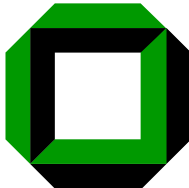
# Formale Entwicklung objektorientierter Software

## Praktikum im Wintersemester 2008/2009

Prof. P. H. Schmitt
Christian Engel, Benjamin Weiß

Institut für Theoretische Informatik
Universität Karlsruhe

5. November 2008



# More on JML

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition
then callee guarantees certain outcome

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation, blame assignment

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation, blame assignment
- Natural language specs are very important

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

> If caller guarantees precondition
> then callee guarantees certain outcome

- Interface documentation, blame assignment
- Natural language specs are very important
- But this course's focus: *"formal" specifications*, i.e., contracts described in a mathematically precise language (JML)

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation, blame assignment
- Natural language specs are very important
- But this course's focus: *"formal" specifications*, i.e., contracts described in a mathematically precise language (JML)
  - higher degree of precision

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

If caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation, blame assignment
- Natural language specs are very important
- But this course's focus: *"formal" specifications*, i.e., contracts described in a mathematically precise language (JML)
  - higher degree of precision
  - *automation* of program analysis of various kinds (runtime assertion checking, static verification)

# Design by Contract

## Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module):

> If caller guarantees precondition
> then callee guarantees certain outcome

- Interface documentation, blame assignment
- Natural language specs are very important
- But this course's focus: *"formal" specifications*, i.e., contracts described in a mathematically precise language (JML)
  - higher degree of precision
  - *automation* of program analysis of various kinds (runtime assertion checking, static verification)
- Errors in specifications are at least as common as errors in code, but their discovery gives deep insights in (mis)conceptions of the system

# JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

# JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications

# JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - if it is the first (non-white) character in the line

# JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - if it is the first (non-white) character in the line
  - if it is the last character before '*/'.

# JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with ‘@’ as first character are JML specifications
- Within a JML annotation, an ‘@’ is ignored:
  - if it is the first (non-white) character in the line
  - if it is the last character before ‘*/’.

⇒ The blue ‘@’s are not required, but it’s a *convention* to use them.

# JML Annotations

```
/*@ public normal_behavior               //hello!
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - if it is the first (non-white) character in the line
  - if it is the last character before '*/'.

  ⇒ The blue '@'s are not required, but it's a *convention* to use them.
- JML specifications may themselves contain comments

# Method Contracts

```
/*@ requires r;
  @ assignable a;
  @ diverges d;
  @ ensures e;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;     //what is the caller's obligation?
  @ assignable a;
  @ diverges d;
  @ ensures e;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;      //what is the caller's obligation?
  @ assignable a;    //which locations may be assigned by m?
  @ diverges d;
  @ ensures e;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;      //what is the caller's obligation?
  @ assignable a;    //which locations may be assigned by m?
  @ diverges d;      //when may m non-terminate?
  @ ensures e;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;       //what is the caller's obligation?
  @ assignable a;     //which locations may be assigned by m?
  @ diverges d;       //when may m non-terminate?
  @ ensures e;        //what must hold on normal termination?
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;       //what is the caller's obligation?
  @ assignable a;     //which locations may be assigned by m?
  @ diverges d;       //when may m non-terminate?
  @ ensures e;        //what must hold on normal termination?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s;
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;      //what is the caller's obligation?
  @ assignable a;    //which locations may be assigned by m?
  @ diverges d;      //when may m non-terminate?
  @ ensures e;       //what must hold on normal termination?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

# Method Contracts

```
/*@ requires r;      //what is the caller's obligation?
  @ assignable a;    //which locations may be assigned by m?
  @ diverges d;      //when may m non-terminate?
  @ ensures e;       //what must hold on normal termination?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

## Abbreviations

**normal_behavior** = **signals**(Exception) **false**;

# Method Contracts

```
/*@ requires r;       //what is the caller's obligation?
  @ assignable a;     //which locations may be assigned by m?
  @ diverges d;       //when may m non-terminate?
  @ ensures e;        //what must hold on normal termination?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

## Abbreviations

**normal_behavior** = **signals**(Exception) **false**;

**exceptional_behavior** = **ensures false;**

# Method Contracts

```
/*@ requires r;      //what is the caller's obligation?
  @ assignable a;    //which locations may be assigned by m?
  @ diverges d;      //when may m non-terminate?
  @ ensures e;       //what must hold on normal termination?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

## Abbreviations

**normal_behavior** = **signals**(Exception) **false**;

**exceptional_behavior** = **ensures false;**

keyword '**also**' separates the contracts of a method

# Class Invariants

```
//@ invariant i;
```

# Class Invariants

```
//@ invariant i;
```

- can be placed anywhere in a class (or interface)

# Class Invariants

```
//@ invariant i;
```

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)

# Class Invariants

```
//@ invariant i;
```

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold "always"
  (cf. *visible state semantics*, *observed state semantics*)

# Class Invariants

```
//@ invariant i;
```

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold "always"
  (cf. *visible state semantics*, *observed state semantics*)
- **`instance`** invariants *can*, **`static`** invariants *cannot* refer to **`this`**

# Class Invariants

```
//@ invariant i;
```

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold "always"
  (cf. *visible state semantics*, *observed state semantics*)
- **`instance`** invariants *can*, **`static`** invariants *cannot* refer to **`this`**
- default: **`instance`** within classes, **`static`** within interfaces

# Expressions

- All Java expressions without side-effects

# Expressions

- All Java expressions without side-effects
- ==>, <==>: implication, equivalence

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`

# Expressions

- All Java expressions without side-effects
- ==>, <==>: implication, equivalence
- **\forall**, **\exists**
- **\num_of**, **\sum**, **\product**, **\min**, **\max**
- **\old**(...): referring to pre-state in postconditions

# Expressions

- All Java expressions without side-effects
- ==>, <==>: implication, equivalence
- **\forall**, **\exists**
- **\num_of**, **\sum**, **\product**, **\min**, **\max**
- **\old**(...): referring to pre-state in postconditions
- **\result**: referring to return value in postconditions

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old(...)`: referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

## Example

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- **\forall**, **\exists**
- **\num_of**, **\sum**, **\product**, **\min**, **\max**
- **\old**(...): referring to pre-state in postconditions
- **\result**: referring to return value in postconditions

## Example

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old(...)`: referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

## Example

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
*equivalent to*
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)

(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```

# Expressions

- All Java expressions without side-effects
- ==>, <==>: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old(...)`: referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

## Example

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
*equivalent to*
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
*equivalent to*
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

# Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old(...)`: referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

## Example

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

# `assignable` expressions

Comma-separated list of:

- `e.f` (where `f` a field)

# **`assignable`** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)

# **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

# **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

**Example**

```
C x, y;
//@ assignable x, x.i;
void m() {




}
```

# **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- **nothing**, **everything** (default)

Example

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;
  tmp.i = 27;
  x = y;
  x.i = 27;
}
```

## **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where a an array expression)
- \ **nothing**, \ **everything** (default)

### Example

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;  //allowed (local variable)
  tmp.i = 27;
  x = y;
  x.i = 27;
}
```

# **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

**Example**

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;  //allowed (local variable)
  tmp.i = 27; //allowed (in assignable clause)
  x = y;
  x.i = 27;
}
```

# **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

**Example**

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;  //allowed (local variable)
  tmp.i = 27; //allowed (in assignable clause)
  x = y;      //allowed (in assignable clause)
  x.i = 27;
}
```

## **assignable** expressions

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

### Example

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;  //allowed (local variable)
  tmp.i = 27; //allowed (in assignable clause)
  x = y;      //allowed (in assignable clause)
  x.i = 27;   //forbidden (not local, not in assignable)
}
```

## **assignable** expressions

Comma-separated list of:

- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- **nothing**, **everything** (default)

### Example

```
C x, y;
//@ assignable x, x.i;
void m() {
  C tmp = x;  //allowed (local variable)
  tmp.i = 27; //allowed (in assignable clause)
  x = y;      //allowed (in assignable clause)
  x.i = 27;   //forbidden (not local, not in assignable)
}
```

**assignable** clauses are always evaluated in the pre-state!

# Other JML Features

- Java visibility modifiers, **`spec_public`** modifier

# Other JML Features

- Java visibility modifiers, **`spec_public`** modifier
- **`pure`** modifier
  ($\approx$ '**`diverges false;`**' + '**`assignable \nothing;`**')

# Other JML Features

- Java visibility modifiers, **`spec_public`** modifier
- **`pure`** modifier
  ($\approx$ '**`diverges false;`**' + '**`assignable \nothing;`**')
- model methods, model fields

# Other JML Features

- Java visibility modifiers, **`spec_public`** modifier
- **`pure`** modifier
  ($\approx$ '**`diverges false;`**' + '**`assignable \nothing;`**')
- model methods, model fields
- assertions '`//@` **`assert`** `e;`'

# Other JML Features

- Java visibility modifiers, **spec_public** modifier
- **pure** modifier
  ($\approx$ '**diverges false;**' + '**assignable \nothing;**')
- model methods, model fields
- assertions '`//@` **assert** `e;`'
- specification inheritance

# Other JML Features

- Java visibility modifiers, **`spec_public`** modifier
- **`pure`** modifier
  ($\approx$ '**`diverges false;`**' + '**`assignable \nothing;`**')
- model methods, model fields
- assertions '`//@ assert e;`'
- specification inheritance
- many more...

# Nullity

JML has modifiers `non_null` and `nullable`

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '**invariant** x != **null**;'

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '**invariant** x != **null**;'

```
void m(/*@non_null@*/ Object p);
```

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '`invariant x != null;`'

```
void m(/*@non_null@*/ Object p);
```

⇝ implicit precondition added to all contracts: '`requires p != null;`'

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '`invariant x != null;`'

```
void m(/*@non_null@*/ Object p);
```

⇝ implicit precondition added to all contracts: '`requires p != null;`'

```
/*@non_null@*/ Object m();
```

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '**invariant** x != **null**;'

```
void m(/*@non_null@*/ Object p);
```

⇝ implicit precondition added to all contracts: '**requires** p != **null**;'

```
/*@non_null@*/ Object m();
```

⇝ implicit postcondition added to all contracts:
'**ensures** \result != **null**;'

# Nullity

JML has modifiers **non_null** and **nullable**

```
private /*@spec_public non_null@*/ Object x;
```

⇝ implicit invariant added to class: '**invariant** x != **null**;'

```
void m(/*@non_null@*/ Object p);
```

⇝ implicit precondition added to all contracts: '**requires** p != **null**;'

```
/*@non_null@*/ Object m();
```

⇝ implicit postcondition added to all contracts:
'**ensures** **result** != **null**;'

**non_null** is the default!
If something may be null, you have to declare it **nullable** explicitly.

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- ESC/Java2: leightweight static verification

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- ESC/Java2: leightweight static verification
- KeY: full static verification

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- ESC/Java2: leightweight static verification
- KeY: full static verification

green: tools that you may find helpful for this course

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- ESC/Java2: leightweight static verification
- KeY: full static verification

green: tools that you may find helpful for this course
blue: tools explicitly used in this course

# JML Tools

Many tools support JML (see JML homepage). Among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- ESC/Java2: leightweight static verification
- KeY: full static verification

green: tools that you may find helpful for this course
blue: tools explicitly used in this course

The tools do not yet support the new features of Java 5!
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

**THE**

**THE**
**END**