Software Engineering I (02161)

Week 11

Assoc. Prof. Hubert Baumeister

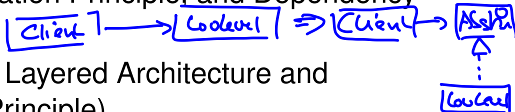
DTU Compute
Technical University of Denmark

Spring 2018

DTU

Recap

- Remaining S.O.L.I.D. principels: Liskov Substitution Principle, Interface Segration Principle, and Dependency Inversion Principle



- Persistency (example of Layered Architecture and Dependency Inversion Principle)
- Hexagonal Architecture / Port and Adapter Pattern

Presentation

Application

Domain

Infrastructure/
Database

Contents

Design by Contract (DbC)
- Contracts
- Implementing DbC in Java
- Assertion vs Tests
- Invariants
- Inheritance
- Defensive Programming

Summary of the course

What does this function do?

```
public List<Integer> f(List<Integer> list) {
   if (list.size() <= 1) return list;

   int p = list.elementAt(0);

   List<Integer> l1 = new ArrayList<Integer>();
   List<Integer> l2 = new ArrayList<Integer>();
   List<Integer> l3 = new ArrayList<Integer>();

   g(p,list,l1,l2,l3);

   List<Integer> r = f(l1);

   r.addAll(l2);
   r.addAll(f(l3));

   return r;
}

public void g(int p, List<Integer> list,
              List<Integer> l1, List<Integer> l2, List<Integer> l3) {
   for (int i : list) {
      if (i < p) l1.add(i);
      if (i == p) l3.add(i);
      if (i > p) l2.add(i);
   }
}
```

What does this function do?

```
public void testEmpy() {
  int[] a = {};
  List<Integer> r = f(Array.asList(a));
  assertTrue(r.isEmpty());
}

public void testOneElement() {
  int[] a = { 3 };
  List<Integer> r = f(Array.asList(a));
  assertEquals(Array.asList(3),r);
}

public void testTwoElements() {
  int[] a = {2, 1};
  List<Integer> r = f(Array.asList(a));
  assertEquals(Array.asList(1,2),r);
}

public void testThreeElements() {
  int[] a = {2, 3, 1};
  List<Integer> r = f(Array.asList(a));
  assertEquals(Array.asList(1,2,3),r);
}
...
```

What does this function do?

```
List<Integer> f(List<Integer> a)
```

Precondition: a is not **null**

Postcondition: For all $result, a \in \text{List}<\text{Integer}>$:

$result == f(a)$ sorting

if and only if

isSorted **p1(result) and p2(a,result)** equals

where

p1(a) if and only if

for all $0 \leq i, j < a.size()$**:**

$i \leq j$ **implies** $a.get(i) \leq a.get(j)$

and

equals *p2(a,b)* if and only if

for all $i \in Integer$**:** $count(a, i) = count(b, i)$

Design by contract

- Pre- and post conditions: Tony Hoare 1969
- Design by contract: Bertrand Meyer 1988
  - Pre- and post conditions in the context of object-orientation

Contract between Caller and the Method

- Caller ensures precondition
- Method ensures postcondition

→ If the client violates precondition (the contract), then the method does not have to guarantee the postcondition

Design by contract

- Pre- and post conditions: Tony Hoare 1969
- Design by contract: Bertrand Meyer 1988
  - Pre- and post conditions in the context of object-orientation

Contract between Caller and the Method

- Caller ensures precondition
- Method ensures postcondition

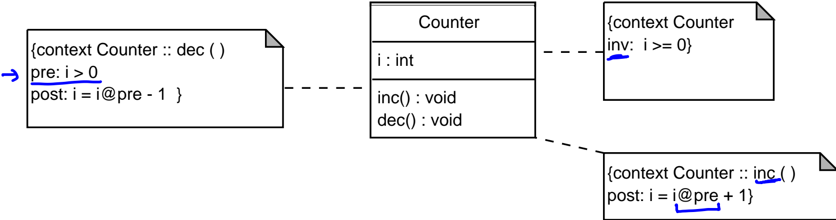
→ If the client violates precondition (the contract), then the method does not have to guarantee the postcondition

→ *The method does not have to check the precondition!!*

- Contracts specify *what* instead of *how*

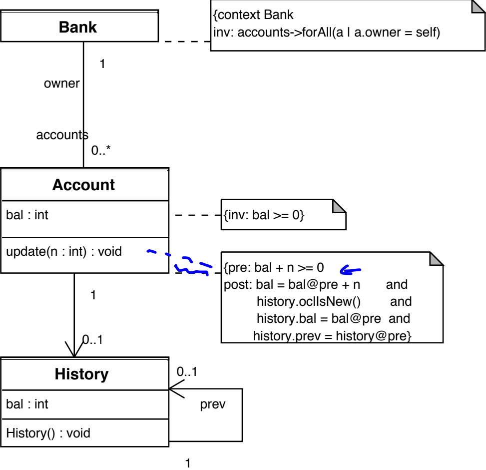
↳ defensive programming

Example Counter




```
public T n(Counter c)
    ...
    // precondition of dec has to hold
    // n has to ensure c.i > 0
    c.dec();
    // client can assume postcondition
    // client can assume c.i = c.i@pre - 1
    ...
```

Bank example with constraints

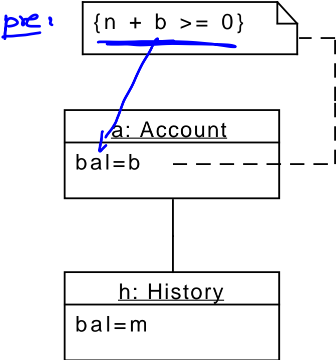


Update operation of Account

```
{pre: bal + n >= 0
post: bal = bal@pre + n       and
        history.oclIsNew()       and
        history.bal = bal@pre  and
       history.prev = history@pre}
```

State **before** executing

`update(n)`

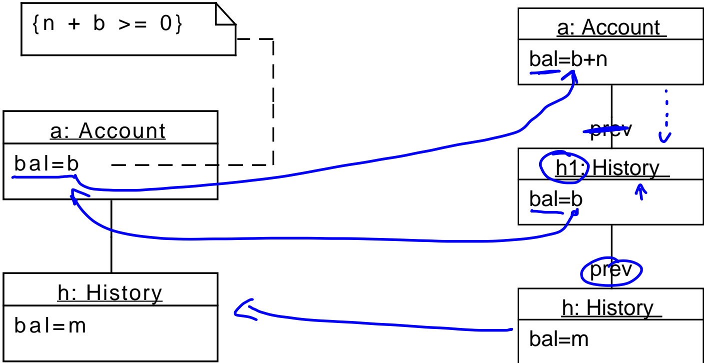


Update operation of Account

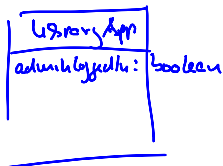
```
{pre: bal + n >= 0
post: bal = bal@pre + n      and
      history.oclIsNew()     and
      history.bal = bal@pre  and
      history.prev = history@pre}
```

State **before** executing `update(n)`

State **after** executing `update(n)`



LibraryApp Example:



Code

```
public void addMedium(Medium medium) {
   checkAdministratorLoggedIn();
   mediumRepository.addMedium(medium);
}
```

pre: adminLoggedIn

post: mediumRepository.getAllMedia =
mediumRepository@pre.getAllMedia.plus(medium)

does not change rep. but creates a new one.

LibraryApp Example:

Code

```
public void addMedium(Medium medium) {
   checkAdministratorLoggedIn();
   mediumRepository.addMedium(medium);
}
```

Contract

```
public void addMedium(Medium medium) {
   pre: adminLoggedIn;
   post: mediumRepository.allAllMedia() ==
         mediumRepository@pre.allAllMedia().plus(medium);
}
```

LibraryApp Example:

Code

```
public List<Medium> search(String searchText) {
   List<Medium> found = new ArrayList<>();
   for (Medium m : mediumRepository.getAllMedia) {
      if (b.match(searchText)) {
         found.add(m);
      }
   }
   return found;
}
```

(pre: true)

post: result = { m | m.match(searchText) && m ∈ mediumRep.getAllMedia }

LibraryApp Example:

Code

```
public List<Medium> search(String searchText) {
   List<Medium> found = new ArrayList<>();
   for (Medium m : mediumRepository.getAllMedia) {
      if (b.match(searchText)) {
         found.add(m);
      }
   }
   return found;
}
```

Contract

```
public List<Medium> search(String searchText) {
   post result == { m | m in mediumRepository.getAllMedia() &&
                        m.match(searchText) }
}
```

User Example:

Code

```
public void borrowMedium(Medium medium, Calendar borrowDate) {
   canBorrow(borrowDate);
   medium.setDueDateFromBorrowDate(borrowDate);
   borrowedMedia.add(medium);
}
```

User Example:

Code

```
public void borrowMedium(Medium medium, Calendar borrowDate) {
   canBorrow(borrowDate);
   medium.setDueDateFromBorrowDate(borrowDate);
   borrowedMedia.add(medium);
}
```

Contract

```
public void borrowMedium(Medium medium, Calendar borrowDate) {
   pre: borrowedMedia.size() < MAX_NUMBER_OF_MEDIA
         && ! hasOverdueMedia(borrowDate)
         && ! hasFine(borrowDate)
   post: medium.getDueDate() == borrowDate + 28 days &&
         borrowedMedia = borrowedMedia@pre.plus(medium)
}
```

MinMax Example

Code

```
public class MinMax {
  int min, max;

  public void minmax(int[] array) throws Error {
    min = max = array[0];
    for (int i = 1; i < array.length; i++) {
      int obs = array[i];
      if (obs > max)
        max = obs;
      else if (min < obs)
        min = obs;
    }
  }
}
```

pre: array != null && array.size() ≥ 1

post: $\forall a \in$ arry: min $\leq a$ && $a \leq$ max && min $\in$ array && max $\in$ array

MinMax Example

Code

```
public class MinMax {
  int min, max;

  public void minmax(int[] array) throws Error {
    min = max = array[0];
    for (int i = 1; i < array.length; i++) {
      int obs = array[i];
      if (obs > max)
        max = obs;
      else if (min < obs)
        min = obs;
    }
  }
}
```

Contract

```
public void minmax(int[] array) throws Error {
  pre: array != null && array.length > 1
  post: forall i in array:
          min <= i <= max
```

min ∈ array && max ∈ array

Postcondition

Assume that `result` denotes the result of the function $f(x : double)$.

$\texttt{result} = \sqrt{x}$

$\texttt{result}^2 = (\sqrt{x})^2 = x$

1) post: $\texttt{result}^2 = x$
2) post: $\texttt{result} = x^2$ → square
3) post: $x^2 = \texttt{result}$ → square — equal sign
4) post: $x = \texttt{result}^2$

Which of this statements describe

a. the postcondition of the square function?
b. the postcondition of the square *root* function?

Precondition

- Given the contract for a method $minmax(int[]array)$ in a class which has instance variables *min* and *max* of type int:

  pre: $array \neq null$ and $array.length > 0$
  post: $\forall i \in array : min \leq i \leq max$ ...

- Which of the following statements is true: if the client calls *minmax* such the precondition is not satisfied

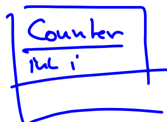
  a) A NullPointerException is thrown
  b) An IndexOutOfBoundsException is thrown
  c) Nothing happens
  d) What happens depends on the implementation of minmax

Implementing DbC with assertions

- Many languages have an assert construct:
  `assert bexp;` or `assert bexp:string;`
- Contract for Counter::dec(i:int)

  Pre: $i > 0$

  Post: $i = i@pre - 1$




```
public void dec() {
  assert      i > 0;
  int i@pre = i
  i--;
  assert i == i@pre - 1
```

Implementing DbC with assertions

- Many languages have an assert construct:
  `assert bexp;` or `assert bexp:string;`
- Contract for Counter::dec(i:int)

  Pre: $i > 0$

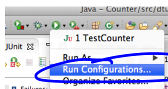
  Post: $i = i@pre - 1$

```
void dec() {
   assert i > 0 : "Precondition violated"; // Precondition
   int prei = i; // Remember the value of the counter
                 // to be used in the postcondition
   i--;
   assert i == prei-1 : "Postcondtion violated"; // Postcondition
}
```

- assert $\neq$ assertTrue

Important

- Assertion checking is switched off by default in Java
  1) `java -ea Main`
  2) In Eclipse



Implementing DbC in Java

Pre: $array \neq null$ and $array.length > 0$

Post: $\forall n \in array : min \leq n \leq max$

```
public class MinMax {
  int min, max;

  public void minmax(Integer[] array) {

    assert array != null && array.length != 0;

    min = max = array[0];
    for (int i = 1; i < array.length; i++) {
      int obs = array[i];
      if (obs > max)
        max = obs;
      else if (min < obs)
        min = obs;
    }

    assert isBetweenMinMax(array);
  }

  private boolean isBetweenMinMax(Integer[] array) {
    return Arrays.asList(array)
                 .stream()
                 .allMatch(i -> min <= i && i <= max);
  }
```

&& array.contains(min) && ...

$\Leftrightarrow \forall$

Assertions

- Advantage
  - Pre-/Postconditions are checked for each computation
- Disadvantage
  - Postcondition checking can be expensive

  $\rightarrow$ Performace problems
- Solution:
  - Assertion checking during developing, debugging, and testing
  - No Assertion checking in production systems

  $\rightarrow$ VM argument `-ea`

Assertion vs. Tests

method under test

Assertions

```
List<Integer> sort(List<Integer> arg) {
   assert arg != null;
   ...
   assert isSorted(result);
   return result;
}
boolean isSorted(List<Integer> arg) {
   boolean sorted = true;
   for (int i = 0; i < arg.length - 1; i++ {
            sorted = sorted && arg[i] <= arg[i+1])
   }
   return sorted;
}
```

DbC

Java language feature

JUnit test

```
@Test
public void t1(List<Integer> arg)
   ... result = sort(arg)
   assertTrue(isSorted(result));
```

→ concrete values

→ JUnit Theories

```
t1([])
t1([5,4,3,2])
t1([6,7,1])
...
```

```
@Test
public void t2() {
```

Assertion vs. Tests

Assertions

```
List<Integer> sort(List<Integer> arg) {
   assert arg != null;
   ...
   assert isSorted(result);
   return result;
}
boolean isSorted(List<Integer> arg) {
   boolean sorted = true;
   for (int i = 0; i < arg.length - 1; i++ {
            sorted = sorted && arg[i] <= arg[i+1])
   }
   return sorted;
}
```

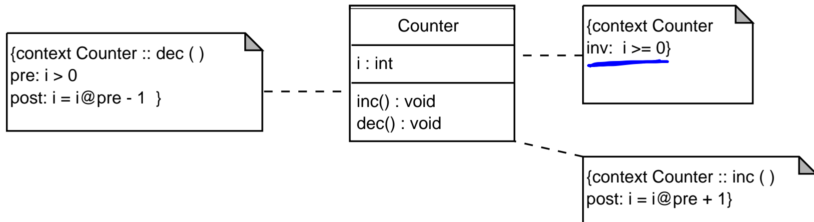
Tests

```
@Test
public void testSorted() {
   List<Integer> result = sort(Array.asList(3,1,2));
   assertTrue(Array.asList(1,2,3),result);
}
```

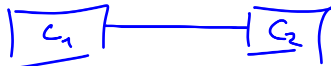
Assertion vs. Tests

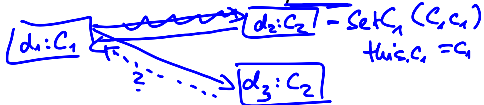
- Assertion
  - Checks *all* computations (is assertion checking is on)
  - Checks also preconditions
- Tests
  - Only checks the concrete test cases
  - Cannot check that clients satisfy contracts

Invariants: Counter



- Methods
  - assume that invariant holds
  - ensure invariants
- When does an invariant hold?
  - After construction
  - After each *public* method





+ setC2(C2 c2)
this.c2 = c2 ←
c2.setC1(this); ← establish the invariant

Invariants

- Contstructor has to ensure invariant

```
public Counter() {
   i = 0;
   assert i >= 0;  // Invariant
}
```

- Operations ensure and assume invariant

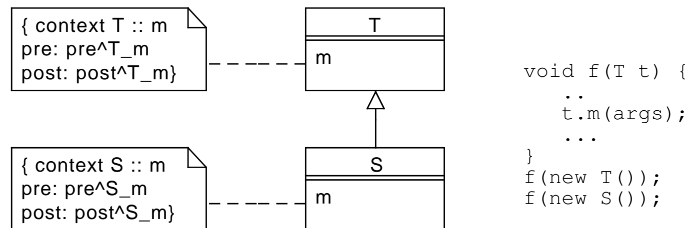
```
void dec() {
-> assert i >= 0; // Invariant
   assert i > 0; // Precondition
   int iatpre = i; // Remember the value of the counter
                 // to be used in the postcondition
   i--;
   assert i == iatpre-1; // Postcondition
-> assert i >= 0; // Invariant
}
```

Contracts and inheritance

Liskov Substition Principle (LSP)

Subtype property S is a **subtype** of T:
"Let $\phi(x)$ be a *property* provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."

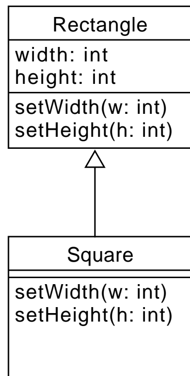
Liskov, B. H.; Wing, J. M. (November 1994). A behavioral notion of subtyping.



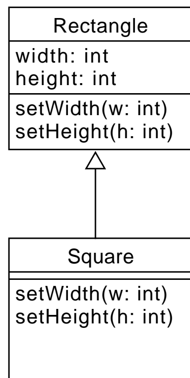

```
void f(T t) {
    ..
    t.m(args);
    ...
}
f(new T());
f(new S());
```

Contracts

Contract Rectangle



Contracts



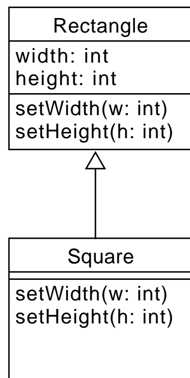
Contract Rectangle

```
inv: width > 0 && height > 0

Rectangle::setWidth(int w)
pre: w > 0
post: width = w && height = height@pre

Rectangle::setHeight(int h)
pre: h > 0
post: height = h && width = width@pre
```

Contract Square extends Rectangle

inv width > 0 && height > 0 && height == width

Contracts



Contract Rectangle

```
inv: width > 0 && height > 0

Rectangle::setWidth(int w)
pre: w > 0
post: width = w && height = height@pre

Rectangle::setHeight(int h)
pre: h > 0
post: height = h && width = width@pre
```

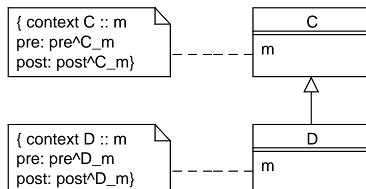
Contract Square extends Rectangle

```
inv: width > 0 && height > 0 && width = height

Square::setWidth(int w)
pre: w > 0
post: width = w && height = w

Square::setHeight(int h)
pre: h > 0
post: height = h && width = h
```

rec.setWidth(w) height is unchanged

sq. setWidth(w) height is changed

Contracts and Inheritance



Weaken precondition

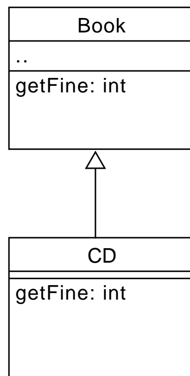
- $Pre^C_m \implies Pre^D_m$

Strengthen postcondition

- $Post^D_m \implies Post^C_m$

```
f(..) {
C c = new C(); C d = new D();
c.m();     <- pre^C is established
d.m();     -> post^C is established
}
```

GetFine Example



Contract Book

```
int Book::getFine()
post: result == 100
```

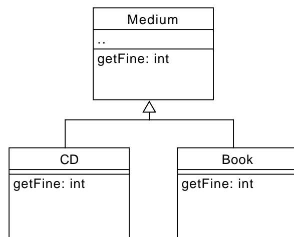
Contract CD extends Book

```
int CD::getFine()
post result == 200
```

(result == 200) ⇏ (result == 1) ⇒ CD is not a subclass of Book

strengthening the postcondition

GetFine Example



Contract Medium

```
int Medium::getFine()
post: result >= 0
```

Contract Book extends Medium

```
int Book::getFine()
post: result == 100
```

result == 100
⟹
result >= 0

Contract CD extends Book

```
int CD::getFine()
post result == 200
```

result == 200
⟹
result >= 0

Defensive Programming

- Contract view
  - client ensures precondition
  - → method does not have to check for the precondition

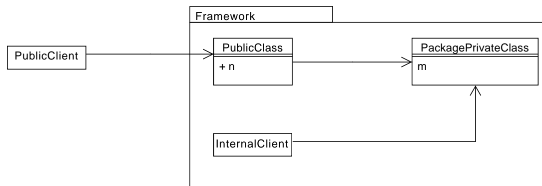
```
void dec() { i--; }
```

Defensive Programming

- Contract view
  - client ensures precondition
  - → method does not have to check for the precondition

```
void dec() { i--; }
```

Package private class vs public class



Defensive Programming

```
void dec() { i--; }                          pre: i > 0
```

- Check the precondition

```
assert i > 0;

if (i <= 0) { System.exit(); }
              throw new Exception(..)
              { return; }
```

Defensive Programming

```
void dec() { i--; }                    pre: i > 0
```

- Check the precondition
- Either

  ```
  void dec() { if (i > 0) { i--; } }
  ```
- Or

  ```
  void dec() {
     if (i <= 0) {
        throw new Exception("Dec not allowed ...");
     }
     i--;
  }
  ```

Defensive Programming

```
void dec() { i--; }                              pre: i > 0
```

- Check the precondition
- Either

  ```
  void dec() { if (i > 0) { i--; } }
  ```
- Or

  ```
  void dec() {
     if (i <= 0) {
        throw new Exception("Dec not allowed ...");
     }
     i--;
  }
  ```
- Don’t rely on the `assert` statement. Why?

  ```
  void dec() {
      assert i <= 0;
      i--;
  }
  ```

Contents

Design by Contract (DbC)

Summary of the course

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams, Domain modelling
- Testing: Systematic Tests, Test-Driven Development, Automated vs Manual Tests
- System Modelling: Class Diagram, Sequence Diagrams, Activity Diagrams
- Design: CRC cards, Refactoring, Layered and Hexagonal Architecture, Design Principles (low coupling/high cohesion, DRY, YAGNI, KISS,...), Design Patterns, Design by Contract, S.O.L.I.D., centralized vs decentralized control
- Software Development Process: Agile Processes

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams, Domain modelling
- Testing: Systematic Tests, Test-Driven Development, Automated vs Manual Tests
- System Modelling: Class Diagram, Sequence Diagrams, Activity Diagrams
- Design: CRC cards, Refactoring, Layered and Hexagonal Architecture, Design Principles (low coupling/high cohesion, DRY, YAGNI, KISS,...), Design Patterns, Design by Contract, S.O.L.I.D., centralized vs decentralized control
- Software Development Process: Agile Processes

- Don't forget the course evaluation

Plan for next weeks

- Week 12: Exercises and status meetings from 13:00 – 15:00
- Week 13: 12.5., 13:00 – 17:00: 10 min demonstrations of the software
  1. Show that all automatic tests run
  2. TA chooses one use case
     - 2.a Show the Cucumber scenarios
     - 2.a Show the systematic tests for that use case
     - 2.b Execute the systematic test manually, i.e. by using your application
  - Schedule will be published this week