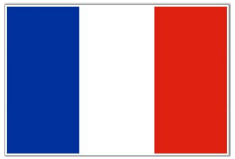
Testing, debugging, (specification) and verification

Team

Teacher: Atze van der Ploeg (atze)

Course assistants:

- Mauricio Chimento (chimento)
- Simon Robillard (simrob)
- Bart van Delft (vandeba)



... @chalmers.se

Problems? Talk to us!!

Student representatives

....

Who gets the store credit?!!

Student representatives

Name	email
Tobias Edvardsson	tobedv
Jesper Kjellqvist	jeskje
Oscar Muhr	oscarmu
Henrik Olsson	henolss
Robin Punell	punell

...@student.chalmers.se

What’s all this then?

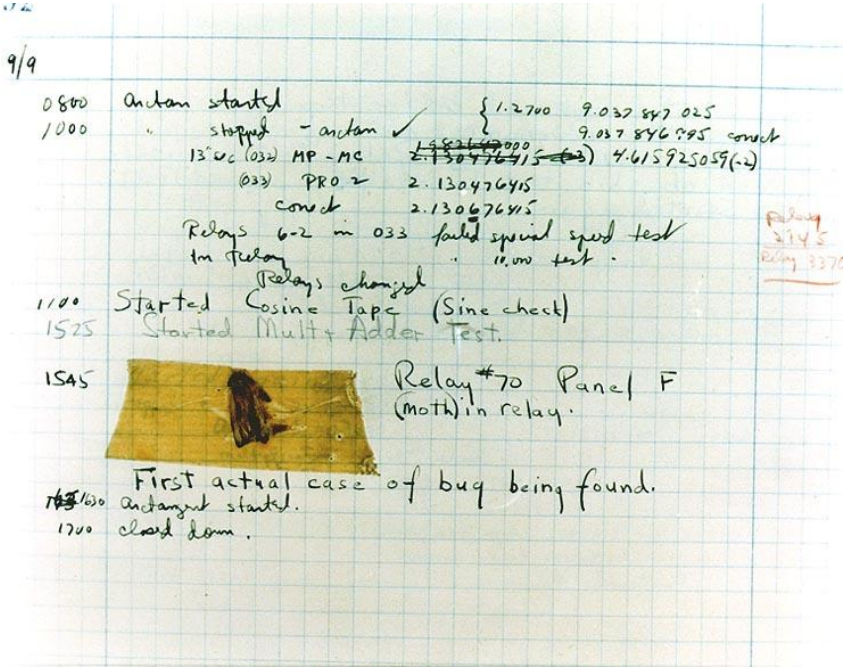
This course is about:

What’s a bug?

A **software bug** is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. -- Wikipedia

Bug etymology

- “bug” used to describe defects in guns, pinball machines, cars, since at least 1878
- popularized for computers by moth in relay tube of harvard mark II in 1946



Terminology

- Testing - Check for bugs
- Debugging - Remove bugs
- Specification - Describe expected behavior (what is a bug?)
- (Formal) Verification - Prove that program conforms to specification (prove that there are no bugs)

This course

Introduction to techniques to get (some) certainty that your program does what it’s supposed to.

Organizational stuff

www.cse.chalmers.se/edu/course/TDA567

Passing criteria:

- Written exam
- 3 lab hand-ins
- Can be passed separately

Labs: Work in pairs, submission via Fire (see website)

Submit **by deadline**!

Course literature: All available as e-book, see website

Course Evaluation

5 Student representatives

- feedback meetings with teachers
- First meeting: Next week 13:00

Everyone: web questionnaire after course

Some really bad bugs

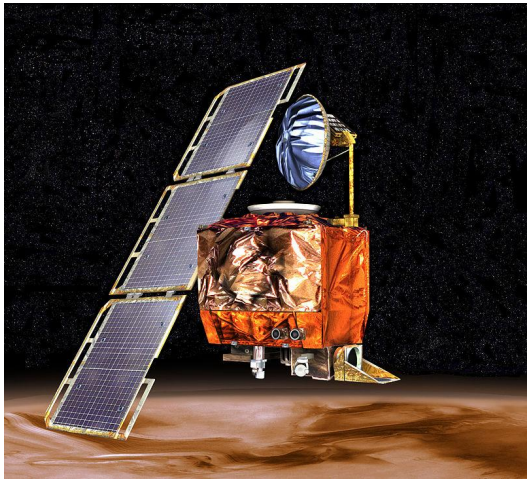
Ariane 5 rocket

- Exploded right after launch
- Conversion of 64-bit float to 16-bit integer caused overflow in guidance system made it crash

Another really bad space bug

Mars Climate Orbiter (1998)

- Came to close to planet
- Disintegrated in Mars atmosphere
- Pounds/second ≠ Newton/second



Scary bug

Therac-25 Radiotherapy Machine (1985-87)

- Patients overdosed
- 3 dead, 2 severly injured
- Cause: race condition

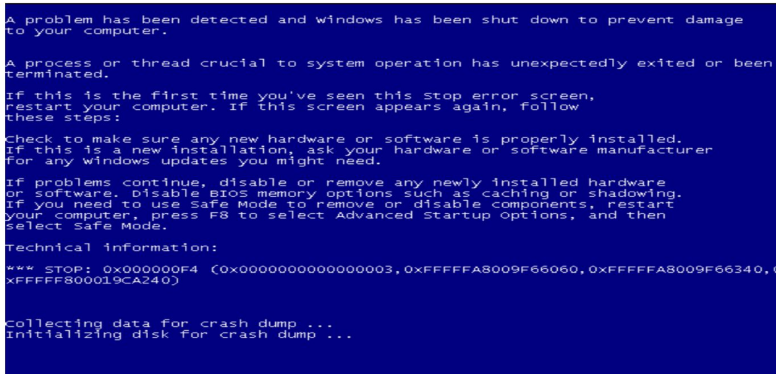
Costly Bug

Pentium Floating-point bug

- Incorrect result through floating point division
- Rarely encountered in practice
- Public outcry, bad handling, image dent

Software bugs abound

- Global cost of bugs $ 312 billion (estimate)
- Estimated 50% of programmers time spend on fixing bugs
- The earlier bug found (or prevented) the better



Brainstorm

How can you get some assurance that a program does what you want it to do?

Techniques for assurance

- Code review, Pair programming
- KISS
- Testing
- Types
- Formal proof of correctness (verification)
- “Proven technology”

Usually: more assurance = more effort
Lots of research focusses on more assurance for less effort

Level of certainty (*)

Field	Testing used?	Verification Used?
Web programming	Yes, typically manual	No
Game programming	Yes	No
OS	Yes	In research, Partial
Hardware	Probably	Yes
Aviation, Cars	Yes, extensive	Sometimes
Medical	Yes, extensive	Sometimes

(*) This is just an indication, I have no figures to back this up

What is a bug?

- Non termination
- Crash
  - type error
  - index-out-of-bounds
  - stackoverflow
  - segfault
  - ....

But that are not all bugs...

Specification

An unambiguous description of what a function should do.

Bug = failure to meet specification

Each (non-crashing) program is correct with respect to SOME specification

Precise

Economist: All cows in Scotland are brown!

Logician: There is at least one brown cow in Scotland

Computer scientist: There is at least one brown cow in Scotland, at least one side of which is brown.

Let's try

```
public static int[] sort(int[] a)
```

Requires: …

Ensures : …

Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers

Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers

Tests:

sort({3,2,5}) == {2,3,5}
sort({}) == {}
sort({17}) = {17}

Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers

Tests:

sort({3,2,5}) == {2,3,5} ✔
sort({}) == {} ✔
sort({17}) = {17} ✔

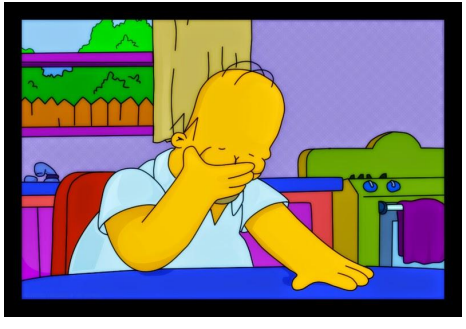
Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers

sort({}) == {1,2,3}



Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing only elements from a

sort({}) == {1,2,3}

Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing only elements from a

sort({1,2}) == {1,1,1,1,1,2,2,2,2}

Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing only elements from a

sort({1,2}) == {1,1,1,1,1,2,2,2,2}

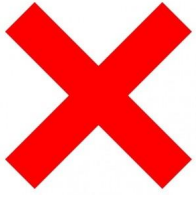
Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing a permutation of the elements in a

sort({1,2}) == {1,1,1,1,1,2,2,2,2}



Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing a permutation of the elements in a

sort({1,2}) == {1,1,1,1,1,2,2,2,2}

Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing a permutation of the elements in a

sort(null) throws nullpointer exception

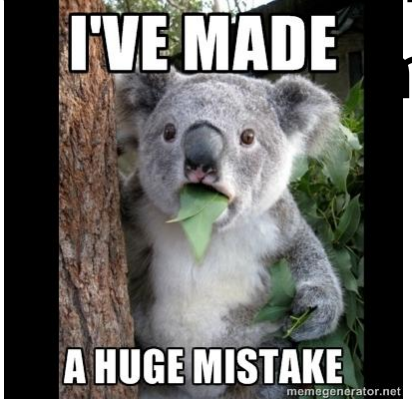
Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is an array of integers

Ensures : Output is a sorted array of integers containing a permutation nents in a

sort(null) throws nullpointer exception



Let’s try

```
public static int[] sort(int[] a)
```

Requires: Input is a non-null array of integers

Ensures : Output is a sorted array of integers containing a permutation of the elements in a

sort(null) throws nullpointer exception

Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is a non-null array of integers

Ensures : Output is a sorted array of integers containing a permutation of the elements in a

```
int[] a = new int[] { 3,2,1};
int[] b = sort (a);
a[0] == ???
```

Let's try

```
public static int[] sort(int[] a)
```

Requires: Input is a non-null array of integers

Ensures : No changes are made to the input array and the output is a new sorted array of integers containing a permutation of the elements in a

```
int[] a = new int[] { 3,2,1};
int[] b = sort (a);
a[0] == 3
```

Contract metaphor

Supplier: (callee)
Implementer of method

Client: (caller) Implementer of calling method or user

Contract:

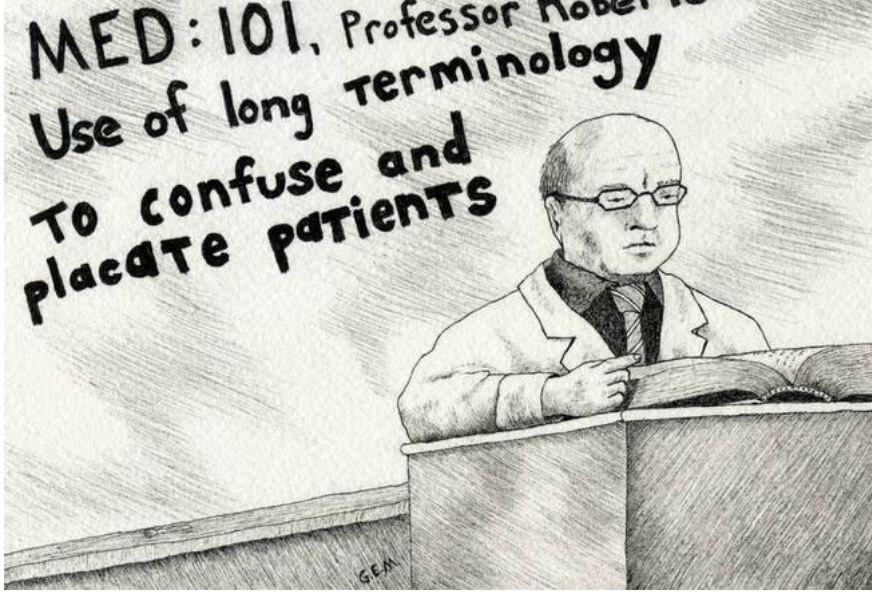
Requires*: What the **client** must ensure*

Ensures*: What the **supplier** must ensure*

Use fancy words!

Requires = Precondition

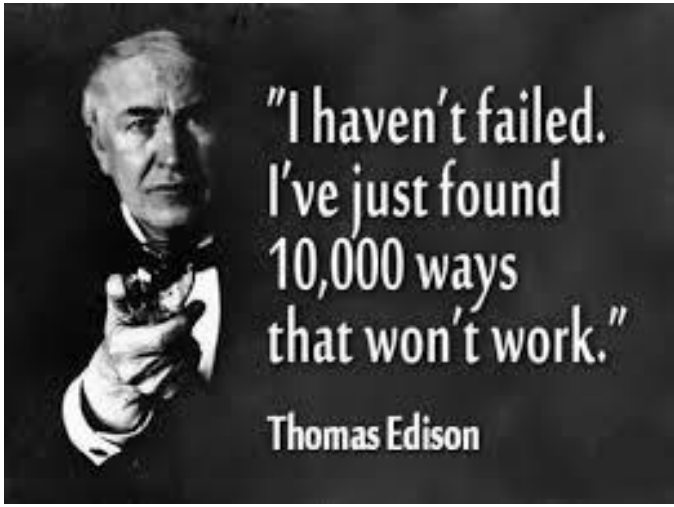
Ensures = Postcondition



If a caller of `C.m()` fullfills the required *Precondition* then the callee, `C.m()`, ensures that the *Postcondition* holds after `C.m()` finishes.

More terminology

Failure: Method `C.m()` fails if
precondition held before `C.m()`,
but postcondition does not hold after `C.m()`
(or if C.m() does not finish)

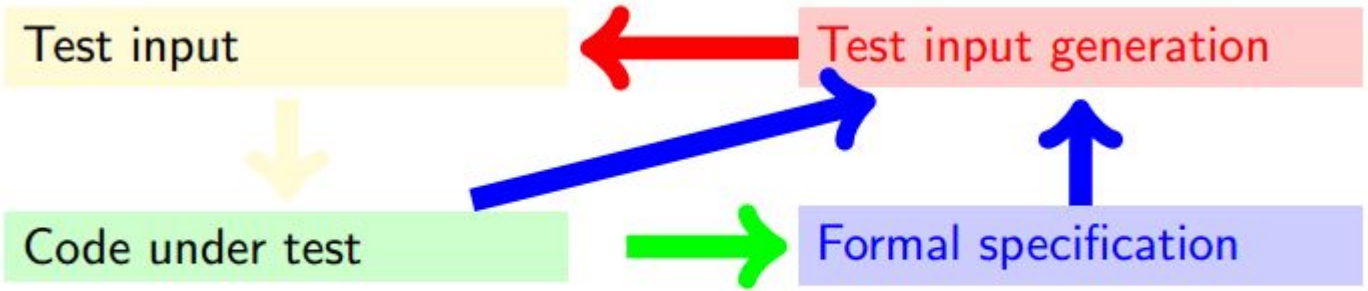


Correct: Method `C.m()` cannot fail.

In other words, whenever
`C.m()` is called and the precondition holds, then `C.m()` finishes and the postcondition holds.

Testing

Test: try out inputs, see if outputs are correct



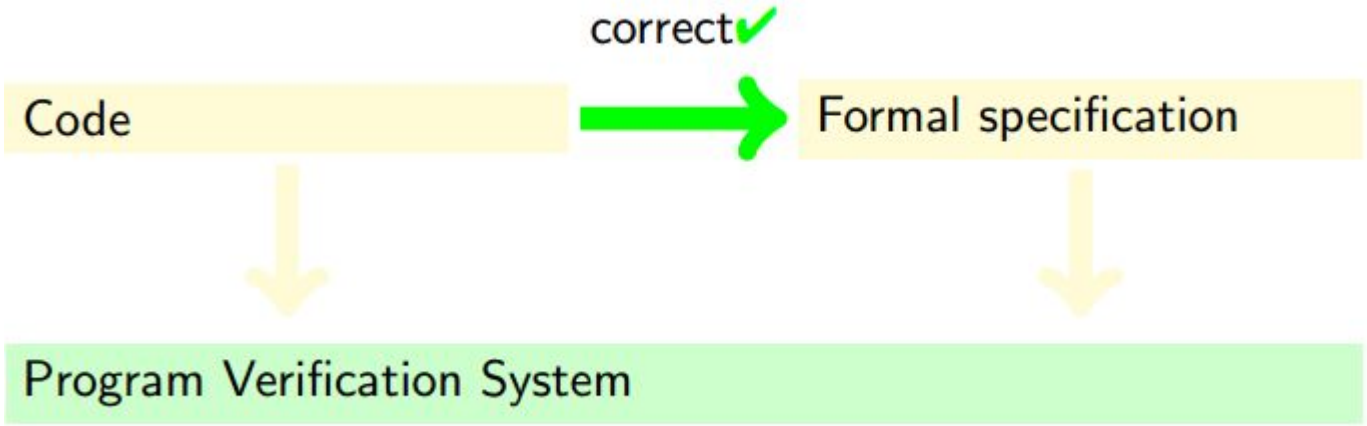
This course: Unit testing, property based testing

Debugging

Understand why a program does not do what it’s supposed to, usually via tool support such as the Eclipse debugger

Verification

Verification: Mathematically prove method correct



This course: First order logic and Dafny

Course contents

Testing: black vs white box, unit test, coverage, property based testing (stateless and stateful)

Debugging: execution control, inspection, localisation

Formal specification: contracts, assertions, invariants, Dafny, first-order-logic

Formal verification: Weakest precondition calculus, formal proofs, loop invariants