basieren auf elementaren mathematischen Konzepten wie Mengentheorie, Prädikatenlogik, etc.
- Aber: Viele sind nur für Experten verständlich.
- *Java Modeling Language* (JML) hat einen anderen Ansatz und zielt darauf ab, vom „Durchschnittsprogrammierer“ genutzt zu werden.
  - speziell für die Spezifikation von Java-Programmen
  - baut syntaktisch auf Java auf
  - wird direkt als Kommentar in `.java`-Dateien geschrieben
  - reichhaltige Sprache; man *muss* aber nicht alle Features nutzen
  - kann verschiedene Werkzeuge bedienen: z.B. statische oder Laufzeitanalysen oder Testfallgenerierung

Formale Methoden

- Fehler sollen möglichst früh erkannt – oder vermieden – werden.
- Manche Methoden greifen daher bereits in der Entwurfsphase.
- Alle Methoden basieren auf elementaren mathematischen Konzepten wie Mengentheorie, Prädikatenlogik, etc.
- Aber: Viele sind nur für Experten verständlich.
- *Java Modeling Language* (JML) hat einen anderen Ansatz und zielt darauf ab, vom „Durchschnittsprogrammierer“ genutzt zu werden.
  - speziell für die Spezifikation von Java-Programmen
  - baut syntaktisch auf Java auf
  - wird direkt als Kommentar in `.java`-Dateien geschrieben
  - reichhaltige Sprache; man *muss* aber nicht alle Features nutzen
  - kann verschiedene Werkzeuge bedienen: z.B. statische oder Laufzeitanalysen oder Testfallgenerierung

Formale Methoden

- Fehler sollen möglichst früh erkannt – oder vermieden – werden.
- Manche Methoden greifen daher bereits in der Entwurfsphase.
- Alle Methoden basieren auf elementaren mathematischen Konzepten wie Mengentheorie, Prädikatenlogik, etc.
- Aber: Viele sind nur für Experten verständlich.
- *Java Modeling Language* (JML) hat einen anderen Ansatz und zielt darauf ab, vom „Durchschnittsprogrammierer“ genutzt zu werden.
  - speziell für die Spezifikation von Java-Programmen
  - baut syntaktisch auf Java auf
  - wird direkt als Kommentar in `.java`-Dateien geschrieben
  - reichhaltige Sprache; man *muss* aber nicht alle Features nutzen
  - kann verschiedene Werkzeuge bedienen: z.B. statische oder Laufzeitanalysen oder Testfallgenerierung

Formale Methoden

- Fehler sollen möglichst früh erkannt – oder vermieden – werden.
- Manche Methoden greifen daher bereits in der Entwurfsphase.
- Alle Methoden basieren auf elementaren mathematischen Konzepten wie Mengentheorie, Prädikatenlogik, etc.
- Aber: Viele sind nur für Experten verständlich.
- *Java Modeling Language* (JML) hat einen anderen Ansatz und zielt darauf ab, vom „Durchschnittsprogrammierer“ genutzt zu werden.
  - speziell für die Spezifikation von Java-Programmen
  - baut syntaktisch auf Java auf
  - wird direkt als Kommentar in `.java`-Dateien geschrieben
  - reichhaltige Sprache; man *muss* aber nicht alle Features nutzen
  - kann verschiedene Werkzeuge bedienen: z.B. statische oder Laufzeitanalysen oder Testfallgenerierung

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
//@ ensures \result == x + y;
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Motivierendes Beispiel

```
//@ requires x >= 0;
//@ ensures \result == x + y;
int add (int x, int y) {
    while (0 < x--) y++;
    return y;
}
```

x	*y*	*res*
0	0	0
0	1	1
1	0	1
1	1	2
⋮	⋮	⋮
23	42	65
⋮	⋮	⋮

Grundlegende Syntax

Alle JML-Spezifikationen sind spezielle Kommentare

- können daher von „normalen“ Java-Parsern ignoriert werden
- Zeile: `//@ requires x >= 0;`
- Block: `/*@ requires x >= 0; @*/`
- Im Block ist es üblich, eine neue Zeile mit `@` zu beginnen.
- JML-Spezifikationen können auch Kommentare enthalten.
- beginnen mit Schlüsselwort (z.B. `requires`) und enden mit Semikolon

```
/*@ requires x >= 0;
  @ requires true; // nicht nützlich
  @ requires /* das auch nicht */ false;
  @*/
```

Grundlegende Syntax

Alle JML-Spezifikationen sind spezielle Kommentare

- können daher von „normalen“ Java-Parsern ignoriert werden
- Zeile: `//@ requires x >= 0;`
- Block: `/*@ requires x >= 0; @*/`
- Im Block ist es üblich, eine neue Zeile mit `@` zu beginnen.
- JML-Spezifikationen können auch Kommentare enthalten.
- beginnen mit Schlüsselwort (z.B. `requires`) und enden mit Semikolon

```
/*@ requires x >= 0;
  @ requires true; // nicht nützlich
  @ requires /* das auch nicht */ false;
  @*/
```

Grundlegende Syntax

Alle JML-Spezifikationen sind spezielle Kommentare

- können daher von „normalen“ Java-Parsern ignoriert werden
- Zeile: `//@ requires x >= 0;`
- Block: `/*@ requires x >= 0; @*/`
- Im Block ist es üblich, eine neue Zeile mit `@` zu beginnen.
- JML-Spezifikationen können auch Kommentare enthalten.
- beginnen mit Schlüsselwort (z.B. `requires`) und enden mit Semikolon

```
/*@ requires x >= 0;
  @ requires true; // nicht nützlich
  @ requires /* das auch nicht */ false;
  @*/
```

Ausdrücke in JML

`&&`	logisches UND
`||`	logisches ODER
`==>`	logische Implikation
`<==>`	logische Äquivalenz
`== !=`	(Un-)Gleichheit (wie in Java!)
`+ - * / %`	Arithmetik wie in Java
`< <= > >=`	Vergleiche wie in Java
`\result`	Rückgabewert einer Methode
`\old(x)`	Wert von *x* im Anfangszustand

Ausdrücke in JML

`&&`	logisches UND
`||`	logisches ODER
`==>`	logische Implikation
`<==>`	logische Äquivalenz
`== !=`	(Un-)Gleichheit (wie in Java!)
`+ - * / %`	Arithmetik wie in Java
`< <= > >=`	Vergleiche wie in Java
`\result`	Rückgabewert einer Methode
`\old(x)`	Wert von *x* im Anfangszustand

Beispiel

```
public class Example {
    private int x;

    /*@ ensures x == \old(x)-1 &&
      @               (x != 0 ==> \result == y/x);
      @*/
    public int foo (int y) {
        if (x-- == 1)
            return 42;
        else
            return y/x;
    }
}
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires α;
//@   ensures  β;
T m (T1 p1, ..., Tn pn);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires α;
//@   ensures  β;
T m (T1 p1, ..., Tn pn);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires α;
//@   ensures  β;
T m (T1 p1, ..., Tn pn);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires α;
//@   ensures  β;
T m (T1 p1, ..., Tn pn);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires x >= 0;
//@   ensures  \result == x + y;
int add (int x, int y);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires x >= 0;
//@   ensures  \result == x + y;
int add (int x, int y);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires x >= 0;
//@   ensures  \result == x + y;
int add (int x, int y);
```

Methodenverträge

„Design by Contract“ (Meyer 1992)

- Wenn eine Methode `m` in einem Zustand aufgerufen wird, in dem eine Vorbedingung α erfüllt ist, dann gilt, nach erfolgreicher Ausführung, eine Nachbedingung β.
- Zwischen aufrufender und aufgerufener Methode wird ein Vertrag geschlossen:
  - Der Aufrufer garantiert die Vorbedingung und
  - der Aufgerufene garaniert die Nachbedingung.

```
//@   requires x >= 0;
//@   ensures  \result == x + y;
int add (int x, int y);
```

Exceptions

- `ensures` behandelt nur den Fall, dass die Methode normal terminiert, d.h. ohne Ausnahmen.
- Erwartete Ausnahmen werden mit `signals (E e)` angegeben (mit `Exception`-Parameter) und Nachbedingung.

Beispiel

```
/*@ ensures \result == y/x;
  @ signals (ArithmeticException e) x == 0;
  @ signals (NullPointerException d) y == null;
  @*/
public int strangeDivide (int x, Integer y) {
    return y.intValue()/x;
}
```

Exceptions

- `ensures` behandelt nur den Fall, dass die Methode normal terminiert, d.h. ohne Ausnahmen.
- Erwartete Ausnahmen werden mit `signals (E e)` angegeben (mit `Exception`-Parameter) und Nachbedingung.

Beispiel

```
/*@ ensures \result == y/x;
  @ signals (ArithmeticException e) x == 0;
  @ signals (NullPointerException d) y == null;
  @*/
public int strangeDivide (int x, Integer y) {
    return y.intValue()/x;
}
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires α;
  @   ensures β;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires α;
  @   ensures β;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires α;
  @   ensures β;
  @   signals (E e) false;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@ normal_behavior
  @   requires α;
  @   ensures β;
  @   signals (E e) false;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires α;
  @   ensures false;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@ exceptional_behavior
  @   requires α;
  @   ensures false;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires p != null && α;
  @   ensures \result != null && β;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires p != null && α;
  @   ensures \result != null && β;
  @   signals (E e) γ;
  @*/
public/*@ non_null @*/ T m (/*@ non_null @*/ U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires p != null && α;
  @   ensures \result != null && β;
  @   signals (E e) γ;
  @*/
public T m ( U p,...);
```

Default-Werte

- Leere Klauseln sind immer `true`.
- `normal_behavior` steht für `signals false`.
- `exceptional_behavior` steht für `ensures false`.
- `non_null` ist der Default.
- Ansonsten muss `nullable` verwendet werden.

```
/*@
  @   requires α;
  @   ensures β;
  @   signals (E e) γ;
  @*/
public/*@ nullable @*/ T m (/*@ nullable @*/ U p,...);
```

Mehrere Spezifikationsfälle

Klauseln gleichen Typs werden konjunktiv verknüft.

```
/*@ requires α1;
  @ requires α2;
  @ ensures  β1;
  @ ensures β2; @*/
```

Mehrere Verträge sind unabhängig voneinander.

```
/*@ normal_behavior
  @   requires α;
  @   ensures  β;
  @ also
  @ exceptional_behavior
  @   requires  δ;
  @   signals (E e) γ; @*/
```

Mehrere Spezifikationsfälle

Klauseln gleichen Typs werden konjunktiv verknüft.

```
/*@ requires α1 && α2;
  @
  @ ensures  β1 && β2;
  @  @*/
```

Mehrere Verträge sind unabhängig voneinander.

```
/*@ normal_behavior
  @   requires α;
  @   ensures  β;
  @ also
  @ exceptional_behavior
  @   requires  δ;
  @   signals (E e) γ; @*/
```

Mehrere Spezifikationsfälle

Klauseln gleichen Typs werden konjunktiv verknüft.

```
/*@ requires α1 && α2;
  @
  @ ensures  β1 && β2;
  @  @*/
```

Mehrere Verträge sind unabhängig voneinander.

```
/*@ normal_behavior
  @   requires α;
  @   ensures  β;
  @ also
  @ exceptional_behavior
  @   requires  δ;
  @   signals (E e) γ; @*/
```

Literatur


Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
Preliminary design of JML: A behavioral interface specification language for Java.
Technical Report 98-06y, Iowa State University, 2003.


Gary T. Leavens et al.
JML reference manual.
Entwurfsfassung Revision 1.235, verfügbar unter `http://www.jmlspecs.org/`, September 2009.


Peter H. Schmitt.
Spezifikation und Verifikation von Software, 2010.
Skript zur Vorlesung. Verfügbar unter `http://i12www.ira.uka.de/~pschmitt/FSS/`.

Übungsaufgabe 1

```
public class Riddle {

    private int x;

    public int twiddle (int z) {
        return (z + x)/ ++x ;
    }
}
```

Übungsaufgabe 2

```
public class Mystery {

    private int secret;

    public static void do (Mystery a, Mystery b){
        int x = a.secret--;
        b.secret += a.secret;
        a.secret -= x;
    }
}
```