Software Engineering I (02161)

Week 11

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2015

Contents

Design by Contract (DbC)

- Contracts
- Implementing DbC in Java
- Assertion vs Tests
- Invariants
- Inheritance
- Defensive Programming

Activity Diagrams

Summary of the course

What does this function do?

sort

```
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?

Contract

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

Precondition: *a* is not **null**

Postcondition: For all *result*, *a* ∈ List<Integer>:

$\mathit{result} == f(a)$

if and only if

isSorted(result) and sameElements(a,result)

where

isSorted(a) if and only if

for all $0 \le i, j < a.\mathit{size}()$**:**

$i \le j$ **implies** $a.\mathit{get}(i) \le a.\mathit{get}(j)$

and

sameElements(a,b) if and only if

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

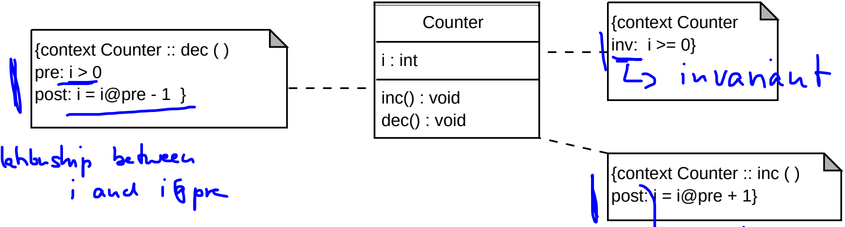
Design by contract

Contract between Caller and the Method

- Caller ensures precondition
- Method ensures postcondition

- Contracts spefify *what* instead of *how*

Example Counter



{context Counter :: dec ( )
pre: i > 0
post: i = i@pre - 1 }

Counter

i : int

inc() : void
dec() : void

{context Counter
inv: i >= 0}

↳ invariant

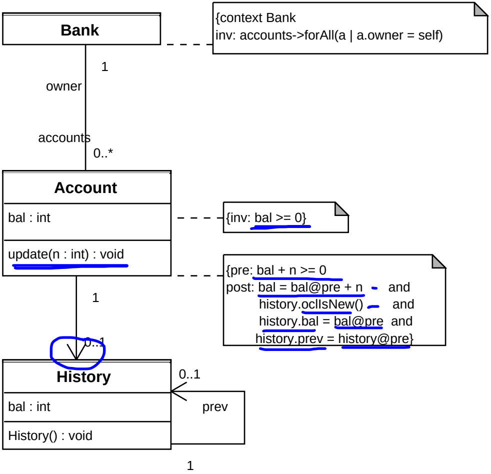
{context Counter :: inc ( )
post: i = i@pre + 1}

Relationship between
i and i@pre

pre : true

```
public T n(T1 a1, .., Tn an, Counter c)
   ...
   // Here the precondition of c has to hold
   // to fulfil the contract
   c.dec();
   // Before returning from dec, c has to ensure the
   // postcondition of dec
   ...
```

Bank example with constraints



Bank

1

owner

accounts

0..*

{context Bank
inv: accounts->forAll(a | a.owner = self)

Account

bal : int

update(n : int) : void

{inv: bal >= 0}

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

1

0..1

History

bal : int

History() : void

0..1

prev

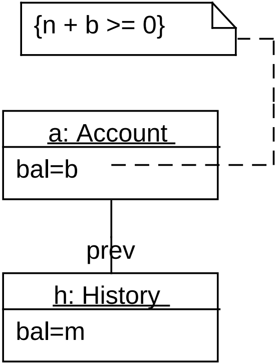
1

Update operation of Account

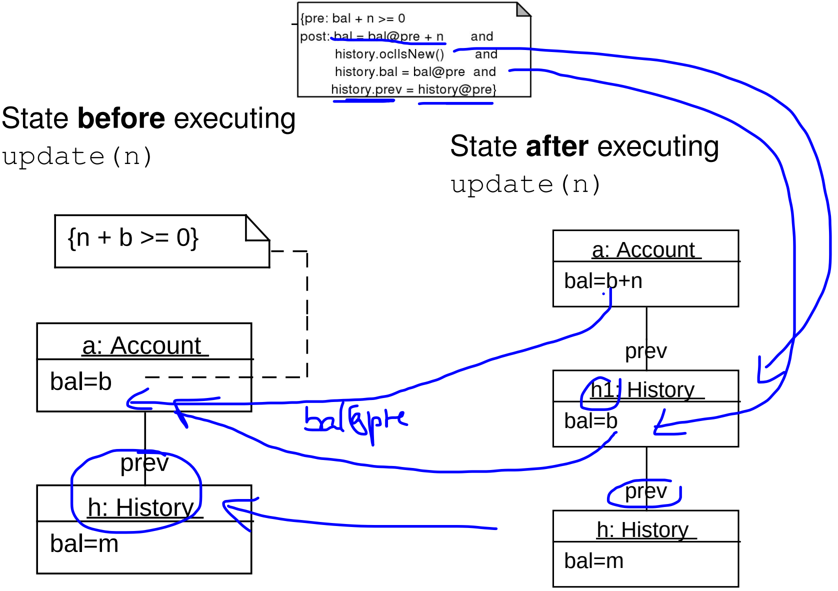
```
{pre: bal + n >= 0
post: bal = bal@pre + n       and
        history.oclIsNew()       and
        history.bal = bal@pre  and
        history.prev = history@pre}
```

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)`

{n + b >= 0}

a: Account
bal=b

prev

h: History
bal=m

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

a: Account
bal=b+n

prev

h1: History
bal=b

prev

h: History
bal=m

bal@pre

Example

```
LibraryApp::addMedium(Medium m)
pre: adminLoggedIn
post: medium = medium@pre->including(m) and
        medium.library = this
```

adding an element
to a set

```
LibraryApp::search(String string) : List<Medium>
post: result = medium->select(m |
                    m.title.contains(string) or
                    m.autor.contains(string) or
                    m.signature.contains(string))
   medium = medium@pre
```

```
User::borrowMedium(Medium m)
pre: borrowedMedium->size < 10
         and m != null
         and not(borrowedMedium->exists(m’  | m’.isOverdue))
post: m.borrowDate = libApp.getDate() and
        borrowedMedium = borrowedMedium@pre->including(m)
```

Postcondition

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

1) post: $\texttt{result}^2 = x$
2) post: $\texttt{result} = x^2$
3) post: $x^2 = \texttt{result}$
4) post: $x = \texttt{result}^2$

Which statements are correct: (multiple answers are possible)

a) 2 + 3 is the postcondition for the function computing the square of a number t 3 f 3
b) Only 2 is the postcondition for the function computing the square of a number t 1 f 5
c) 3 is the postcondition of the square root function t 2 f 6
e) 1 is the postcondition of the square root function t 5 f 0

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 2
  b) An IndexOutOfBoundsException is thrown 0
  c) Nothing happens 0
  t [✓ d) What happens depends on the implementation of minmax] 2

Implementing DbC with assertions

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

  Pre: $i > 0$

  Post: $i = i@pre - 1$

Implementing DbC with assertions

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

  Pre: $i > 0$

  Post: $i = i@pre - 1$

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

Implementing DbC with assertions

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

  Pre: $i > 0$

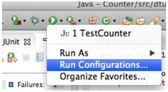
  Post: $i = i@pre - 1$

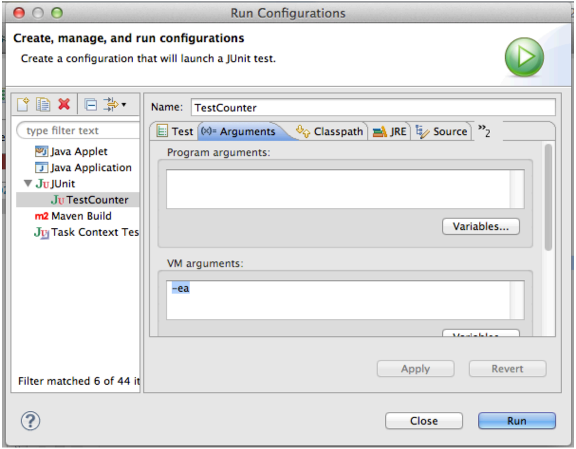
```
void dec() {
   assert i > 0; // Precondition
   int prei = i; // Remember the value of the counter
                 // to be used in the postcondition
   i--;
   assert i == prei-1; // Postcondition
}
```

- assert $\neq$ assertTrue

Important

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





Name: TestCounter

VM arguments: -ea

Implementing DbC in Java

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

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

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

  public void minmax(int[] args) throws Error {
    assert args != null && args.length != 0;
    min = max = args[0];
    for (int i = 1; i < args.length; i++) {
      int obs = args[i];
      if (obs > max)
        max = obs;
      else if (min < obs)
        min = obs;
    }
    assert isBetweenMinMax(args);
  }

  private boolean isBetweenMinMax(int[] array) {
    boolean result = true;
    for (int n : array) {
      result = result && (min <= n && n <= max);
    }
    return result;
  }
```

Assertions

- Advantage
    - Postcondition is checked for each computation
    - Precondition is checked for each computation
- Disadvantage
    - Checking that a postcondition is satisfied can take as as much time as computing the result

    → Performace problems
    - Solution:
        - Assertion checking is switched on during debugging and testing and switched off in production systems
        - Only make assertions for precondition

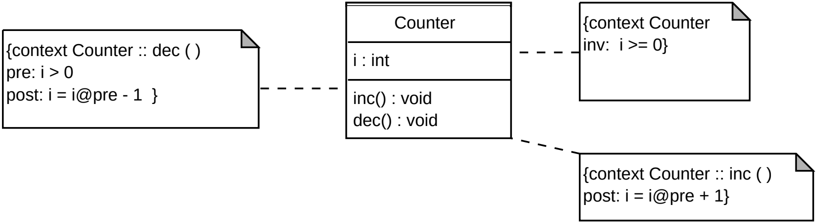
        → Preconditions are usually faster to check

        → Contract violations by the client are more difficult to find than postcondition violations (c.f. assertions vs tests)

Assertion vs. Tests

- Assertion
  - Check all computations (as long as assertion checking is switched on)
  - Check also for contract violations from the client (i.e. precondition violations)
- Tests
  - Only check test cases (concrete values)
  - Cannot check what happens if the contract is violated by the client

Invariants: Counter


{context Counter :: dec ( )
pre: i > 0
post: i = i@pre - 1 }

Counter

i : int

inc() : void
dec() : void

{context Counter
inv: i >= 0}

{context Counter :: inc ( )
post: i = i@pre + 1}

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

Invariants

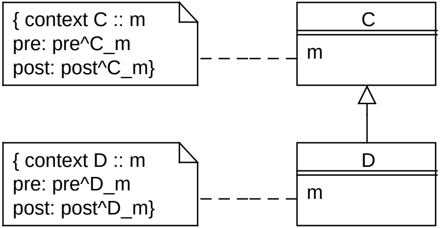
- Contstructor has to ensure invariant

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

- Operations ensure and assume invariant

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

Contracts and inheritance



{ context C :: m
pre: pre^C_m
post: post^C_m}

{ context D :: m
pre: pre^D_m
post: post^D_m}

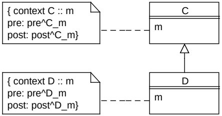
Contracts and Inheritance

Liskov / Wing Substitution principle:

At every place, where one can use objects of the superclass C, one can use objects of the subclass D

new C
new D

```
public T n(C c)
    ...
    // has to ensure Pre^C_m
    c.m();
    // n can rely Post^C_m
    ...
```


{ context C :: m
pre: pre^C_m
post: post^C_m}

{ context D :: m
pre: pre^D_m
post: post^D_m}

- Compare $t.n(newC())$ with $t.n(newD())$.

→ $Pre^C_m \implies Pre^D_m$ *weaker* precondition 1)

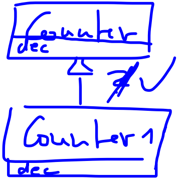
→ $Post^D_m \implies (Pre^C_m \implies Post^C_m)$ 2)
stronger postcondition

Counter vs. Counter1

Counter and Counter1 are identical with the exception of operation dec:

$i>0 \Longrightarrow true$

- Counter::dec
  - pre: $i > 0$
  - $(i>0) \Longrightarrow$ post: $i = i@pre - 1$
- Counter1::dec
  - pre: true
  - post: $(i@pre > 0) \Longrightarrow i = i@pre - 1$ and
    $(i@pre \leq 0) \Longrightarrow i = 0$



Which statement is true?

a) Counter is a subclass of Counter1

b) Counter1 is a subclass of Counter

c) There is no subclass relationship between Counter and Counter1

Defensive Programming

- Can one trust the client to ensure the precondition?

Defensive Programming

- Can one trust the client to ensure the precondition?
- Defensive Programming: don’t trust the client

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

Defensive Programming

- Can one trust the client to ensure the precondition?
- Defensive Programming: don't trust the client

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

- New Contract: No requirement for the client
  - Method has to ensure it works with any argument

    pre: true

    post: $(i@pre > 0) \implies (i = i@pre - 1)$ and
    $(i@pre \leq 0) \implies (i = 0)$

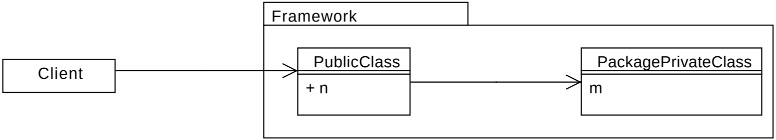
Defensive Programming

- Can one trust the client to ensure the precondition?
- Defensive Programming: don't trust the client

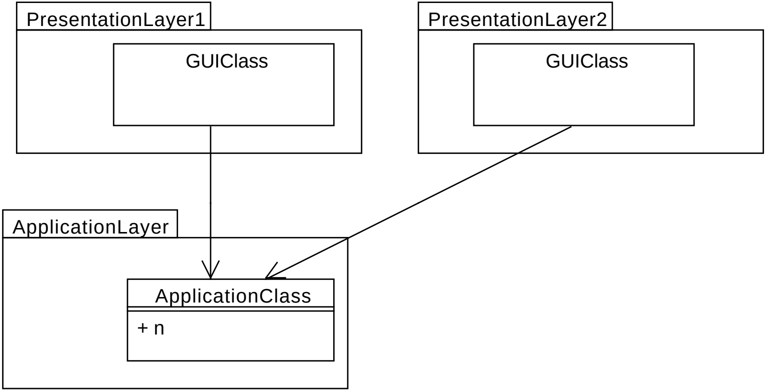
```
void dec() { if (i > 0) { i--; } }
```

- New Contract: No requirement for the client
  - Method has to ensure it works with any argument
    pre: true
    post: $(i@pre > 0) \implies (i = i@pre - 1)$ and
    $(i@pre \leq 0) \implies (i = 0)$
- Or, using *under specification*
  pre: true
  post: $(i@pre > 0) \implies (i = i@pre - 1)$

Defensive Programming



Defensive Programming



Defensive Programming

Given method contracts 1)

```
LibraryApp::addMedium(Medium m)
pre: adminLoggedIn
post: medium = medium@pre->including(m) and
        medium.library = this)
```

and 2)

```
LibraryApp::addMedium(Medium m)
post: adminLoggedIn implies
            medium = medium@pre->including(m) and
           medium.library = this)
```

Which statement is correct?

a) 1) uses defensive programming

b) 2) uses defensive programming

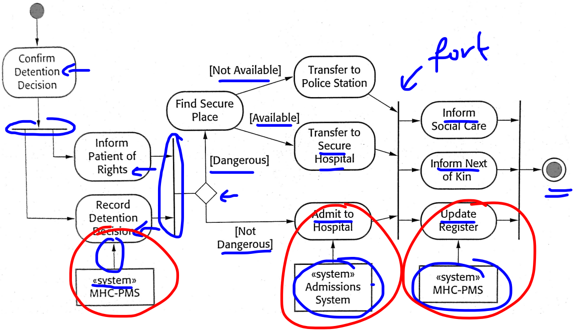
Contents

Design by Contract (DbC)

Activity Diagrams

Summary of the course

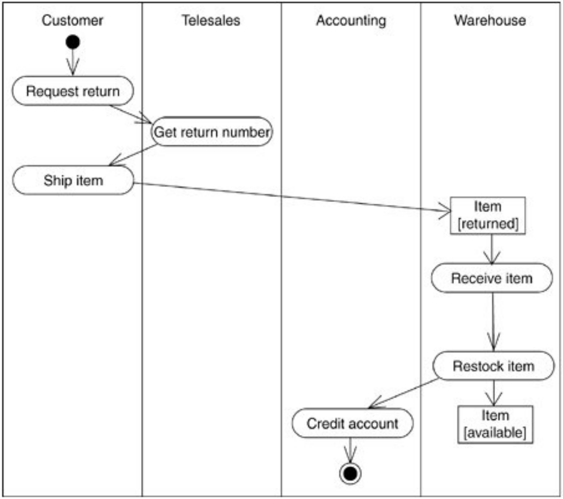
Activity Diagram: Business Processes



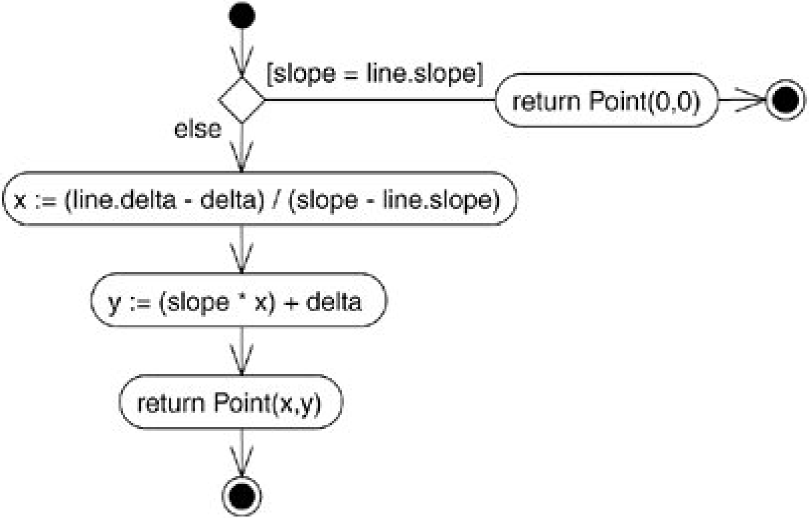
- Describe the *context* of the system
- Helps finding the requirements of a system
  - modelling business processes leads to suggestions for possible systems and ways how to interact with them
  - Software systems need to fit in into existing business processes

Activity Diagram Example Workflow

Swimlane



Activity Diagram Example Operation

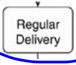


UML Activity Diagrams

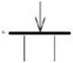
- Focus is on *control flow* and *data flow*
- Good for showing *parallel/concurrent* control flow
- Purpose
  - Model business processes
  - Model workflows
  - Model single operations
- Literature: UML Distilled by Martin Fowler

Activity Diagram Concepts

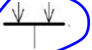
- *Actions*



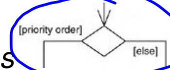
  - ~~Are~~ Can be atomic or can be composite .
  - E.g Sending a message, doing some computation, raising an exception, . . .
    - UML has approx. 45 Action types
- *Concurrency*
  - Fork: Creates concurrent flows



    - Can be true concurrency
    - Can be interleaving
  - Join: Synchronisation of concurrent activities

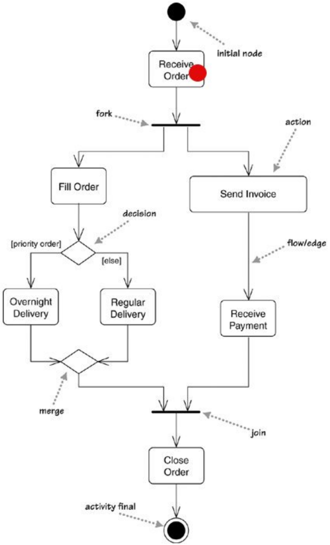


    - Wait for all concurrent activities to finish (based on token semantics)

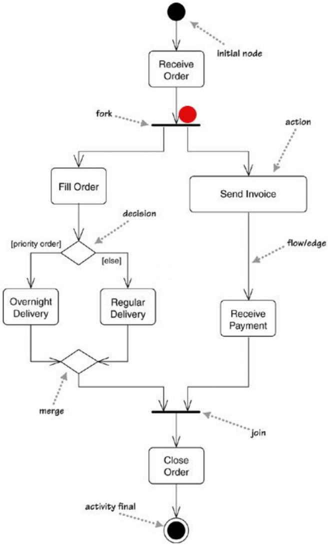


- *Decisions*
  - Notation: Diamond with conditions on outgoing transitions
  - `else` denotes the transition to take if no other condition is satisfied

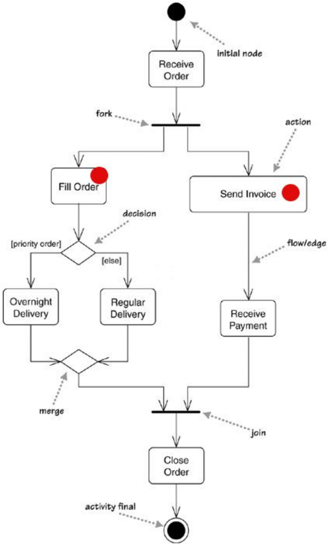
Activity Diagrams Execution



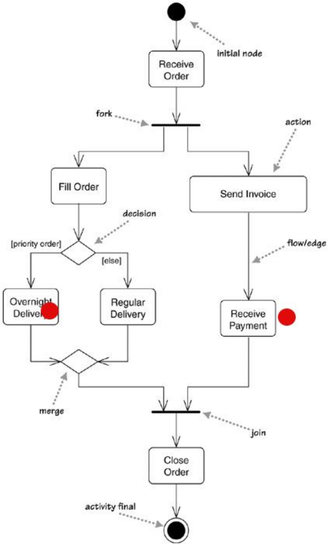
Activity Diagrams Execution



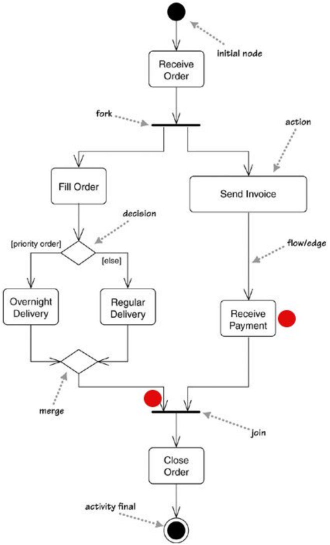
Activity Diagrams Execution



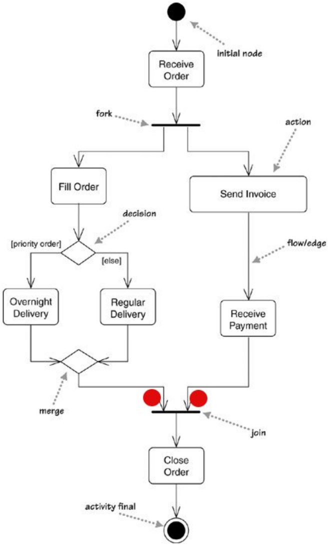
Activity Diagrams Execution



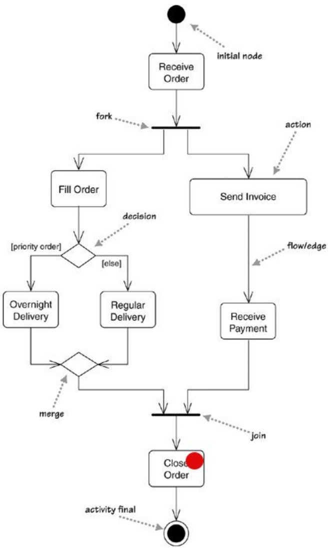
Activity Diagrams Execution



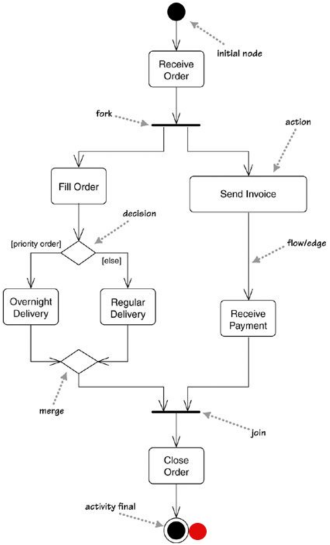
Activity Diagrams Execution



Activity Diagrams Execution

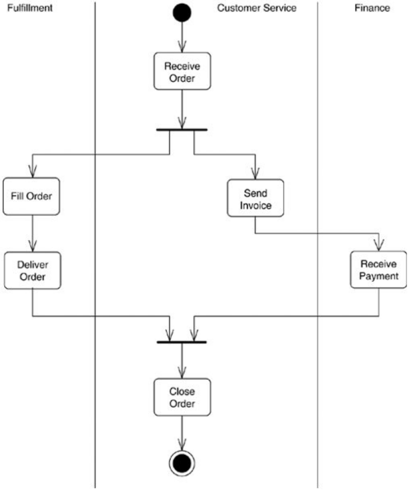


Activity Diagrams Execution

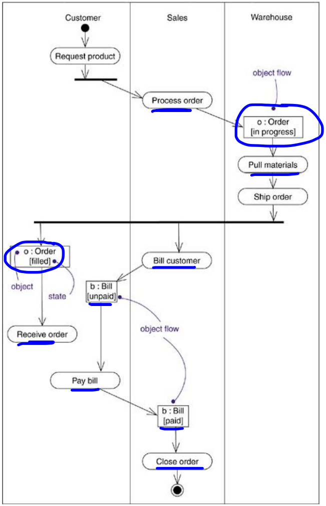


Swimlanes / Partitions

- Swimlanes show **who** is performing an activity

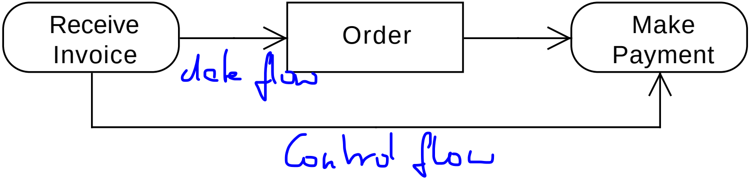


Objectflow example

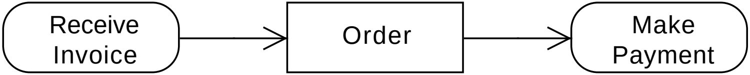


Data flow and Control flow

- *Data flow* and *control flow* are shown:



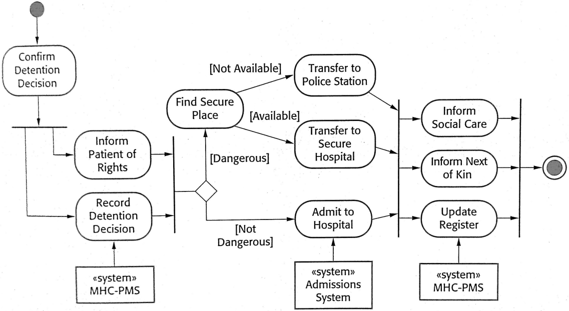
- Control flow can be omitted if implied by the data flow:

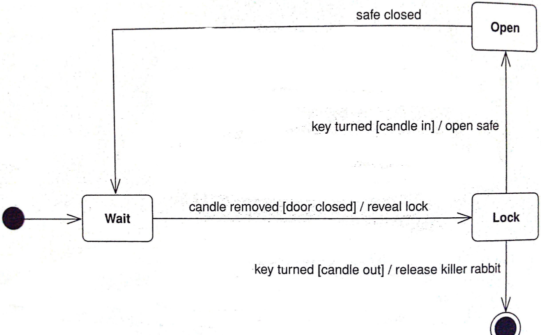


Use of Activity Diagrams

- Emphasise on concurrent/parallel execution
- Requirements phase
    - To model business processes / workflows to be automated
- Design phase
    - Show the semantics of one operation
        - Close to a graphic programming language

Activity Diagram vs State Machines




safe closed
Open
key turned [candle in] / open safe
Wait
candle removed [door closed] / reveal lock
Lock
key turned [candle out] / release killer rabbit

Contents

Design by Contract (DbC)

Activity Diagrams

Summary of the course

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams
- Testing: Systematic Tests, Test-Driven Development
- System Modelling: Class Diagram, Sequence Diagrams, State Machines, Activity Diagrams
- Design: CRC cards, Refactoring, Layered Architecture, Design Principles, Design Patterns
- Software Development Process: Agile Processes, Project Planning
- Design by Contract

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams
- Testing: Systematic Tests, Test-Driven Development
- System Modelling: Class Diagram, Sequence Diagrams, State Machines, Activity Diagrams
- Design: CRC cards, Refactoring, Layered Architecture, Design Principles, Design Patterns
- Software Development Process: Agile Processes, Project Planning
- Design by Contract

- Don’t forget the course evaluation

Plan for next weeks

- Week 12: No lecture. Focus on examination proect.
  - Exercises 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 systematic tests for that use case
     - 2.b Execute the systematic test **manually**
  - Schedule will be published this week