Danmarkskort: Visualisering, Navigation, Søgning og Ruteplanlægning

Lecture 1: Introduction and white box testing

Troels Bjerre Sørensen

Today's lecture

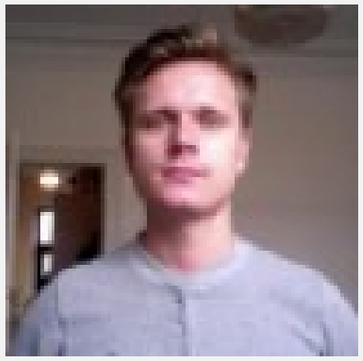
- Introduction
  - Teaching staff
  - Course goals
  - Hand ins and exam
  - Plan for the course

- White-box testing
  - Purpose of testing
  - Coverage criteria
  - Examples

Teaching Staff

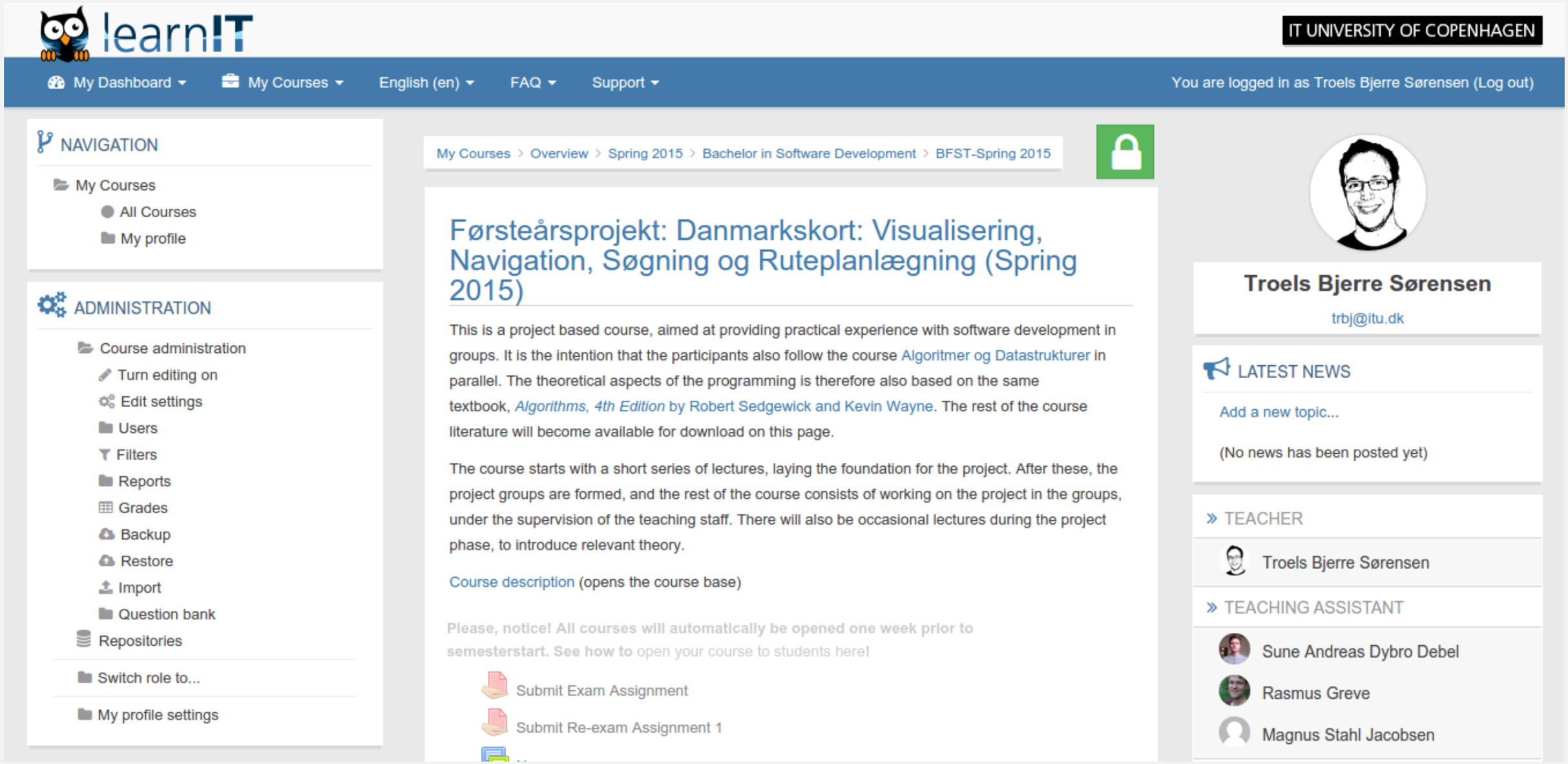
- Lecturer:
  - Troels Bjerre Sørensen
  - Assistant Professor in the TCS section
  - Office: B404

- TAs:
  - Sune Andreas Dybro Debel
  - Magnus Stahl Jacobsen
  - Rasmus Boll Greve



Course webpage

https://learnit.itu.dk/course/view.php?id=3003022



Course goals

After the course the student should be able to:

- Identify, define and delimit a problem in software development
- Identify and analyze relevant options, such as tools, technologies, data structures and algorithms
- Compare the relevant options for the application at hand, selecting the most suited ones based on their theoretical and practical performance
- Combine the selected options, develop them further, and apply them to achieve a complete solution;
- Document the project incrementally through the project diary and detailed version control log;
- **Systematically test the product**
- Evaluate the realized product with respect to the prepared problem statement;
- Reflect on the realized product and other aspects related to the project
- Explain in writing: problem, background research, steps towards the solution, the product, evaluating the product, and other relevant materials.

What is the project about?

- Visualize a map of Denmark:
- Input:
  - a text-based data set with points (Open Street Maps)
  - connections between points and other data (road names, speed limits, road directions)
- Expected output:
  - Visualization
  - Address search
  - Navigation

Course Schedule

- Week 5, 6, 7, and 8:
  - Monday 10-12: Lecture
  - Monday 12-14: Exercises
  - Tuesday 10-12: (TA session)
  - Wednesday 10-12: Hands-on lecture (weekly hand-in presented)
- Week 9: group formation and start of project
- Week 13 and 16:
  - Wednesday 10-12: Suplementary theory lectures
- Week 20:
  - Wednesday: Code freeze and hand-in of code
- Week 21:
  - Wednesday: Hand-in of report

Testing

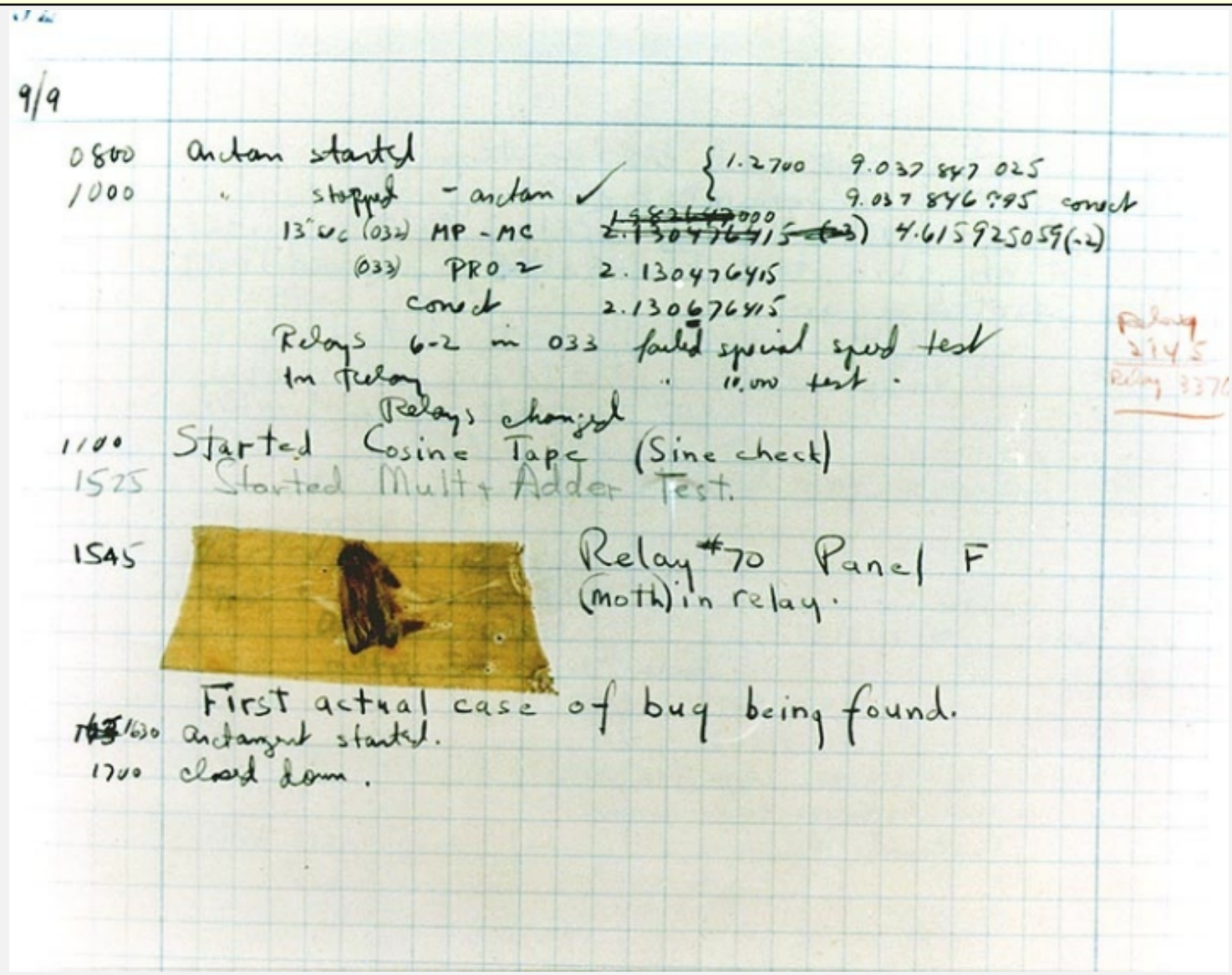
You always test your code, right?

```
[leet@h4x0rz:~/code] javac MakeMoney.java
[leet@h4x0rz:~/code] java MakeMoney
lotz'n'lotz of $$$$  lulz
[leet@h4x0rz:~/code]
```

The first bug ever found

"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



Famous software failures

Ariane 5 (1996)

- First test flight
- Code reused from Ariane 4
- 64 bit float converted to
  16 bit signed int
- During launch, the on-board nav computer tried to turn 90°
- ... tearing the rocket to pieces

Famous software failures

Therac-25 radiation therapy (1985-1987)

- Hardware safeties replaced by software
- Race conditions caused 100 times increase in radiation dose
- At least six “incidents”

Famous software failures

F-22 Raptor (2007)

- First overseas deployment
- Crossing the international date line:
  - Complete loss of navigation
  - Complete loss of communication

Program correctness is important

Nuclear powerplants

Robot assisted surgery

Airplane control systems

Driverless trains

Functionality Testing is part of QA

Covered by ISO 25010 (2011)

- In this course, we focus on functionality testing

Definition of functionality testing

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

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Not: Process of finding all bugs
- Not: Process to prove that there are no bugs
- Not: Process to prove that the code does what it is supposed to

E.W. Dijkstra:
“Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence”

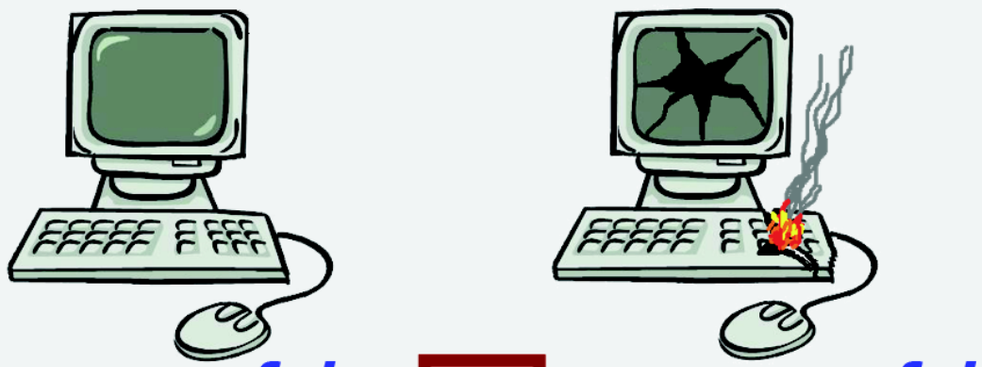
Psychology of Testing

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

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Goal: find as many errors as possible

Problematic terminology:



vs.

Much better terminology:

Constructive thinking

- Test to pass
- Testing own code

vs.

Destructive thinking

- Test to fail
- Testing someone else’s code

Not all errors are “bugs”

Error type	Example

Syntactical	`public static void main (String[] args) {`
	`  for int i = 0 ; i <= 50 ; i++ {  }`
	`}`

Semantical	`public static void main (String[] args) {`
	`  while (i <= 50) {  }`
	`}`

Logical	`double computeCircleArea(double radius) {`
	`  return radius * Math.PI;`
	`}`

Q: Which of these is hardest to find?

A: Logical errors are not found by the compiler!

Functionality testing deals with finding logical errors

Test suite

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Test case: an input to the program, and an expected output
- Test suite: the collections of all tests associated with a project

Test suite

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Test case: an input to the program, and an expected output
- Test suite: the collections of all tests as

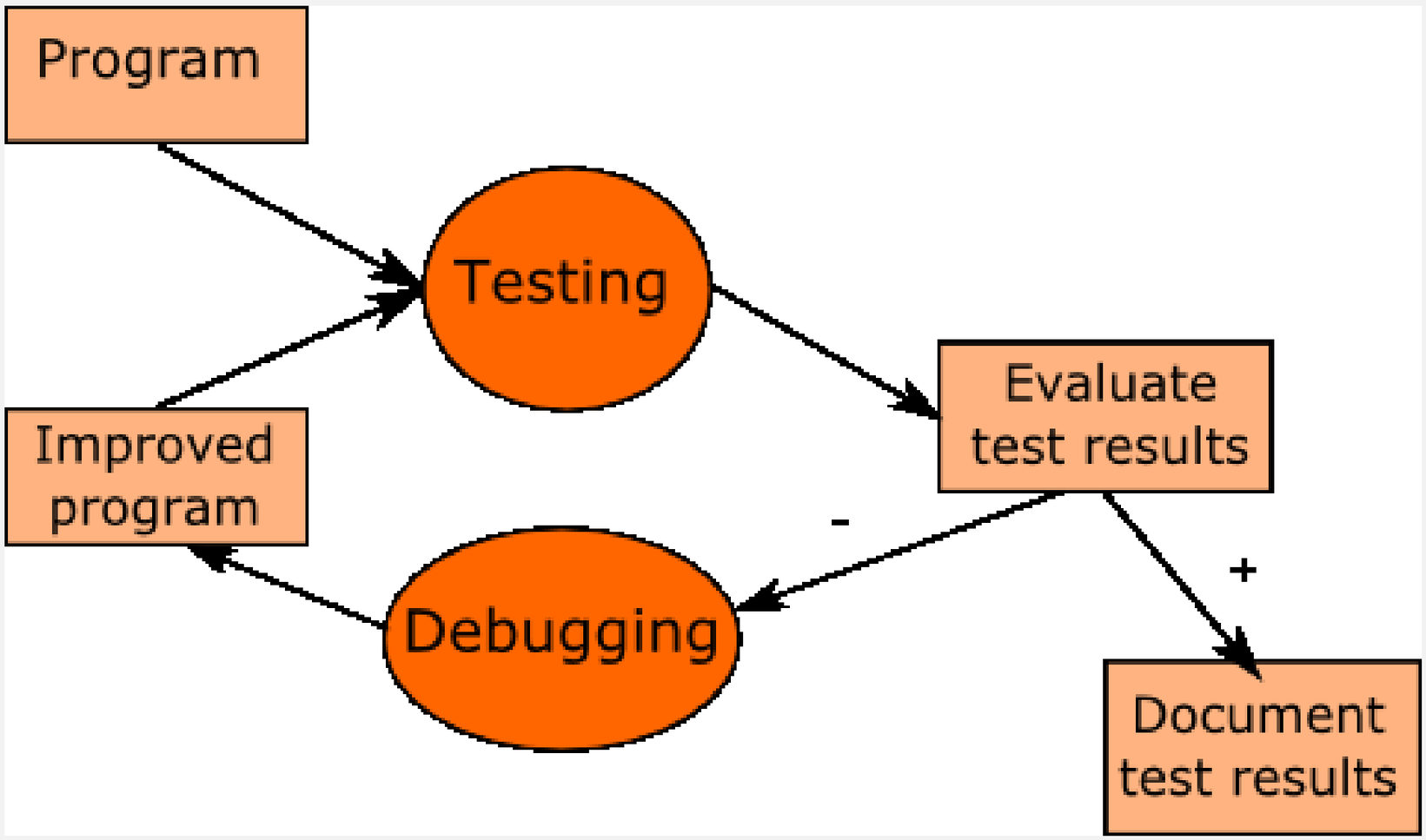
```
void m(int account, int rate) {
    account = account + 10;

    if (account > 90)    /* 1 */
        rate = rate * 2;
    else
        rate = rate / 2;

    out(“account = ” + account
        + “ rate = ” + rate);
}
```

- Test suite is an expectancy table:

Test ID	Input	Expected output
A	(98, 3)	”a=108, r=6”
B	(76, 4)	”a=86, r=2”

Testing vs. debugging

- Testing and debugging are not the same
- Testing provides the to-do list for debugging:



Testing: an incomplete process

- Most programs have:
  - Infinitely many valid inputs
  - Infinitely many invalid inputs
- Unintended behavior on any input is a bug

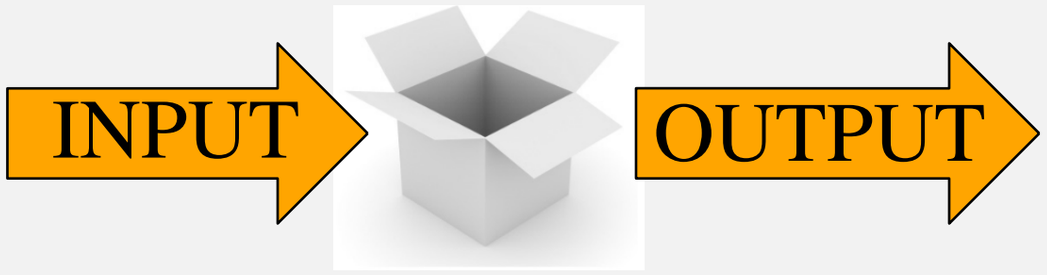
Q: How can we be sure we have tested for every bug?

A: We can't!

- Two approaches to systematic test suite construction:
  - Black box testing
  - White box testing

Complementary approaches to test suite construction

White box testing

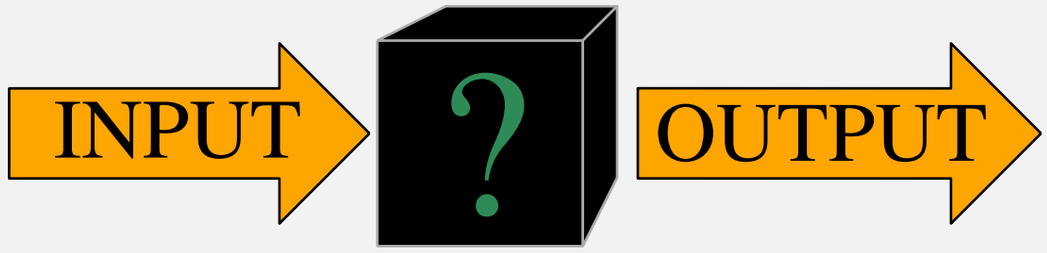


- Generate test input from Source Code:

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

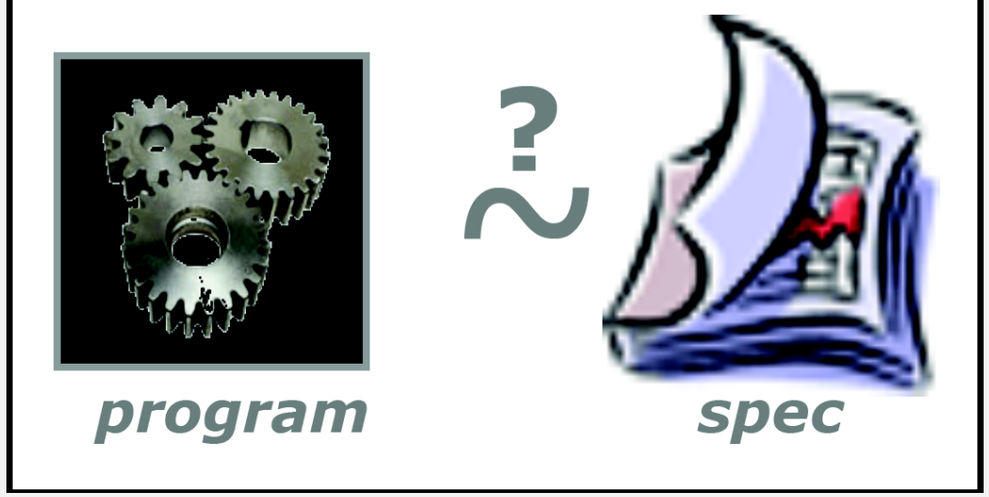

- ... then check specs for expected behavior

Black box testing



- Generate test cases from specs

> The program should compute a single step of the Collatz conjecture function

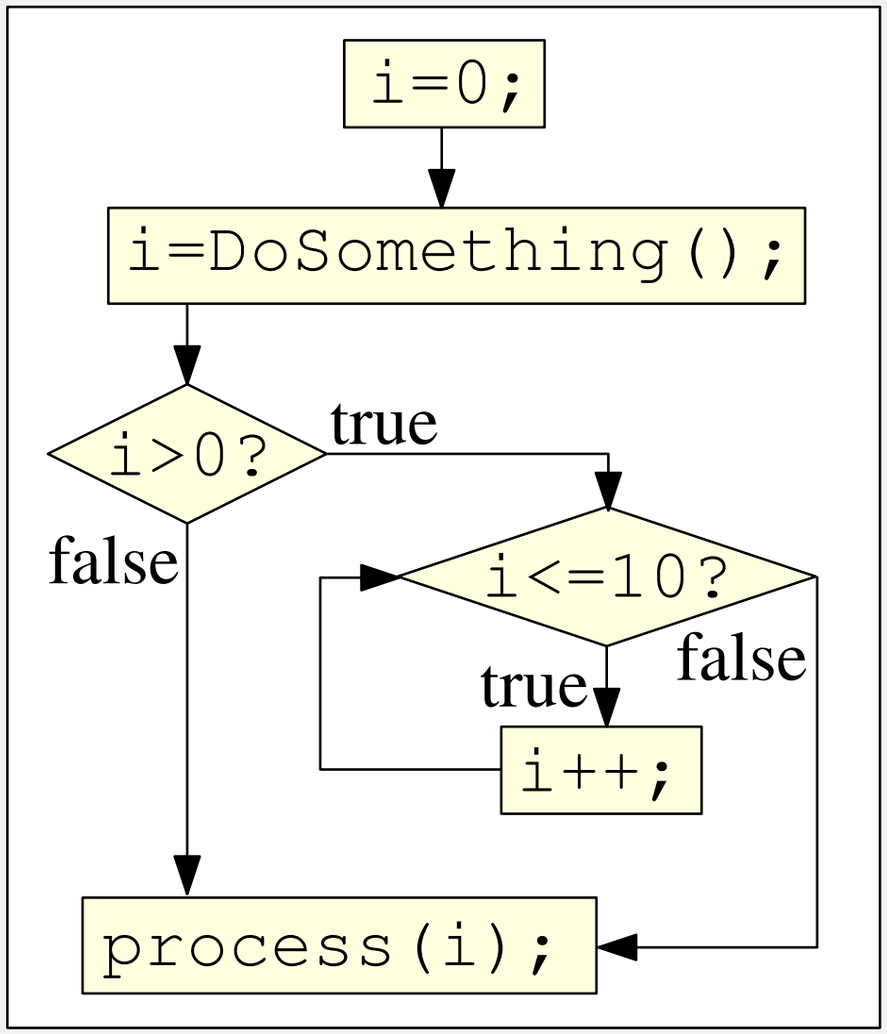


Systematic testing

Different criteria for coverage:

- **Method coverage:**
  Every method is tested
  (at least once)
- **Statement coverage:**
  Every statement is executed
  (at least once)
- **Branch coverage:**
  All branches in control flow
  are executed (at least once)
- **Path coverage:**
  All paths are executed
  (at least once)

```
int i = 0;
i = DoSomething();
if (i>0){
    while (i<=10)
          i++;
}
process(i);
```

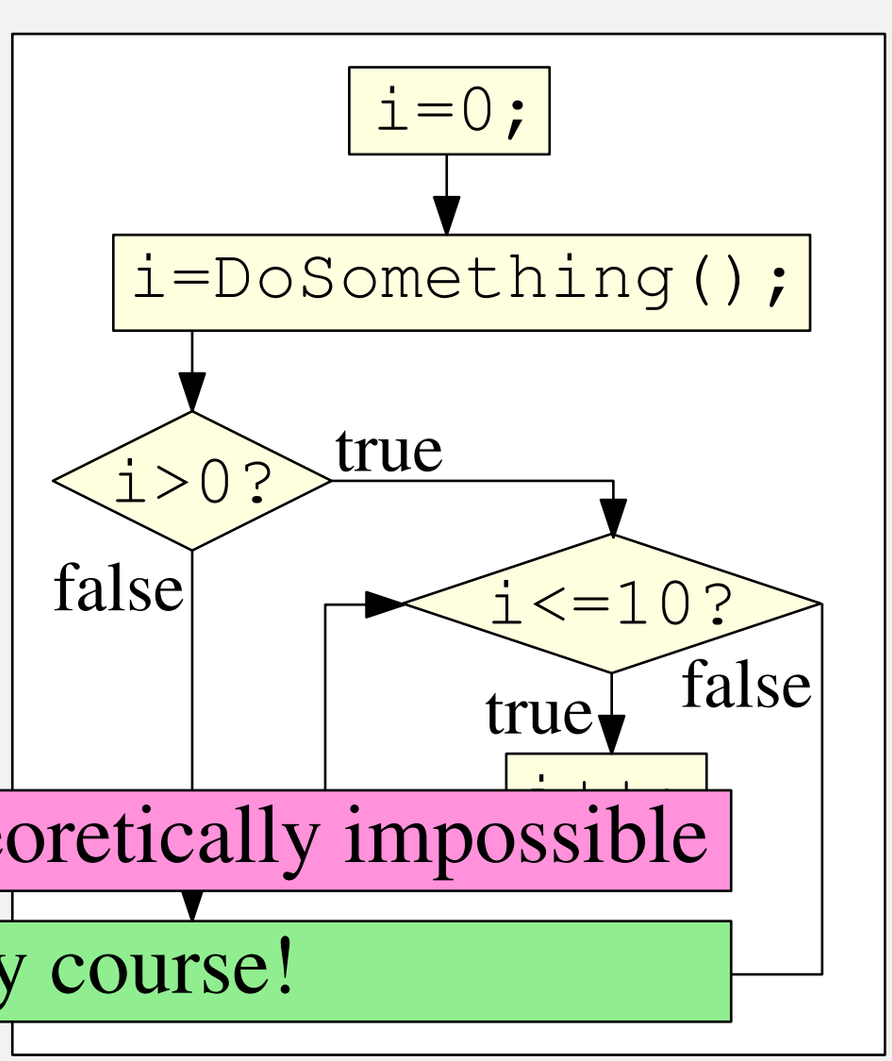


Systematic testing

Different criteria for coverage:

- **Method coverage:**
  Every method is tested
  (at least once)
- **Statement coverage:**
  Every statement is executed
  (at least once)
- **Branch coverage:**
  All branches in control flow
  are executed (at least once)
- **Path coverage:**
  All paths are executed
  (at least once)

```
int i = 0;
i = DoSomething();
if (i>0){
    while (i<=10)
          i++;
}
process(i);
```



Bad news: All of these are theoretically impossible

Good news: This isn't a theory course!

Branch coverage: simple control flow

If statements:

- `if ( `*Expr*` ) { `*Statement*`; }`
- Test with *Expr* being `true` and being `false`.

If-else statements:

- `if ( `*Expr*` ) { `*Statement1*`; }`
  `else { `*Statement2*`; }`
- Test with *Expr* being `true` and being `false`.

For loops:

- `for ( `*Expr1*` ; `*Expr2*` ; `*Expr3*` ) {`
  Statement`; }`
- `while ( `*Expr*` ) { `*Statement*`; }`
- Test with zero, one, and more interations.

Branch coverage (cont.)

Q: How should you cover the following constructs:

"?:" expresions:

- *Expr1* `?` *Expr2* `:` *Expr3*

Lazy evaluation:

- *Expr1* `&&` *Expr2*
- *Expr1* `||` *Expr2*

Switch statement:

```
• switch (Expr ) {
  case Expr1  :   Statement1; break;
  default     :   Statement2; }
```

Exceptions:

```
• try { ... } catch { ... } finally {
  ... }
```

Coverage table

- A white box provides a coverage table to ensure coverage

```
void m(int account, int rate) {
    account = account + 10;

    if (account > 90)    /* 1 */
        rate = rate * 2;
    else
        rate = rate / 2;

    out("account = " + account
        + " rate = " + rate);
}
```

Coverage table:

Branch	Input property	Covered by test
(1) `true`	`account > 90`	A
(1) `false`	`account <= 90`	B

Expectancy table:

Test ID	Input	Expected output	Actual output	Outcome
A	(98, 3)	”a=108, r=6”	”a=108, r=6”	pass
B	(76, 4)	”a=86, r=2”	”a=86, r=2”	pass

Example 1

- This program should compute the min and max of input:

```
public static void main (String[] args) {
    int mi, ma;
    if (args.length == 0)                          /* 1 */
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i = 1; i < args.length; i++) {    /* 2 */
            int obs = Integer.parseInt(args[i]);
            if (obs > ma) ma = obs;                /* 3 */
            else if (mi < obs) mi = obs;           /* 4 */
        }
        System.out.println("Minimum = " + mi + "; maximum = " + ma);
    }
}
```

- Label the branches
- Contruct expectancy- and coverage tables

Example 1 (tables)

Coverage table:

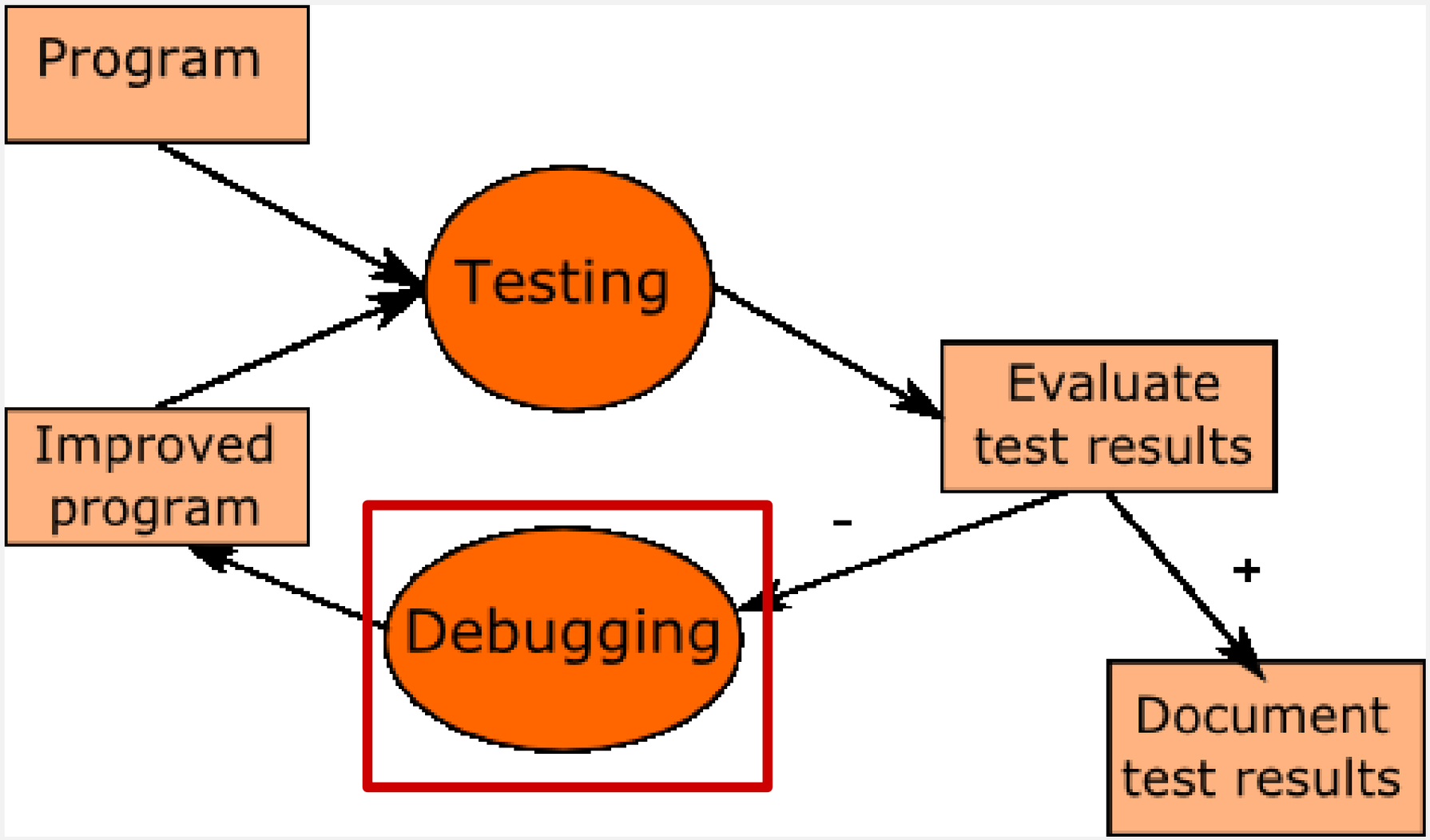
Choice	Input property	Input data set
1 true	No numbers	A
1 false	At least one number	B
2 zero times	Exactly one number	B
2 once	Exactly two numbers	C
2 more than once	At least three numbers	E
3 true	Number $>$ current maximum	C
3 false	Number $<$ current maximum	D
4 true	Number $\leq$ current maximum and $>$ current minimum	E, 3rd number
4 false	Number $\leq$ current maximum and $\leq$ current minimum	E, 2nd number

Expectancy table:

Input data set	Input contents	Expected output	Actual output
A	(no numbers)	No numbers	*No numbers*
B	17	17 17	*17 17*
C	27 29	27 29	*27 29*
D	39 37	37 39	*39 39*
E	49 47 48	47 49	*49 49*

Evaluate test results

- The test suite caught a bug!



- Next step: debugging
- The failed test case often provide hints to where the bug is

Debugging

```
public static void main (String[] args) {
    int mi, ma;
    if (args.length == 0)                          /* 1 */
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i = 1; i < args.length; i++) {    /* 2 */
            int obs = Integer.parseInt(args[i]);
            if (obs > ma) ma = obs;                /* 3 */
            else if (mi < obs) mi = obs;           /* 4 */
        }
        System.out.println("Minimum = " + mi + "; maximum = " + ma);
    }
}
```

- For both test cases, the min value is wrong:
- Check statement where `mi` is set
- Conditional is the wrong way round

Actual output
No numbers
17 17
27 29
39 39
49 49

Debugging

```java
public static void main (String[] args) {
    int mi, ma;
    if (args.length == 0)                          /* 1 */
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i = 1; i < args.length; i++) {    /* 2 */
            int obs = Integer.parseInt(args[i]);
            if (obs > ma) ma = obs;                /* 3 */
            else if (mi > obs) mi = obs;           /* 4 */
        }
        System.out.println("Minimum = " + mi + "; maximum = " + ma);
    }
}
```

- For both test cases, the min value is wrong:
- Check statement where `mi` is set
- Conditional is the wrong way round

Actual output
No numbers
17 17
27 29
39 39
49 49

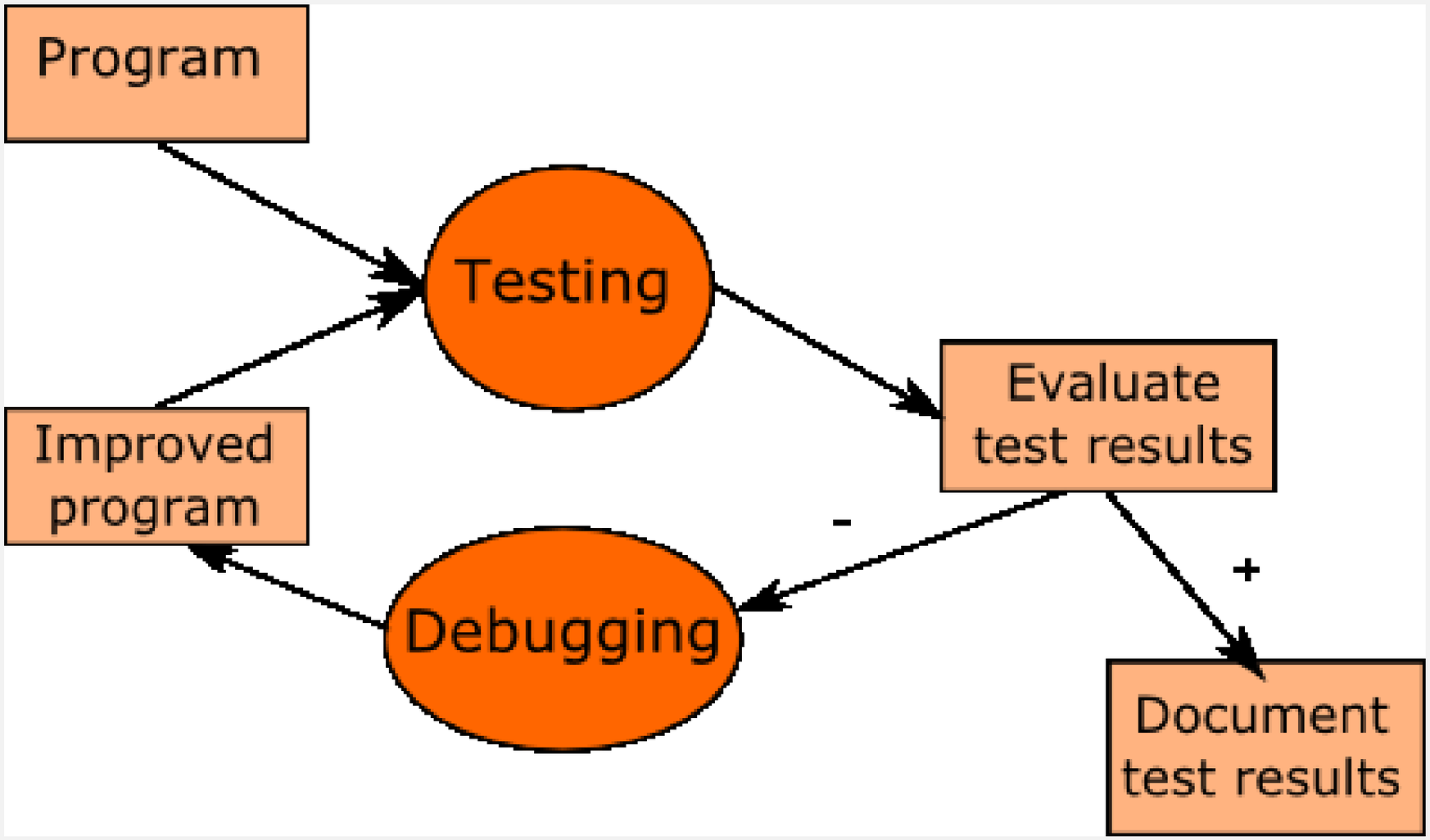
Re-check coverage table

- We changed the code, so the coverage table is outdated
- In this case, it is an easy change:

Choice	Input property	Input data set
1 true	No numbers	A
1 false	At least one number	B
2 zero times	Exactly one number	B
2 once	Exactly two numbers	C
2 more than once	At least three numbers	E
3 true	Number $>$ current maximum	C
3 false	Number $\leq$ current maximum	D
4a true	Number $\leq$ current maximum and	$<$ current minimum E, 2nd number
4a false	Number $\leq$ current maximum and	$\geq$ current minimum E, 3rd number

- The expectancy table stays the same (this time)

Rerun tests



- It might be tempting to rerun only the two test cases
- Regression testing: run entire test suite again, to check that you haven't introduced any new "features"

Example 2

- The following code should compute the two smallest input

```
public static void main (String[] args) {
    int mi1 = 0, mi2 = 0;
    if (args.length == 0) {                                  /* 1 */
        System.out.println("No numbers");
    } else {
        mi1 = Integer.parseInt(args[0]);
        if (args.length == 1) {                              /* 2 */
            System.out.println("Smallest = " + mi1);
        } else {
            int obs = Integer.parseInt(args[1]);
            if (obs < mi1) { mi2 = mi1; mi1 = obs; }         /* 3 */
            for (int i = 2; i < args.length; i++) {          /* 4 */
                obs = Integer.parseInt(args[i]);
                if (obs < mi1) { mi2 = mi1; mi1 = obs; }/* 5 */
                else if (obs < mi2) mi2 = obs;               /* 6 */
            }
            System.out.println("The two smallest are " + mi1 + " and " + mi2);
}   }   }
```

- Label branches
- Contruct expectancy- and coverage tables

```java
public static void main (String[] args) {
    int mi1 = 0, mi2 = 0;
    if (args.length == 0) {                              /* 1 */
        System.out.println("No numbers");
    } else {
        mi1 = Integer.parseInt(args[0]);
        if (args.length == 1) {                          /* 2 */
            System.out.println("Smallest = " + mi1);
        } else {
            int obs = Integer.parseInt(args[1]);
            if (obs < mi1) { mi2 = mi1; mi1 = obs; }     /* 3 */
            for (int i = 2; i < args.length; i++) {      /* 4 */
                obs = Integer.parseInt(args[i]);
                if (obs < mi1) { mi2 = mi1; mi1 = obs; }/* 5 */
                else if (obs < mi2) mi2 = obs;           /* 6 */
            }
            System.out.println("The two smallest are " + mi1 + " and " + mi2);
}   }   }
```

Choice	Input property	Input data set
1 true	No numbers	A
1 false	At least one number	B
2 true	Exactly one number	B
2 false	At least two numbers	C
3 false	Second number $\geq$ first number	C
3 true	Second number $<$ first number	D
4 zero time	Exactly two numbers	D
4 once	Exactly three numbers	E
4 more than once	At least four numbers	H
5 true	Third number $<$ current minimum	E
5 false	Third number $\geq$ current minimum	F
6 true	Third number $\geq$ current minimum and $<$ second least	F
6 false	Third number $\geq$ current minimum and $\geq$ second least	G

Example 2 (expectancy table)

- Expectancy table:

Input data set	Contents	Expected output	Actual output
A	(no numbers)	No numbers	*No numbers*
B	17	17	*17*
C	27 29	27 29	*27 0*
D	39 37	37 39	*37 39*
E	49 48 47	47 48	*47 48*
F	59 57 58	57 58	*57 58*
G	67 68 69	67 68	*67 0*
H	77 78 79 76	76 77	*76 77*

- Test case C and G indicated a bug
- For both, `mi2` has its initial value 0, and not one from input

Example 2 (expectancy table)

- Expectancy table:

```
public static void main (String[] args) {
    int mi1 = 0, mi2 = 0;
    if (args.length == 0) {                              /* 1 */
        System.out.println("No numbers");
    } else {
        mi1 = Integer.parseInt(args[0]);
        if (args.length == 1) {                          /* 2 */
            System.out.println("Smallest = " + mi1);
        } else {
            int obs = Integer.parseInt(args[1]);
            if (obs < mi1) { mi2 = mi1; mi1 = obs; }     /* 3 */
            for (int i = 2; i < args.length; i++) {      /* 4 */
                obs = Integer.parseInt(args[i]);
                if (obs < mi1) { mi2 = mi1; mi1 = obs; }/* 5 */
                else if (obs < mi2) mi2 = obs;           /* 6 */
            }
            System.out.println("The two smallest are " + mi1 + " and " + mi2);
}   }   }
```

- Test case C and G indicated a bug
- For both, `mi2` has its initial value 0, and not one from input
- `mi2 = obs` is never executed for increasing input

Example 2 (expectancy table)

- Expectancy table:

```
public static void main (String[] args) {
    int mi1 = 0, mi2 = 0;
    if (args.length == 0) {                                  /* 1 */
        System.out.println("No numbers");
    } else {
        mi1 = Integer.parseInt(args[0]);
        if (args.length == 1) {                              /* 2 */
            System.out.println("Smallest = " + mi1);
        } else {
            int obs = Integer.parseInt(args[1]);
            mi2 = obs;
            if (obs < mi1) { mi2 = mi1; mi1 = obs; }         /* 3 */
            for (int i = 2; i < args.length; i++) {          /* 4 */
                obs = Integer.parseInt(args[i]);
                if (obs < mi1) { mi2 = mi1; mi1 = obs; }/* 5 */
                else if (obs < mi2) mi2 = obs;               /* 6 */
            }
            System.out.println("The two smallest are " + mi1 + " and " + mi2);
}   }   }
```

- Test case C and G indicated a bug
- For both, `mi2` has its initial value 0, and not one from input
- `mi2 = obs` is never executed for increasing input
- Fixed with initialization

Summary

- We can't test every input
- Tests cannot prove the absence of bugs, but proper coverage can heighten our confidence in the correctness of the program
- White box testing uses source code to generate test input
- A test case takes a path through the program
- Coverage criteria:
  - Method coverage:
    each method is reached by some path
  - Statement coverage:
    each node in the control graph is used by some path
  - Branch coverage:
    each edge in the control graph is used by some path
  - Path coverage:
    all paths are included in the suite

Next time

- Exercises today from 12-14 in 3A 12/14 (problem set on learnIT)
- No class tomorrow, tuesday
- Wednesday: hands-on lecture
- Install git on your laptops (git-scm.com) if you want to play along