Software Quality FS 2011

Discussion Exercise 2

Cédric Jeanneret

Requirements Engineering Research Group
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/rerg/people/jeanneret

University of Zurich[UZH]

Outline

- Frequent problems in exercise 2
  - Modularity
  - Dependencies
- Introduction to exercise 3
- Formalities of the exam
- Your questions

Exercise 2

- Very well solved in general
- Good discipline within the development environment
  - some errors in commits
  - 1 ticket left open
  - (useful) commit comments

- ImageJ’s quality in use VS ImageJ’s internal quality
- http://imagejdev.org/

Modularity is **important** for software testing and evolution

Especially when working on a large piece of software written by somebody else…

"It was not easy to find the responsible class for the clearing, filling and inverting tasks. I finally found the 'Menus' class, which indicatings which class to use. In those, there are various redirections without clear structure. Image manipulations are not separated to classes, which makes it difficult at times to follow the program logic."

Modularity is **important** for software testing and evolution

- Allows decomposition of a system into simpler pieces & understanding that system in terms of these pieces
- Confines the search for a fault / an enhancement to a single module
- Drives the testing process: unit tests, integration tests, system tests
- Allows the composition of systems from pieces (**reuse**)
- **MVC** is only one possible pattern for decomposing applications

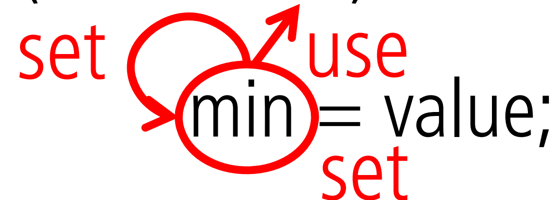
Dependencies

- **Set-Use**: an instruction depends on the result of a previous instruction
- **Use-Set**: an instruction requires a value that is later updated
- **Set-Set**: the ordering of instructions will affect the final output value of a variable
- An instruction B is **control** dependent on a preceding instruction A if the latter determines whether B should execute or not.

Dependencies

Loops are challenging…

```
for (int i=0; i<a.length; i++)
        value = a[i];
        if (value<min)
```




```
        if (value>max)
                max = value;
```

```
int i = 0;
while ( i < a.length)
        value = a[i];
        ...
        i = i + 1;
```

Dependencies

... so are arrays.

set

```
double[] minAndMax = new double[2];
```

use

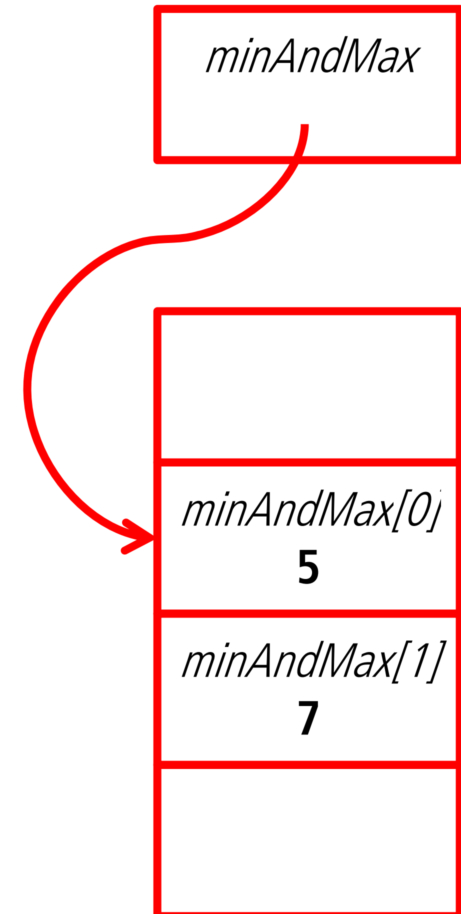
```
minAndMax[0] = min;
```

use

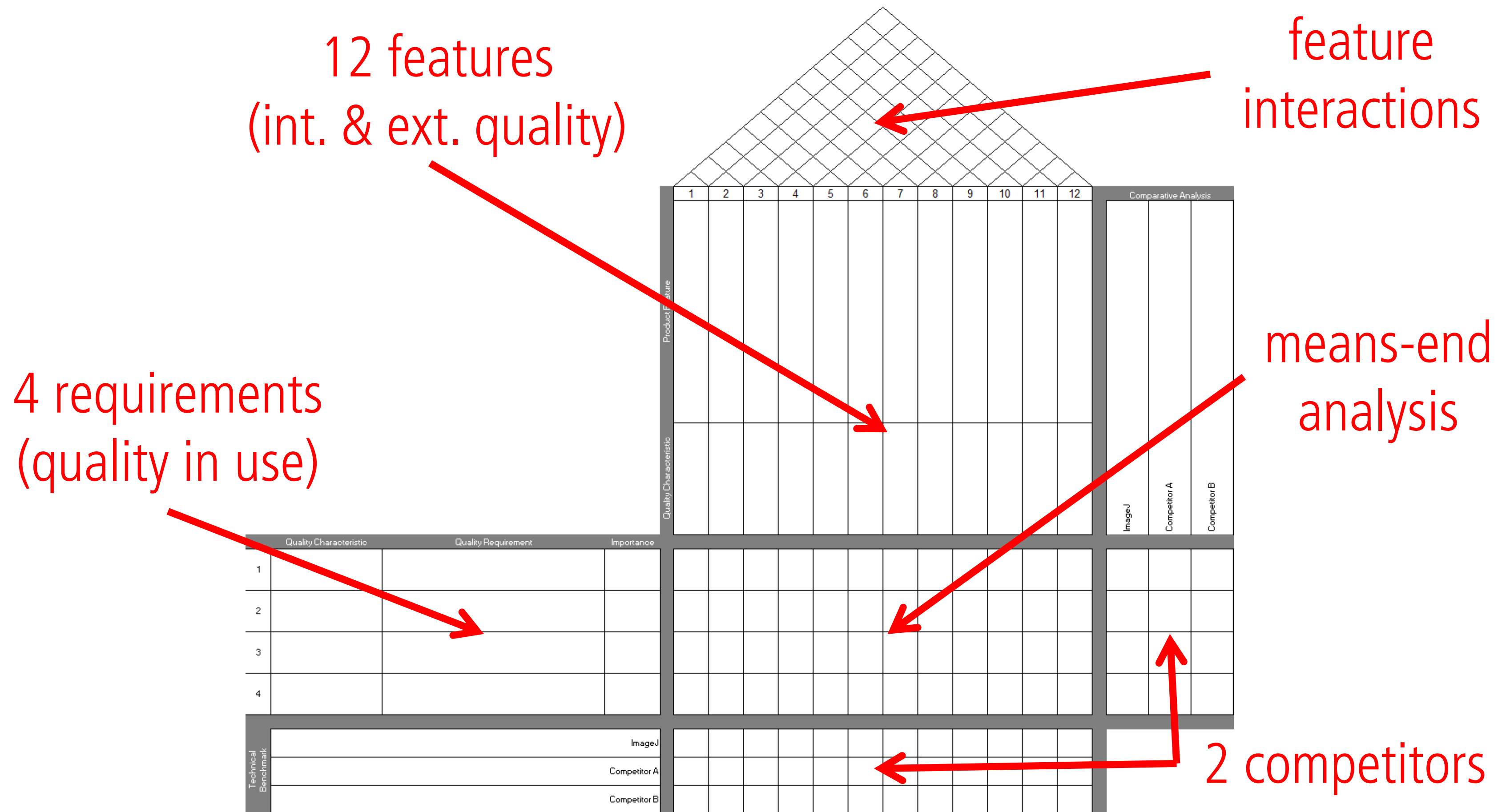
```
minAndMax[1] = max;
```

use

```
return minAndMax;
```



Exercise 3: Improving ImageJ with **QFD** and **ISO / IEC 9126-1**



Exam

Location: BIN 2.A.10

Date: Monday May 2nd, 2pm

Duration: 90 minutes

Language: German

Structure: ~1/3 MCQ, ~1/3 Case Study and ~1/3 Essay

- Sample exam is available on the lecture's website

Scope: Lecture's slides + Exercises

Cheat sheet: 1 double-sided handwritten A4 page

JUnit

?

Program Dependency Graph

?

Müssen bei Spin die LTL-Formeln invertiert werden oder nicht?

Spin looks for an execution **satisfying** a given property

When investigating whether property *P* holds for...
... all executions, let Spin search for a **counterexample**

spin –a –f "!P" ...
spin –a –f "![]P" ... or spin –a –f "<>!P" ...

... none of the executions, let Spin search for an **example**

spin –a –f "P" ...
spin –a –f "<>P" ...

Note the following equivalences:
$\Box\neg A \Leftrightarrow \neg\Diamond A$ and $\Diamond\neg A \Leftrightarrow \neg\Box A$

Müssen bei **Spin** die LTL-Formeln invertiert werden oder nicht?

Pr1: Absence of deadlock

$\Box$ (t1enabled $\lor$ t2enabled $\lor$ t3enabled $\lor$ t4enabled)

Pr2: T4 can be fired at least once

$\Diamond$ t4enabled

Pr3: T3 can be fired an infinite number of time

$\Box\Diamond$ t3enabled

Pr4: As soon as P4 receives a token, it never gets empty again

$\Box\neg p4 \lor (\neg p4 \ U \ \Box \ p4)$

Was sind acceptance cycles bei Spin?

Never-claims generated from a LTL formulae have acceptance states (labels beginning with "accept").

An *acceptance cycle* is an execution that passes through an accept state infinitely often.

Executions violating a **liveness** property are infinite!

The verifier looks for them only with the parameter –a.

Müssen wir den Output von Spin verstehen können?

(never claims generated from
pan: claim violated! (**at dep**
pan: wrote Colony.pml.trail

Statistics about the **trace found**

- *Depth*: # of transitions from the initial system state

[...]

State-vector 28 byte, **depth reac**
 22 states, stored
 0 states, matched
 22 transitions (= stored+m
 0 atomic steps

Statistics about the **search**

- *Transitions*: # of system states
- *Stored states*: # of unique system states
- *Depth*: longest trace

Wie sieht Lamport's Bakery Algorithmus in der **Promela** syntax aus?

- Define 2 arrays as global variables
  - bit choosing[N] and byte number[N]
- Define 1 inline "procedure"
  - To compute the maximum number in the number[] array
- Define 1 process (given number of iterations)
  - Sequence: enter CS, do something in CS, exit CS
  - Local variable: _pid
- Do not use atomic or d_step blocks

Lamport's Bakery Algorithm

Maximum of an array

```
byte number[N];

inline maximum(max, i) {
  i = 0; max = 0;
  do
  :: i < N  ->
     if
     :: max <  number[i] -> max = number[i];
     :: max >= number[i];
     fi; i++;
  :: i == N -> break;
  od; }
```

Lamport's Bakery Algorithm

Skeleton of a **client**

byte number[N];

active [N] **proctype** client()

Doorway:

compute a ticket number and store it (in number[_*pid*])

Backery:

inspect each client process. If process *i* ...

- is choosing a ticket, wait until it has a ticket
- has a lower ticket number, wait until *i* has gone through CS
- has the same ticket number, but *i*<_*pid*, wait until *i* has gone through CS

Service (Critical Section):

set the ticket number to 0 and **goto** doorway again