Test & Debugging
(af programmer)

GRPRO: "Grundlæggende Programmering"

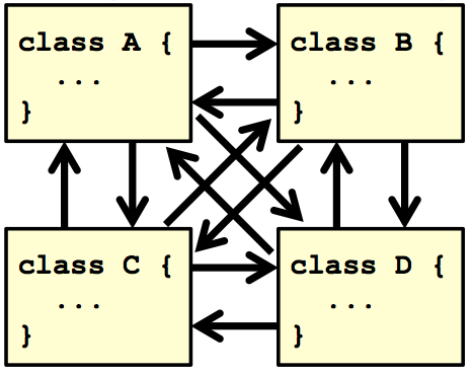
I Tirsdags

Efter denne uge skal du kunne...:

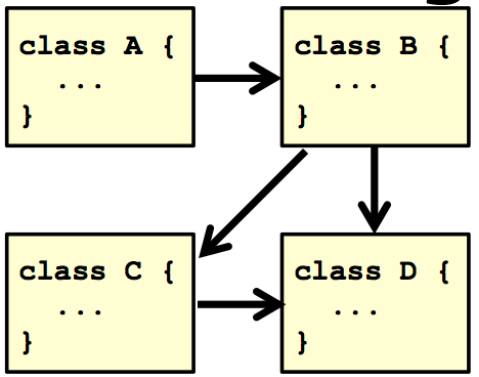
- ***Identificere*** og ***forbedre*** flg. eksempler på dårlig kode:
  - ***Kode-duplikering***
  - ***Høj kobling***
  - ***Lav sammenhæng***
- ***Refaktorisere*** et programdesign mhp udvidelse

Kobling

Høj kobling:



Lav kobling:



- **Mål for hvor tæt forbundne klasser er!**
- En vis grad af kobling er nødvendig
- Lav kobling er ønskværdig
- Problemer med høj kobling:
  - Ændringer og fejlretning er svære at lokalisere
  - Programmer bliver uoverskuelige
  - Alt involverer hurtigt mange forskellige klasser
  - Mange afhængigheder

Ansvars-drevet design

- **Hver klasse har et ansvar**
- Ansvar kan handle om:
  - at vide ting
  - at gøre ting

- ***"Each class should be responsible for handling its own data"***

- **Fx:** I eksemplet havde **`Room`** ansvar for at kende sine udgange og bør derfor også have ansvaret for fx at liste dem

Sammenhæng (cohesion)

- ***Høj sammenhæng for klasser*** betyder at hver klasse har ***et velafgrænset og sammenhængende ansvarsområde***
- En klasse bør svare til netop én type entitet

- ***Høj sammenhæng for metoder*** betyder at hver metode gør netop én ting

- ***Konsekvenser af høj sammenhæng:***
  - Øget læselighed
  - Bedre mulighed for kode-genbrug
  - Det metoden gør burde kunne afspejles i navnet
  - Højere design-stabilitet

A G E N D A

- **Testing:**
  - Motivation & Psychology of Testing
  - Bugs
  - Testing vs Debugging
  - Test cases (via opgaver)
  - Test automation (JUnit !)
- **Forebyggelse af fejl og kode-stil:**
  - Modularitet
  - Comments, JavaDoc, Indentation, Variable names
  - Eksempel: Lommeregner
- **Debugging:**
  - Manuel kodegennemgang (walkthrough)
  - Print statements
  - Debuggers

Definition: "bug"

Main entry: **²bug**
Pronunciation: /ˈbəg/
Function: *noun*
Etymology: *origin unknown*
Date: 1622

1 a: an insect or other creeping or crawling invertebrate (as a spider or centipede)
b: any of several insects (as the bedbug or cockroach) commonly considered obnoxious
c: any of an order (Hemiptera and especially its suborder Heteroptera) of insects that have sucking mouthparts, forewings thickened at the base, and incomplete metamorphosis and are often economic pests —called also true bug

2: an *unexpected* defect, fault, flaw, or imperfection <e.g., "the software was full of bugs">

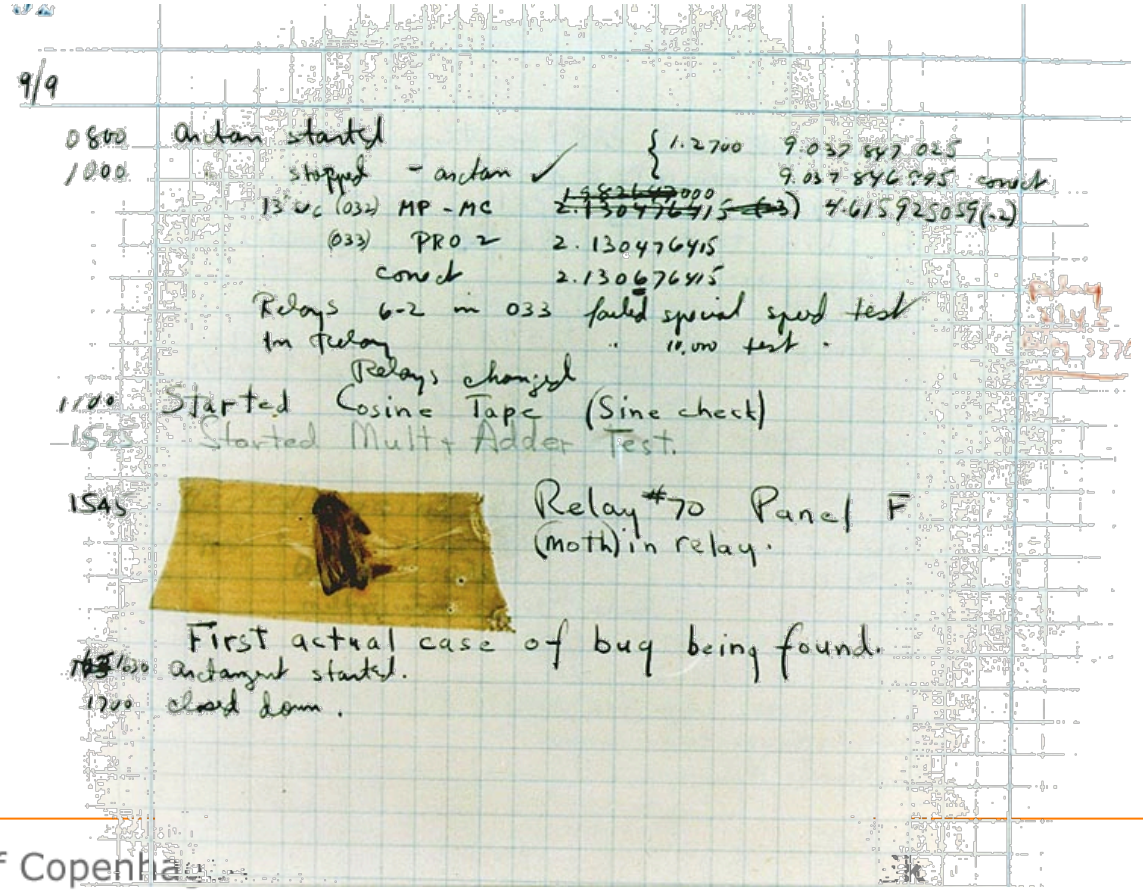
3 a: a germ or microorganism especially when causing disease b: an unspecified or nonspecific sickness usually presumed due to a bug
4: a sudden enthusiasm
5: enthusiast <a camera bug>
6: a prominent person
7: a crazy person
8: a concealed listening device
9: a weight allowance given apprentice jockeys

”The Harvard Mark II Bug”

*“The first **documented** computer **bug** was a **moth** found trapped between points at Relay # 70, Panel F, of the **Mark II Aiken Relay Calculator** while it was being tested”*

Harvard University, Sep. 9, 1947

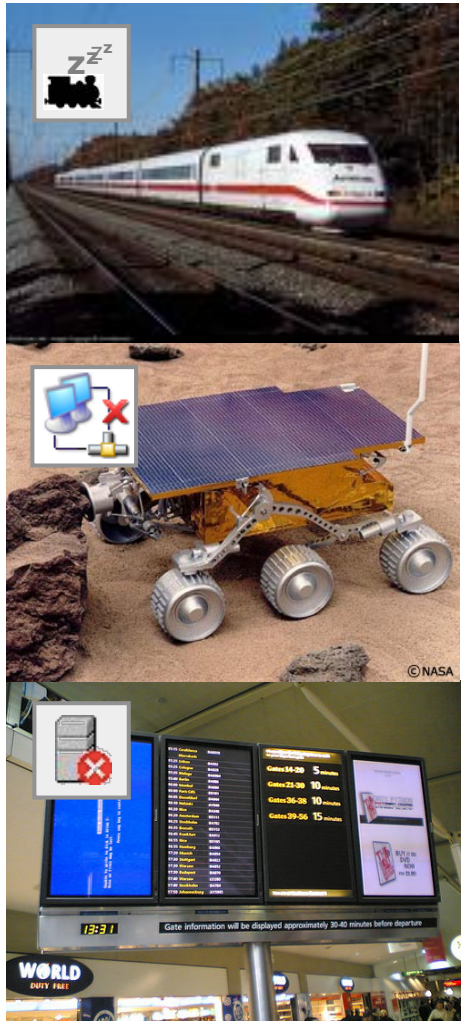
Photo of first ***actual*** ”bug”:



Software Errors

- ***Therac-25 Radiation Therapy***
  - ’85-’87
    - Concurrency bug ⇒ 6 deaths + amputations
- ***Patriot Missile Guidance System***
  - ’91 (Gulf War 1.0)
    - Accumulating rounding errors ⇒ deaths
- ***Ariane V***
  - ’96 (one of the most expensive bugs, ever)
    - 64-bit float to 16-bit int overflow ⇒ explosion

Software Errors



- ***Train Control System***
  - ’98 (Berlin)
    - Train cancellations
- ***Mars Pathfinder***
  - July ’97
    - Periodic resets **...on mars!!! :-(**
- ***Win95/98 Device Drivers***
  - late ’90es
    - Dysfunction (*“blue screen of death”*)!

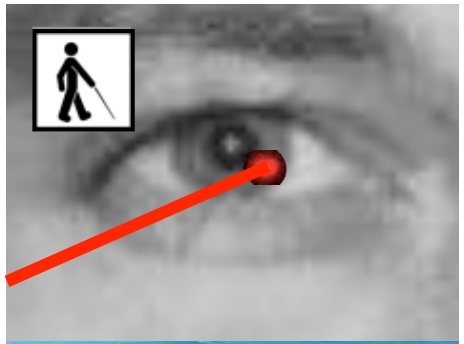
Software Errors

- ***Mobile Phones***
  - ’00
    - Freeze and odd behaviors (*really annoying*)!

- ***Cruise Control System Model***
  - ’86 (Grady Booch)
    - Accellerated after car ignition ⇒ car crashes

- ***Baggage Handling System***
  - ’94-’95 (at Denver Int’l Airport)
    - $ 360,000,000 USD

...og hvad så med ?!?



- ***Laser Eye Surgery Software?***
  - Eye damage?

- ***Aircraft Autopilot?***
  - Plane crash?

- ***Nuclear Powerplant Control System?***
  - Core melt-down?

[cf. "The Art of Software Testing" (Chap. 2), Glenford J. Myers, 1979]

Psychology of Testing

> *"**Testing** is the process of executing a program with the **intent of finding errors**."*

- **Goal:** find as many errors as possible
  - Note: realistically assumes errors are present
    - » *Constructive goal* (actually ***destructive***)

Problematic terminology:

successful test case

unsuccessful test case

vs

Much better terminology:

successful test case

unsuccessful test case

Constructive vs. Destructive Thinking

Constructive thinking:

(e.g., *programming*)

- ***"Test-to-pass"***

- Often not a good idea to "test" your own code :-(

Destructive thinking:

(e.g., *testing*)

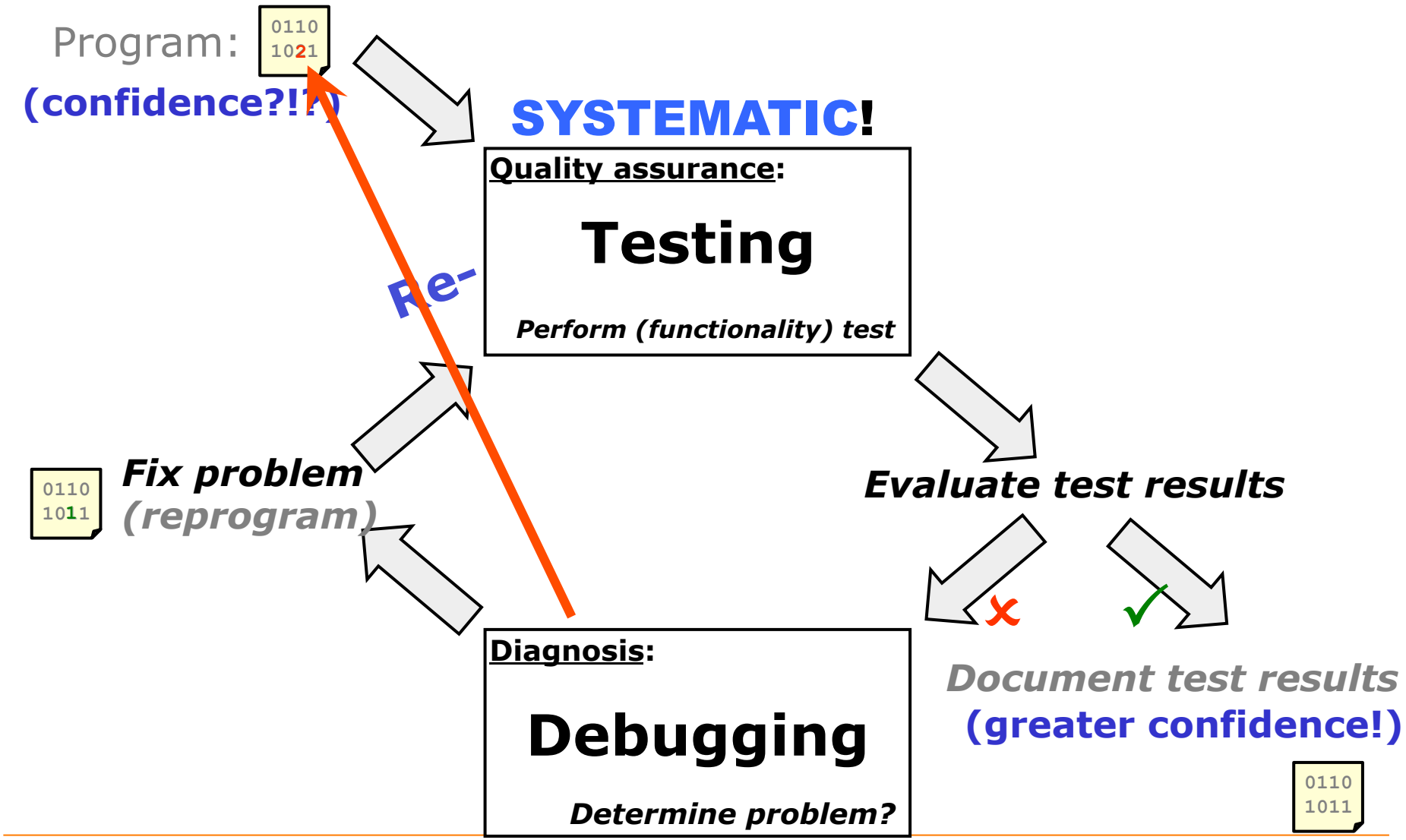
- ***"Test-to-fail"***

- Often better to test/break someone else's code :-)

Recommendation:

- Have someone else **test** (as in take a hammer to) your software!

Testing vs Debugging



Testing: Incomplete Process

Testing is an incomplete process!

- A program has:
  - ∞ many possible ***valid*** inputs (inkl: tilstand, tid, ...)
  - ∞ many possible ***invalid*** inputs

- Hence:

Testing can never prove absence of errors!

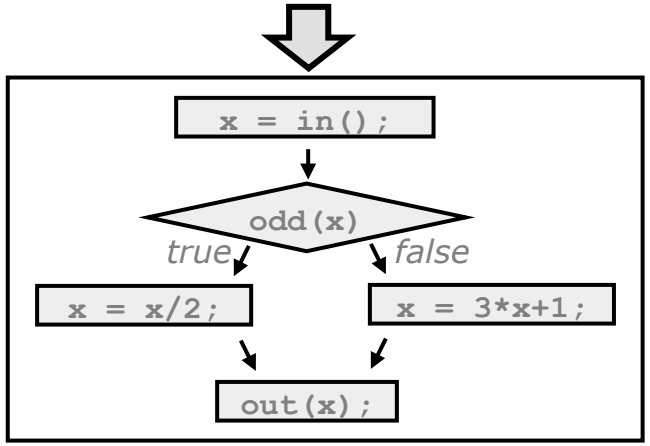
White-box vs. Black-box Test

White-box Testing:

- (aka., *"structural testing"*)
- (aka., *"internal testing"*)

- Test focus: ***src code:***

```
x = in();
if (odd(x)) {
   x = x/2;
} else {
   x = 3*x+1;
}
out(x);
```

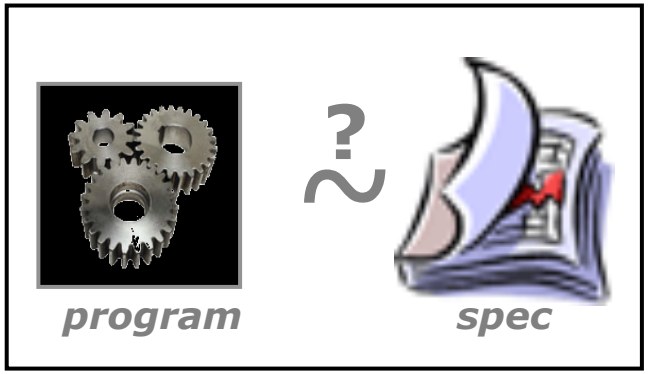


Black-box Testing:

- (aka., *"behavioral testing"*)
- (aka., *"external testing"*)
- (aka., *"input-ouput testing"*)

- Test focus: ***spec:***

or:
"intention"

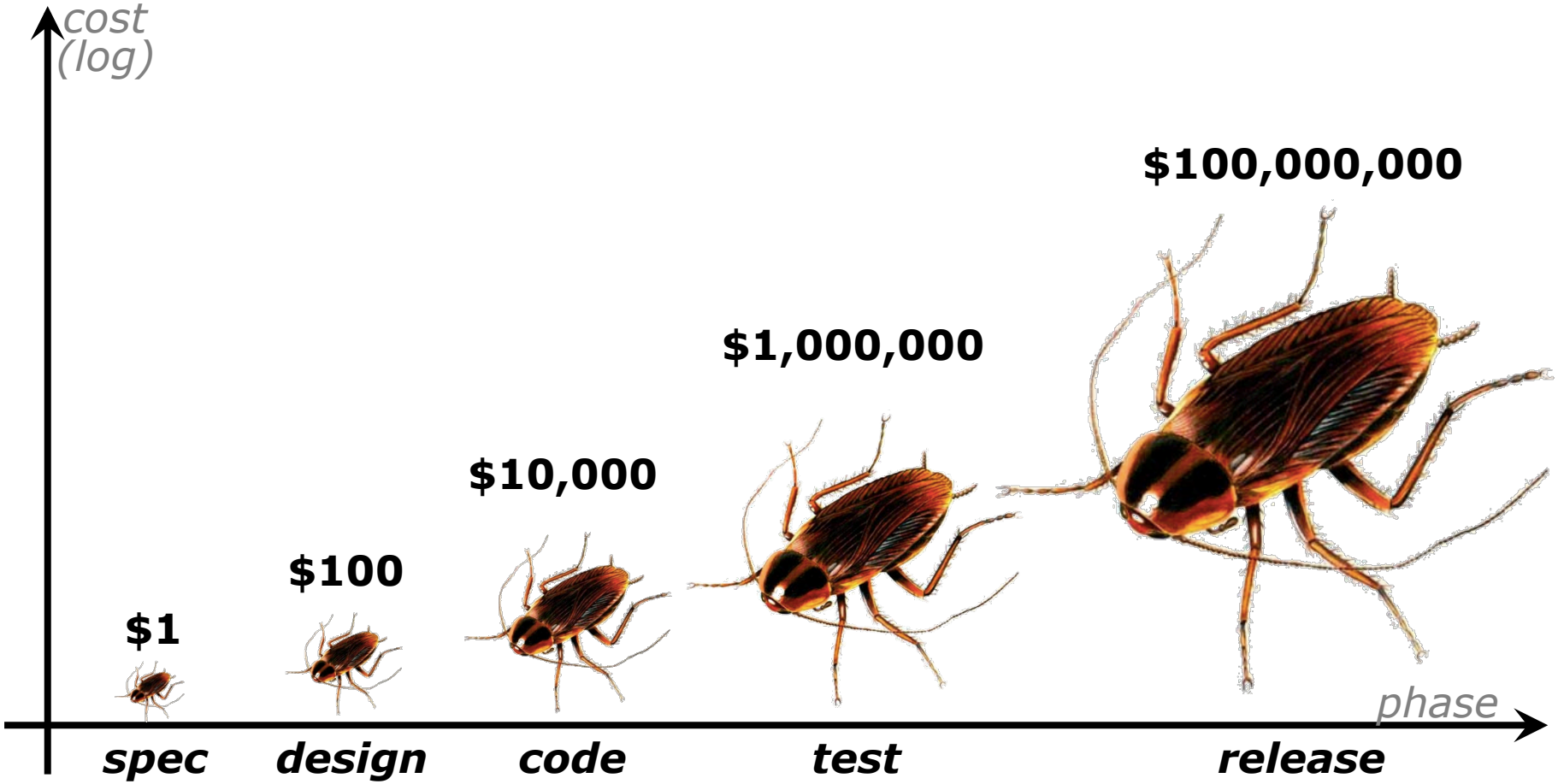


Complementary Approaches!!!

[cf. *"Software Testing"*, R.Patton, p.18]

Cost of (Fixing) Bugs

- Cost of bugs increases ***exponentially*** (over time):



”The Pesticide Paradox”

- ***”The pesticide paradox”:***

> *”The more you test a software,*
> *the more **immune** it becomes **to your tests**”*
>
> B.Beizer, *“Software Testing Techniques”*, 1990

Different 'kinds' of errors

- ***Syntactic*** errors:
  - Mal-formed program:

```
int square(int x) {
  return x*x
}
```
```
*** syntax error at line 2
    ';' expected
```

- ***Semantic*** errors:
  - ***Symbol*** errors
  - ***Type*** errors
  - ...

```
int square(int x) {
  return n*n;
}
```
```
*** symbol error at line 2
    undefined variable 'n'
```

```
int square(float x) {
  return x*x;
}
```
```
  *** type error at line 2
      fun' returns float, not int
```

- ***Logical*** errors:
  - Compiler: ”no errors”

```
int square(int x) {
  return x+x;
}
```
```
no errors found!!!
```

Fejl i programmer

- **Alle programmer har fejl !**

- Typer af fejl:
  - ***Syntaktiske fejl*** (compileren finder **alle** disse)
  - ***Semantiske fejl*** (compileren finder **nogle** af disse)
  - ***Logiske fejl*** (compileren finder **IKKE** disse)

- **Testing** har til formål at finde fejl

- **Debugging** har til formål at finde årsagen til fejlene samt rette dem

Q?

- **Hvad gør I for at undgå fejl i *jeres programmer* ?**

Typer af tests

Unit test:

- Tester ***enkelte dele*** af programmet
  - Unit = ***method*** (or ***class*** (or ***package***))
  - God til at fange fejl på tidligt tidspunkt
  - God til at lokalisere fejl

System test: (aka, "Application test")

- Tester at ***samlet program*** gør hvad det skal
  - God til at *sandsynliggøre* at programmet virker
  - Ikke nødvendigvis god til at lokalisere fejl :-(
  - Fejl findes desværre ofte sent :-(

Test som spil

Programmør vs tester:

- Tester skal bevise at der er fejl
- Test aggressivt

- Brug træning som programmør til at "gætte fejl" ("evil input" = -1, 0, 256, ...)

- I praksis er tester og programmør ofte en og samme person, men…

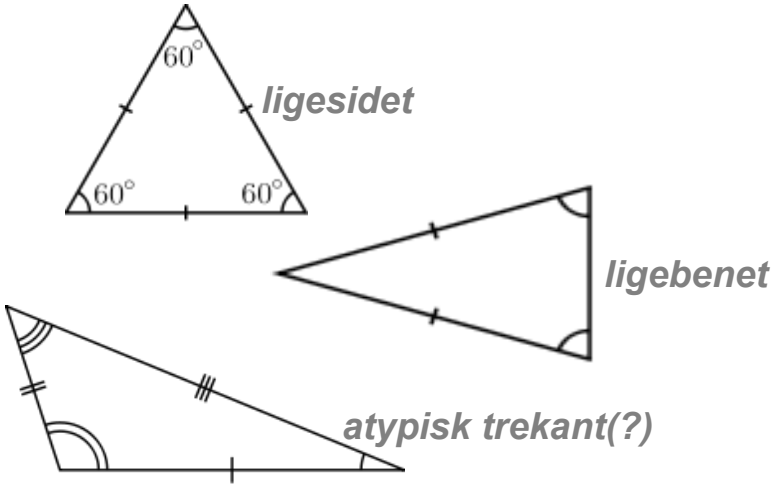
Unit test som del af udvikling

- **Test hver metode, når den skrives!**
  - **Husk:** Tidlig fejlfinding sparer tid!
  - (Faktisk kan man skrive test ***før*** metode)

- **Regression test:**
  - Del-programmer der har bestået test kan fejle test senere
  - Fejl-rettelser (inkl. refactoring) kan føre til nye fejl
  - Test igen senere i udvikling af program
  - Test igen ved tilføjelse af funktionalitet
  - Test igen ved vedligehold af programmet

EXERCISE: Triangle Test Test

- ***Equilateral triangle (T-3):***
  - All three sides have equal length
- ***Isosceles triangle (T-2):***
  - Two sides have equal length
- ***Scalene triangle (T-1):***
  - All sides have different length




```
public enum Triangle { EQUILATERAL, ISOSCELES, SCALENE, NOT_A_TRIANGLE; }
public Triangle isTriangle(int a, int b, int c) { ... }
```

The program reads ***three integer values*** from an input dialog.
(The three values represent the lengths of the sides of a triangle.)
The program displays a message that states whether the triangle is:
- **`equilateral`**, **`isosceles`**, or **`scalene`**

Q: Which test cases should we use?
- E.g., (1,1,1), (2,2,2), (3,3,3), (4,4,4), (5,5,5), (6,6,6), (7,7,7), …?

Appropriate Test Cases?

- ***Representativity?***

SYSTEMATIC TESTING !

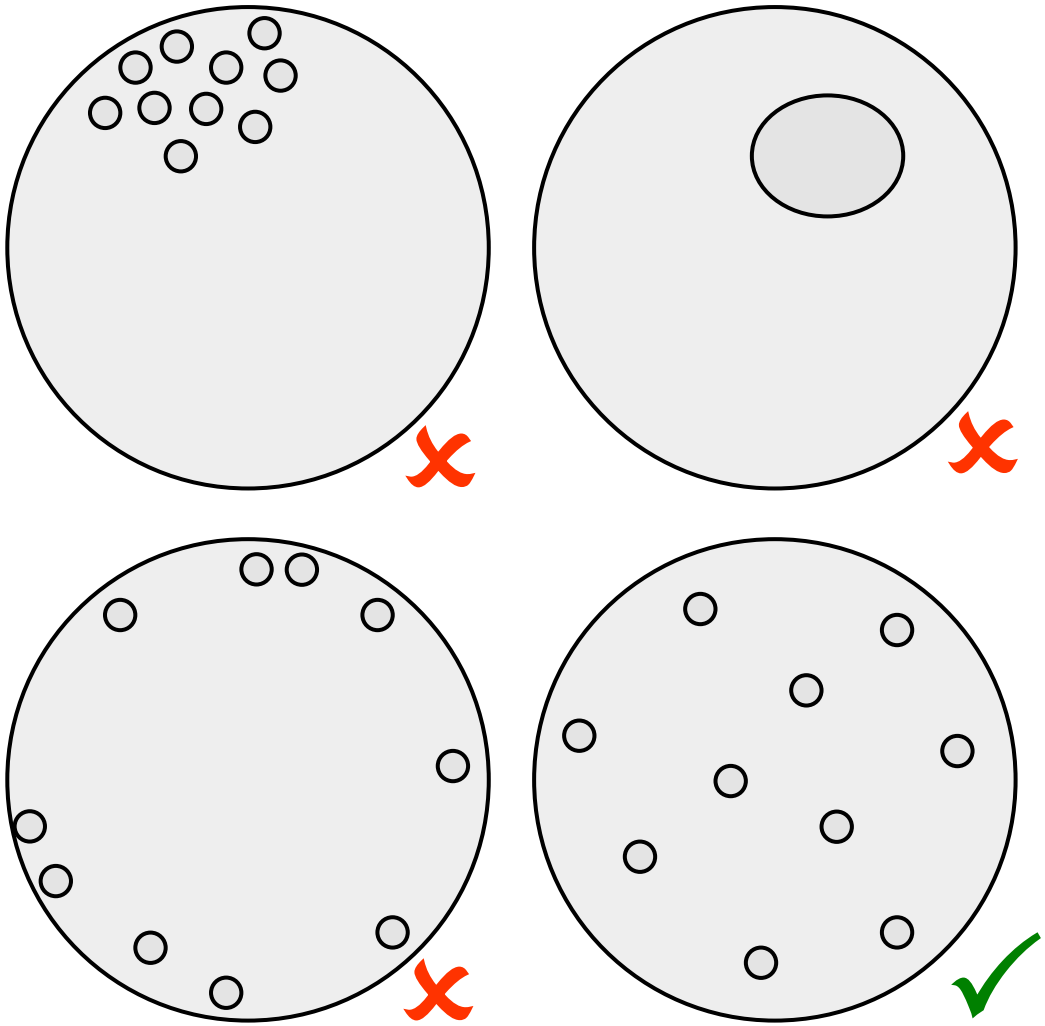
Advice:
Avoid making lots of ad-hoc tests "just to be on the safe side".

Advice:
Cases relative to how we ***might*** conceive solving the problem & how it ***might*** be wrong.
(This involves *"guessing"* how implementations ***might*** work)

Advice:
Carefully chose cases such that:

- same case => same error; *and*
- same error => same case.

Advice:
Test ***typical*** and ***extreme*** cases.



Design af unit test

- **Unit test i Java:**
  - Test hver metode for sig

<u>Systematisk tilgang til test</u>:

- Test af grænsetilfælde:
  - Nul, en, mange. Især ved løkker
  - Tomme collections, collections med mange elementer
- Positive og negative tests:
  - Test tilfælde der bør gå godt
  - Test tilfælde der bør fejle

Test ”Boundaries”

- Programs are vulnerable ”around the edges”:
  - e.g. testing legal inputs (time, in hours):

Property	Input	Expected output	Actual output
Minimum-1	-1	invalid	
Minimum	0	valid	
Typical	15 (e.g.)	valid	
Maximum	23	valid	
Maximum+1	24	invalid	

  - e.g. testing legal inputs (dates, in April):

Property	Input	Expected output	Actual output
Minimum-1	00/4	invalid	
Minimum	01/4	valid	
Typical	17/4 (e.g.)	valid	
Maximum	30/4	valid	
Maximum+1	31/4	invalid	

Test "Powers-of-Two"

- Programs vulnerable "around powers-of-2":
  - e.g. years of age (assume held in a ***byte***):

Property	Input	Expected output	Actual output
Minimum-1	-1	invalid	
Minimum	0	valid	
Typical	27 (e.g.)	valid	
Maximum	255	valid	
Maximum+1	256	invalid	

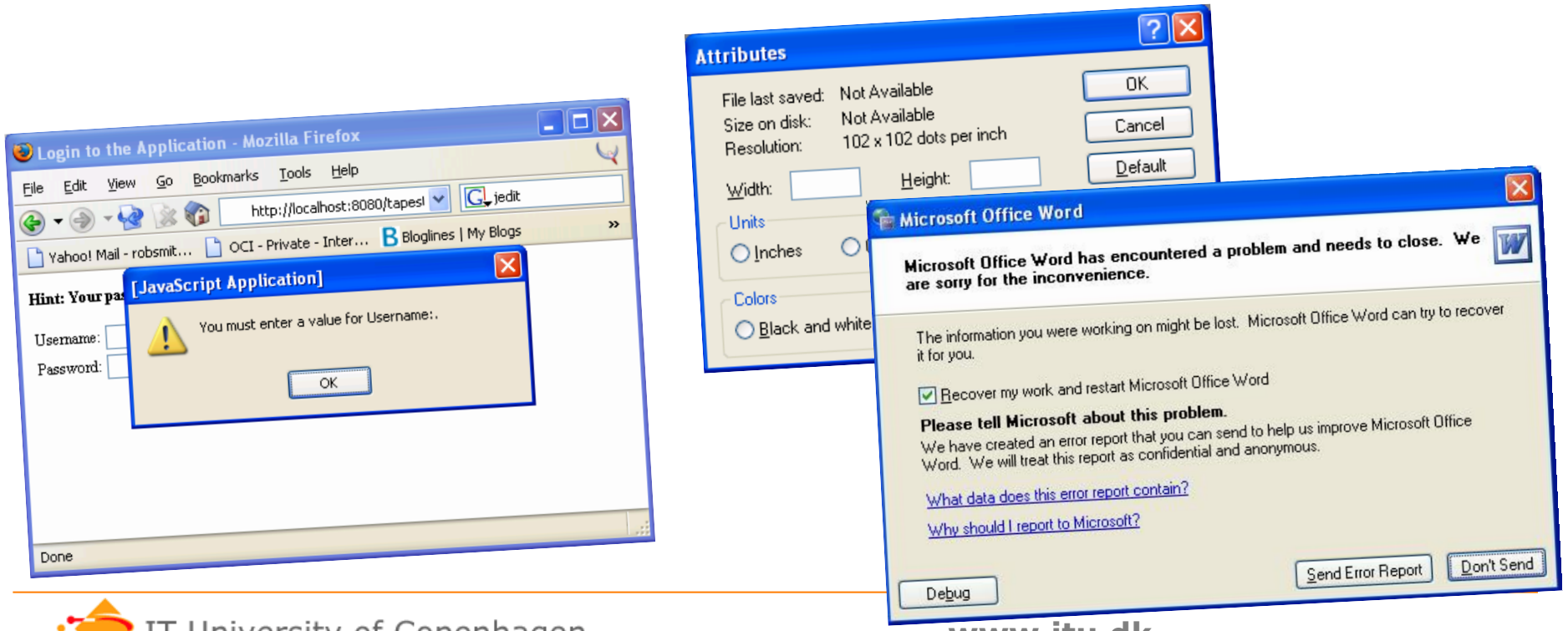
  - e.g. #game-spectators (assume held in a ***16-bit word***):

Property	Input	Expected output	Actual output
Minimum-1	-1	invalid	
Minimum	0	valid	
Typical	12345 (e.g.)	valid	
Maximum	65535	valid	
Maximum+1	65536	invalid	

Test "Empty Input"

- Default / empty / blank / null / zero / none:
  - e.g., *'any program'*:

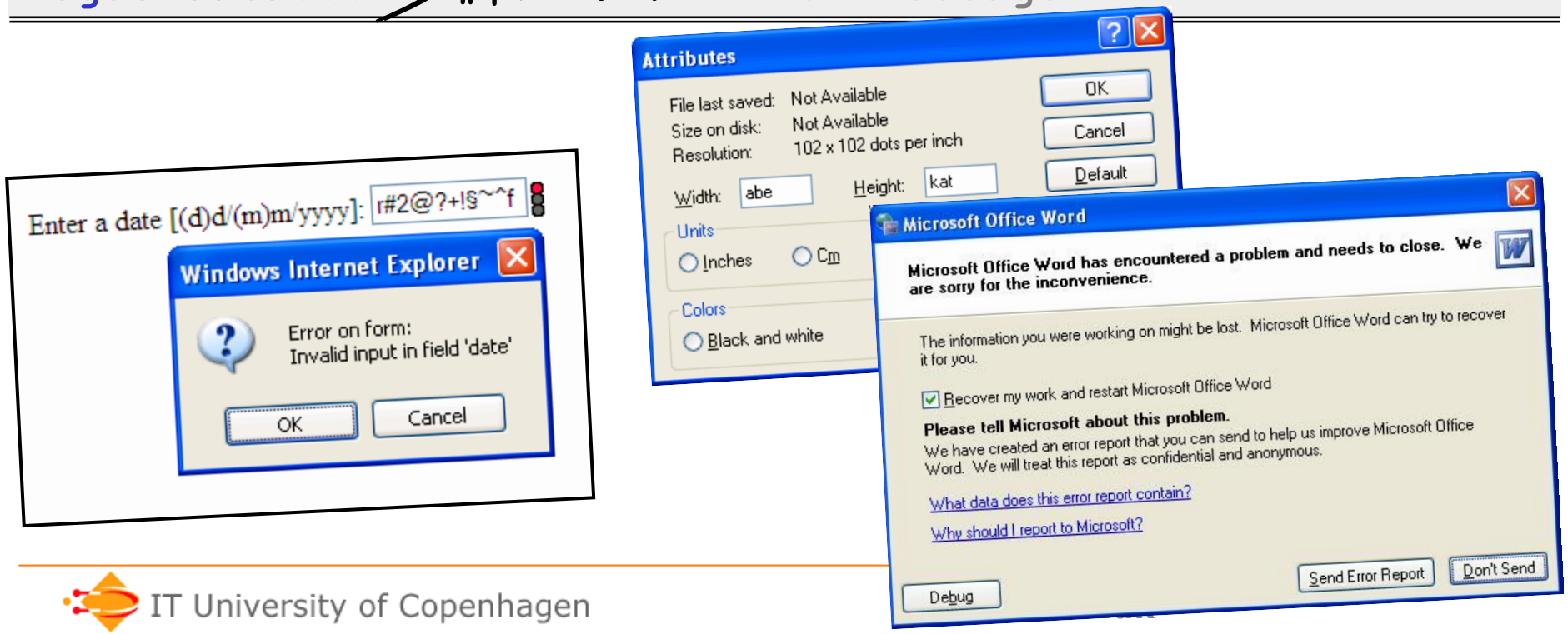
Property	Input	Expected output	Actual output
No input	(none)	Error message	



Test "Invalid Input"

- Invalid / illegal / garbage / bogus data:
  - e.g., calculator:

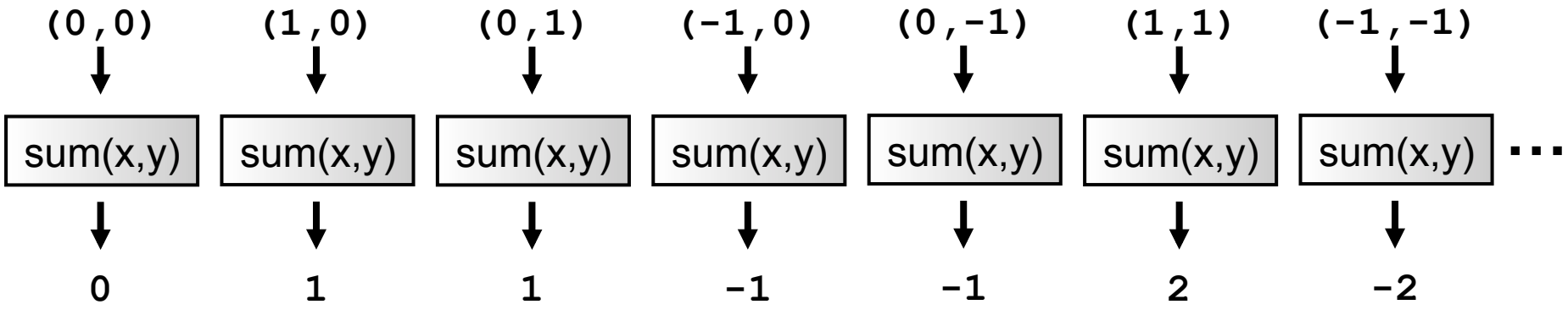
Property	Input	Expected output	Actual output
Invalid input	+*31	Error message	
Bogus data!!!	#$+~´?=!	Error message	



Testing: Infinite process

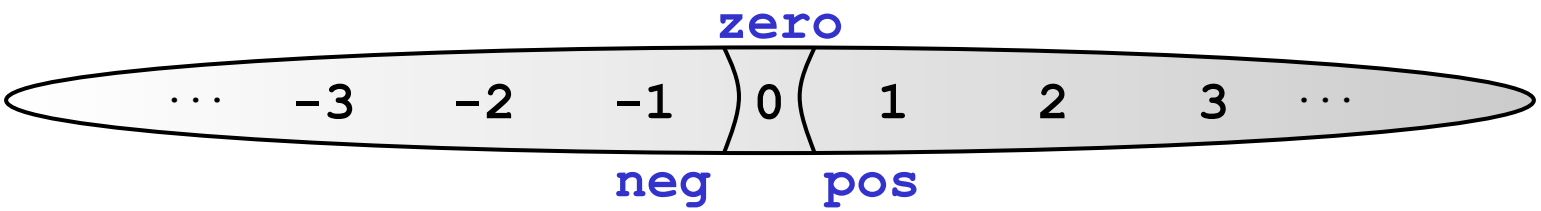
- **Recall:** ”testing is an incomplete process”
  - (in particular: ”testing can’t prove absence of bugs”)

- There are ***infinitely*** many possible inputs:
  - (hence: testing will take an ***infinite*** amount of time)



Equivalence Partitioning

- **Partition input:**



- **Finitary partition:**
  - If finite # categories (aka. *”equivalence classes”*)
    - » Now only three: { ”`zero`”, ”`pos`”, ”`neg`” }

- We can now test ***all equivalence classes***
  - Using ***representative*** elements from each category

Test Sum (cont'd)

- We can now **test *all*** ***equivalence classes***
  - Using ***representative*** input from each category

- Sum (testing ***all*** equivalence classes):

3*3 = 9

Property	Input	Expected output	Actual output
Pos, Pos	(1,2)	3	
Neg, Pos	(-3,4)	1	
Zero,Pos	(0,5)	5	
Pos, Neg	(6,-7)	-1	
Neg, Neg	(-8,-9)	-17	
Zero,Neg	(0,-10)	-10	
Pos, Zero	(11,0)	11	
Neg, Zero	(-12,0)	-12	
Zero,Zero	(0,0)	0	

Frequent Partitions for Testing

- **Numbers:**
  - positive, negative, zero
  - zero (0), one (1), many (2+) aka., ”Eskimo Numbers”
- **Lists:**
  - length-0, length-1, length-2+
  - ascending-elements, descending-elements, un-sorted
- …

Advice:
Consider how problem ***might*** be solved
Partition into ***qualitatively different*** categories such that:
- ”same case ⇒ same error”; *and*
- ”same error ⇒ same case”.

EXERCISE: Min-Max

Specification: ”Min-Max”
*The program receives some non-negative numbers as arguments; finds the **minimum** and **maximum** among those, and prints the results*

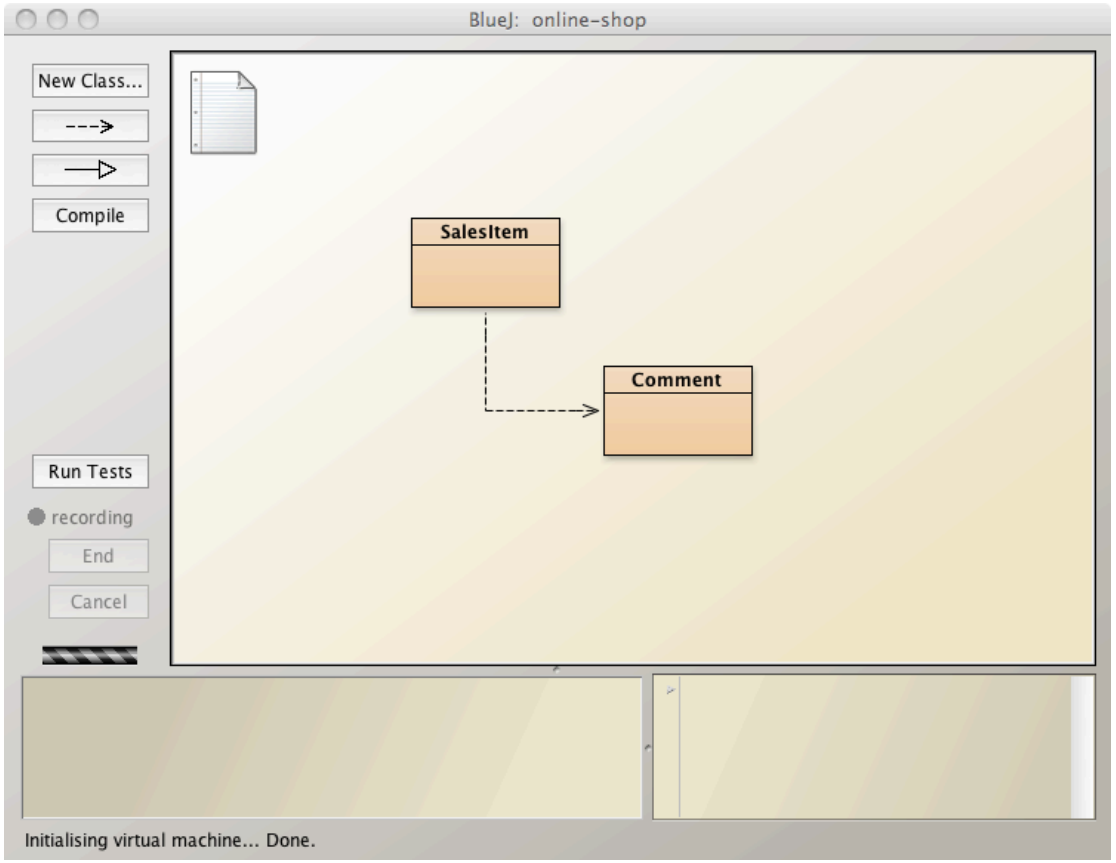
EXERCISE: Sorting

Specification: ”Sorting”
*The program takes a list of positive numbers, **sorts them**, and prints the result.*

EXERCISE: Insert-into-Sorted-List

Specification: ”Insert-into-sorted-list”
The method takes a positive integer `x` *and a list of sorted positive ints* `L` *and* ***inserts*** `x` ***into*** `L`*, yielding another sorted list.*

Online-shop


BlueJ: online-shop
New Class...
Compile
SalesItem
Comment
Run Tests
recording
End
Cancel
Initialising virtual machine... Done.

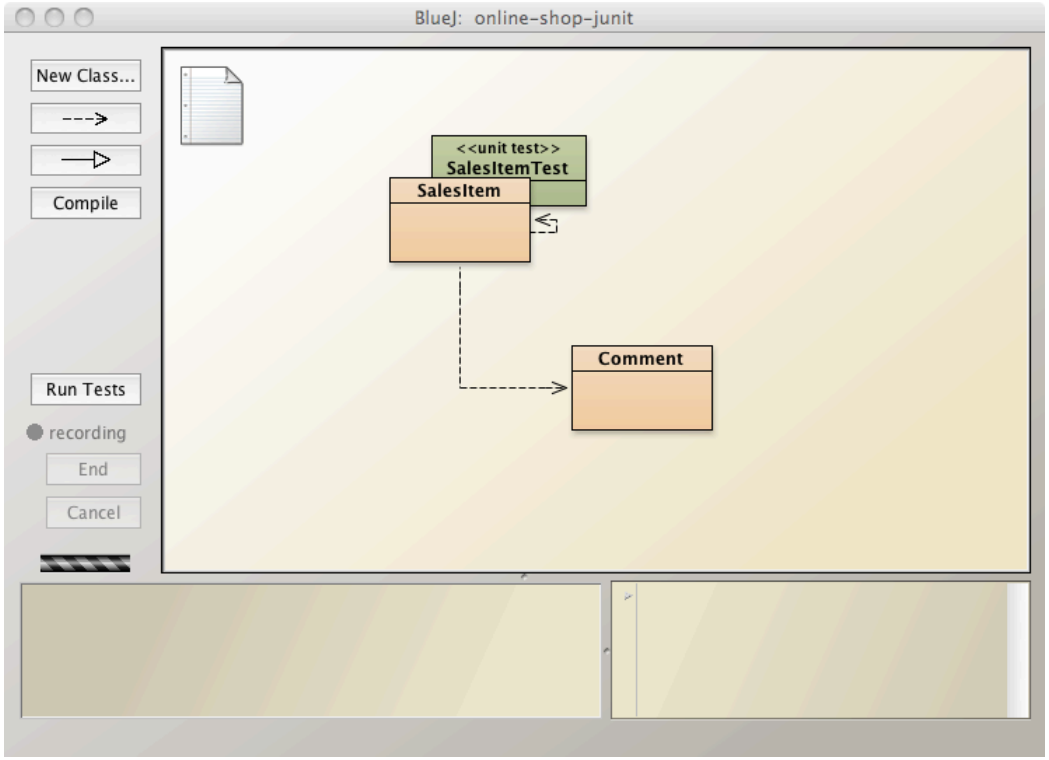
Manual Testing

- Hvad skal der testes for I SalesItem

Test automatisering

- **Manuelle test er besværlige:**
  - De tager tid at udføre
  - Booooooooooring! :-/
  - Man skal vide hvad det forventede resultat er (skal dokumenteres)
- **JUnit** værktøj til **"test automation":**
  - Skriver programmer der tester programmet
  - Testprogram afgør om test gik godt
- Resultat: Dobbelt kode
- Skriv test-programmet **sammen med programmet**

Test automatisering med JUnit



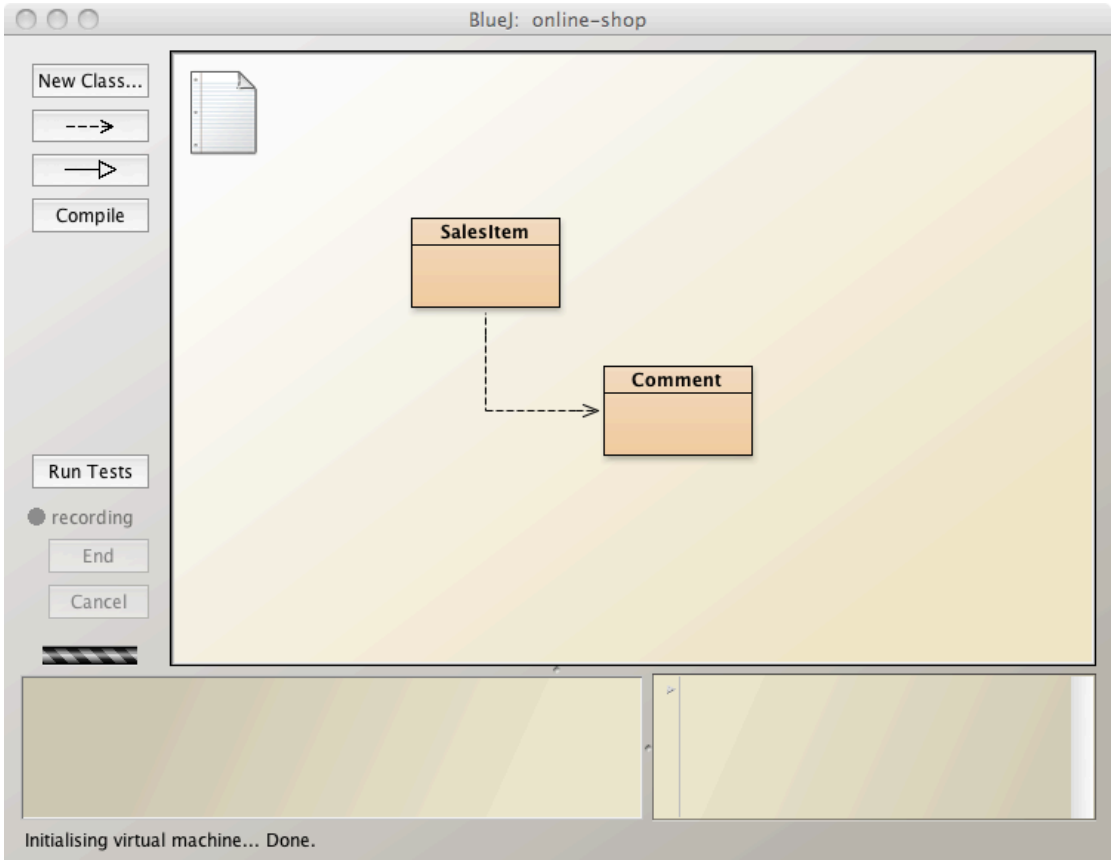
Test initialisering

```
@Test
public void testInit() {
    SalesItem si = new SalesItem("test name", 1000);
    assertEquals("test name", si.getName());
    assertEquals(1000, si.getPrice());
}
```

Test for at tilføje en kommentar

```
@Test
public void testAddComment() {
   SalesItem si = new SalesItem("Java for complete Idiots", 21998);
   assertEquals(true, si.addComment("James Duckling",
                                    "This book is great.",
                                    4));
   assertEquals(1, si.getNumberOfComments());
 }
```

Optagelse af test sekvens



@Before & @After (fixtures)

```
class SalesItemTest {
   SalesItem si;

   @Before // Called before every test case method
   public void setUp() {
      si = new SalesItem("test name", 1000);
   }

   @Test
   public void testInit() {
      assertEquals("test name", si.getName());
      assertEquals(1000, si.getPrice());
   }

   @After // Called after every test case method.
   public void tearDown() {
      si = null;
   }
}
```

Debugging

Find cause of error and fix it

Modularitet, idé

- Større konstruktioner deles op i mindre dele (moduler)
- Eksempel: Biler består af mange dele, f.eks. motor, gearkasse, styretøj etc.
- Enkelte moduler kan konstrueres uafhængigt
- Enkelte moduler kan udskiftes

Modularitet i software

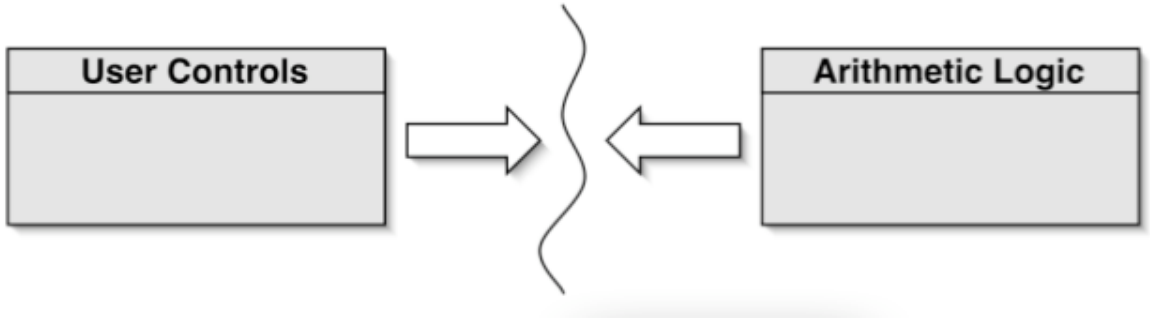
- Et modul kan f.eks. være en klasse
- Hver klasse har en veldefineret grænseflade (interface)
  - Hver klasse skal opfylde en "kontrakt"
- Forsimpler udvikling
  - Projektet kan deles op i mindre dele
  - Programmering kan uddelegeres
- Enkelte dele kan udskiftes
  - Øget fleksibilitet
- Vedligehold er nemmere

Grænseflader (interfaces)

- Hver modul (klasse) skal opfylde en kontrakt
- Kontrakten kan indeholde
  - en liste af metode signaturer
  - krav til hvad disse metoder skal gøre
- Det første kan verificeres af oversætteren, det sidste kan ikke

Eksempel

- Lommeregner



- De to objekter kan skrives uafhængigt af hinanden
- User controls kan være forskellige former for brugergrænseflade

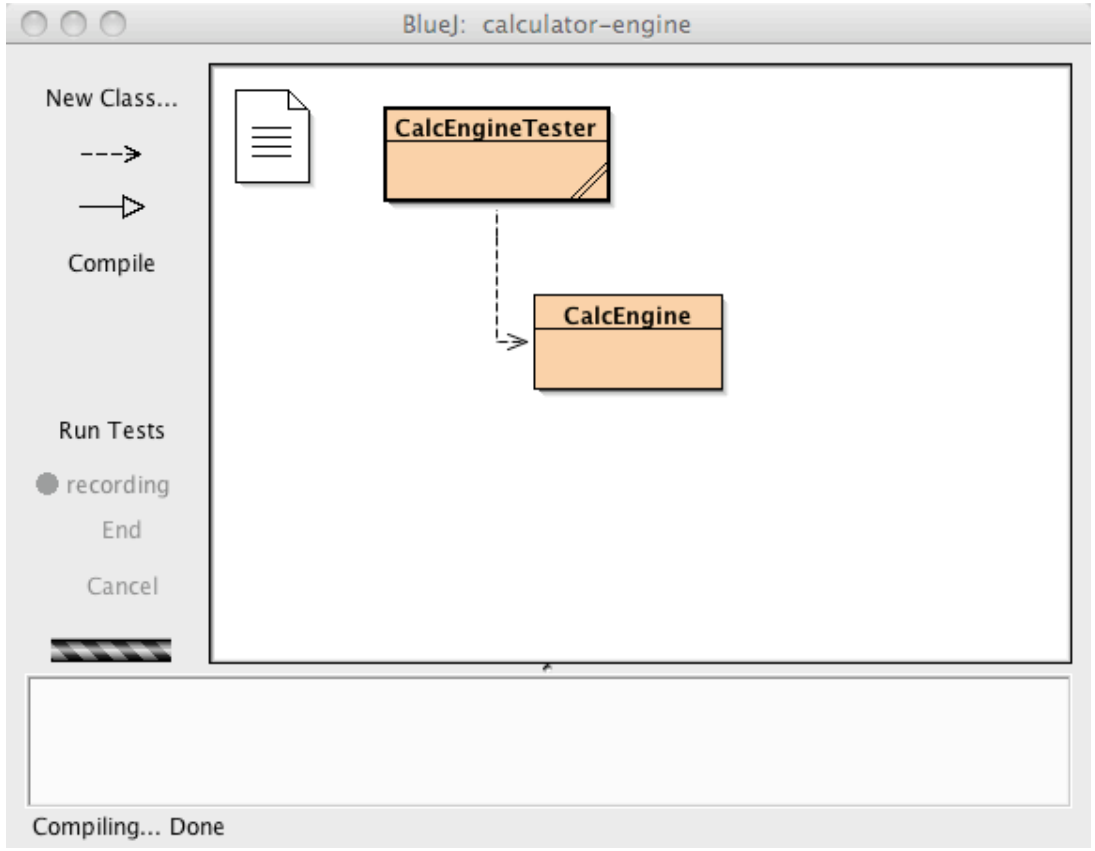
Lav et interface før implementation

```
// Return the value to be displayed.
public int getDisplayValue();

// Call when a digit button is pressed.
public void numberPressed(int number);

// Call when a plus operator is pressed.
public void plus();

// Call when a minus operator is pressed.
public void minus();

// Call to complete a calculation.
public void equals();

// Call to reset the calculator.
public void clear();
```

Modularitet og unit test

- Unit test kræver modularitet
  - Man skal kunne teste hver enhed for sig
  - Der skal være en kontrakt man kan teste
- Modularitet gør det lettere at lokalisere fejl
- Modularitet forebygger fejl
  - Programmør kan koncentrere sig om delopgave

Calculator-engine eksemplet



Manuel gennemgang af kode

- Skriv kode ud og gå væk fra computer
  - Low tech og undervurderet
  - Gennemgå kørsel linie for linie
- Hold styr på tilstand (state)
  - Hvilke værdier har hvilke variable?
- Kan også gøres mundligt over for andre
  - Ofte finder man selv fejlen

Manuel gennemgang

`testPlus()`:

- Manuel gennemgang af ***kode*** (papir)
  - *"High-level code walkthrough"*
- Manuel gennemgang af ***tilstand*** (BlueJ)
  - *"State-inspection walkthrough"*
- Manuel gennemgang for ***person*** (Homo Sapiens)
  - *"Walkthrough and explain to someone"*

Kodestil og debugging

- Dårlig kodestil kan gøre kode ulæselig og svær at debugge

- Eksempler på dårlig kodestil:
  - Dårlig indrykning (indentation)
  - Intetsigende variabelnavne (**v1**, **v2**, **v3**)
  - Manglende kommentarer
  - ...

```
int money = 100;
if (money < 10000)
   if (money < 0)
      System.out.println("Whoa, I'm broke!");
else
   System.out.println("Hey, I'm rich - let's paaaarty!");
```

Print statements

- Indsæt prints i kode:
  - Midlertidigt printing:

```
System.out.println("called X");
```

  - Print variabel-værdier:

```
System.out.println("x = " + x);
```

- Meget anvendt:
  - Enkel metode
  - Kan bruges i alle udviklingsmiljøer
- Kan føre til rod:
  - Information overload?
  - Husk at fjerne print statements igen!
  - Men hvad hvis vi skal bruge dem igen?

Variationer

- Boolean variabel indikerer om der debugges:

```
if (debugging) {
   System.out.println("NumberPressed called with " + number);
}
```

- Særlig print funktion for debugging

```
/** prints message 'msg' if boolean flag 'debugging' is enabled */
public void printDebugging(String msg) {
   if (debugging) System.out.println(msg);
}
```

```
printDebugging("NumberPressed called with " + number);
```

- NB: Print i stedet til "log-fil"

Brug af debugger

- Indbygget i BlueJ
- Automatiseret kodegennemgang
- Holder styr på
  - Værdier af variable
  - Kaldestak
- Ingen oprydning nødvendig
- Men man kan ikke gå baglæns

Opsummering: Testing

- Test skal finde de værste fejl
- Systematisk testing!
- Test kan bruges som dokumentation for at (sandsynliggøre at) programmet virker
- Unit test bør skrives samtidig med programmet
- Programmet testes
  - Under udvikling
  - Under vedligehold
  - Efter fejlretning
- JUnit gør det muligt at automatisere tests

Opsummering: Debugging

- Modularisering er godt
  - Forenkler udvikling
  - Forhindrer fejl
  - Gør det lettere at lokalisere fejl
- Brug forskellige metoder til debugging
  - Manuel kodegennemgang
  - Forklar kode til andre
  - Print statements
  - Debuggers
- **Quote: *"In practice, we would use different strategies at different times"***

Tak

Spørgsmål?

Husk: Obl.Opg. **B** (deadline: Oct 7)