TDDD04: Software Testing Course outline and Introduction

Lena Buffoni lena.buffoni@liu.se

LINKÖPING UNIVERSITY

Teaching Team

Course leader :

Lena Buffoni

Course assistants (3 lab groups):

Leif Eriksson

John Tinnerholm

Anders Märak Leffler

(Lena – for the 726A88 ? TBD)

Course contents

- *Introduction to testing and the testing process*
- Black-box/white box testing
- Unit testing
- Integration testing
- System testing, model-based testing
- Model checking, symbolic execution
- Research in testing
- Agile testing
- Mutation testing

Course history

- Taken over in 2017 from Ola Leifler
- Guest participations from Liu, Saab, Ericsson
- Labs revised to focus more on practical skills and recent technologies

2020 – we are ~~going~~ still remote

- No written examination – continuous grade setting
- 6 lab assignements –graded 3,4,5
- Lectures and labs via Zoom
  - Use chat or raise hand for questions
  - Break-out rooms for discussions
- Teams group
- Gitlab repositories for code
- Lisam for submissions

Labs

- Done in pairs, and in WebReg groups of 20 students: *https://www.ida.liu.se/webreg3/TDDD04-2021-1/Labs*
- Some labs require collaboration (3,4), use Gitlab and Teams
- Hand in labs on time – late labs get grade 3 max
- All labs need to get at least 3 to pass the course
- Last lab session will be used for demonstrations
- Use your teams as support but work and grades are in pairs

Recommended Literature

A Practitioner's Guide to Software Test Design

Lee Copeland

Main course literature

Focus on software test design

Available online as an electronic book via the university library

- Complementary:

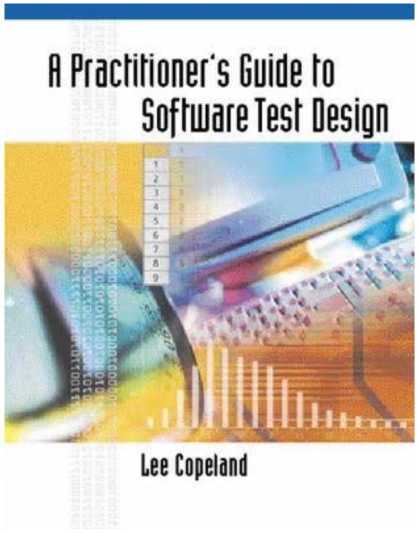
Software Testing, A Craftsman's Approach

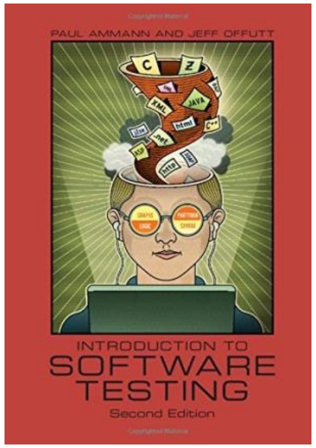
Paul C. Jorgensen (available online)

The Art of Software testing by Glenford J. Myers

Introduction to Software Testing by Paul Amman and Jeff Offutt (available online)

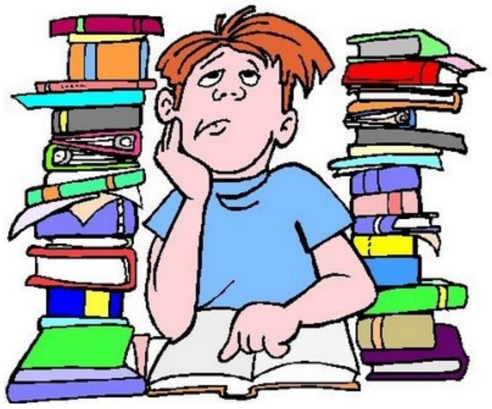
- Additional research papers (see course web)





How to achieve the best results?

- Participate in the lectures
- Read the recommended literature
- Read the instructions carefully
- Come prepared to the labs
- Hand in assignments on time
- Don't hesitate to ask for help



Introduction

Why do we test software?

LINKÖPING UNIVERSITY

What is the most important skill of a software tester?

communication

Discussion time:

What is software testing?

What is software testing?

IEEE defines software testing as

- A process of analyzing a software item to detect the *differences between existing and required conditions* (that is defects/errors/bugs) and to evaluate the features of the software item.

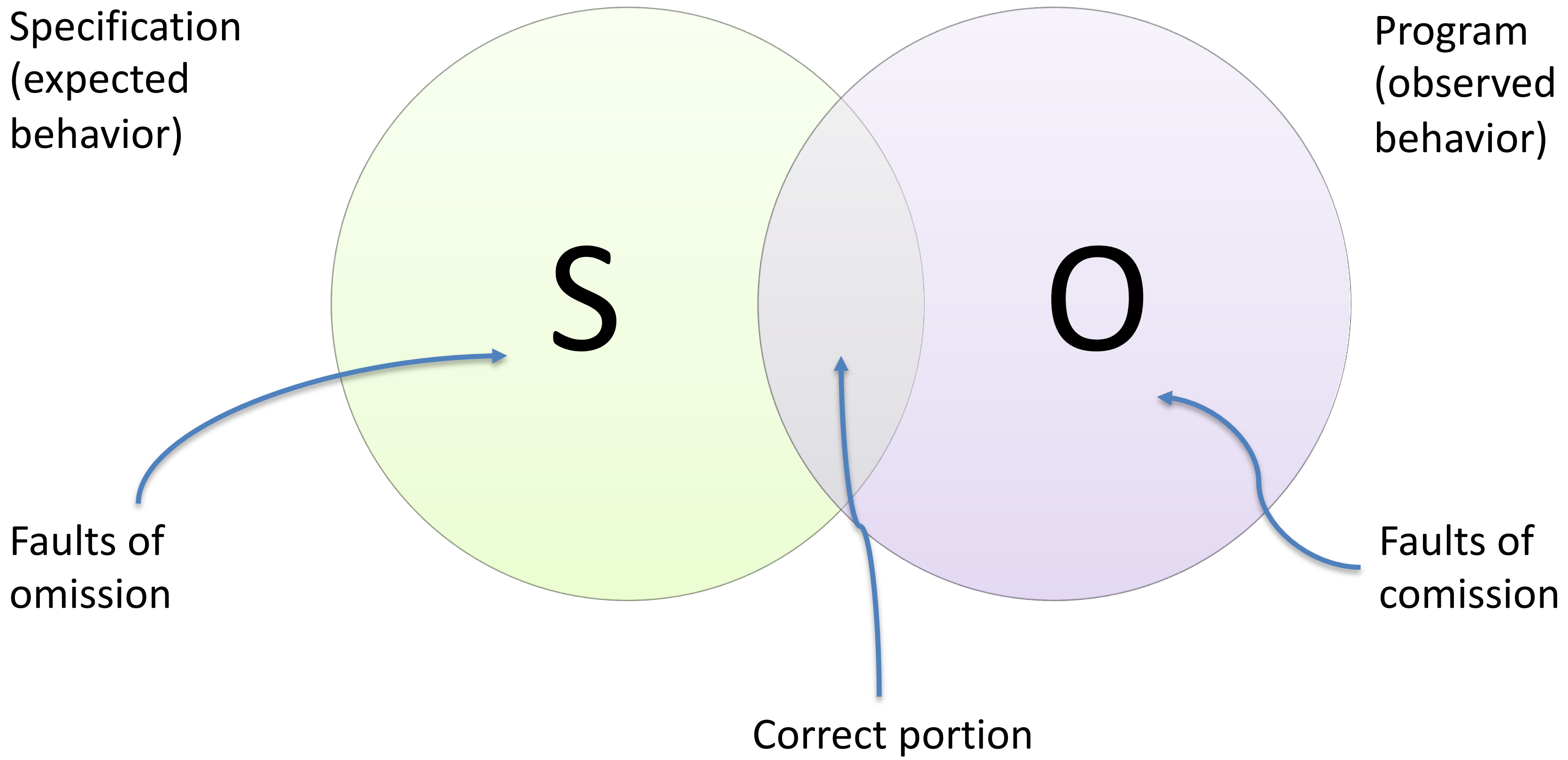
Note that…

…one needs to know the **required conditions**

…one needs to be able to observe the **existing conditions**

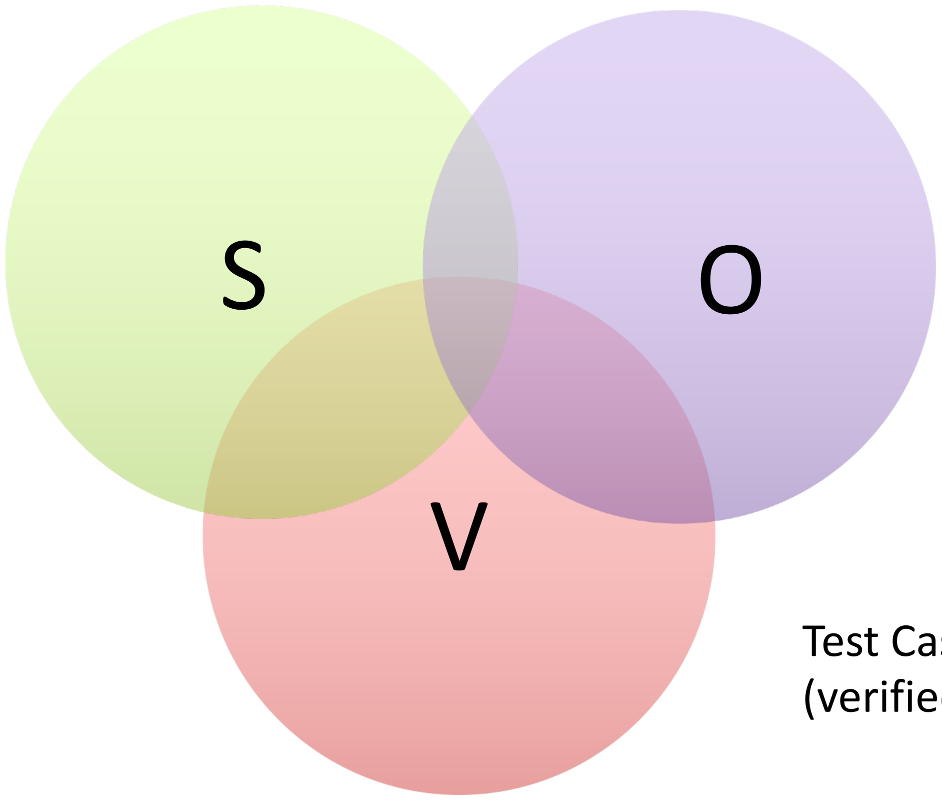
Testing focuses on behavioral (what the program does) and structural (how the program is) aspects

Program behavior


Specification (expected behavior)
Program (observed behavior)
S
O
Faults of omission
Faults of comission
Correct portion

Program behavior and testing

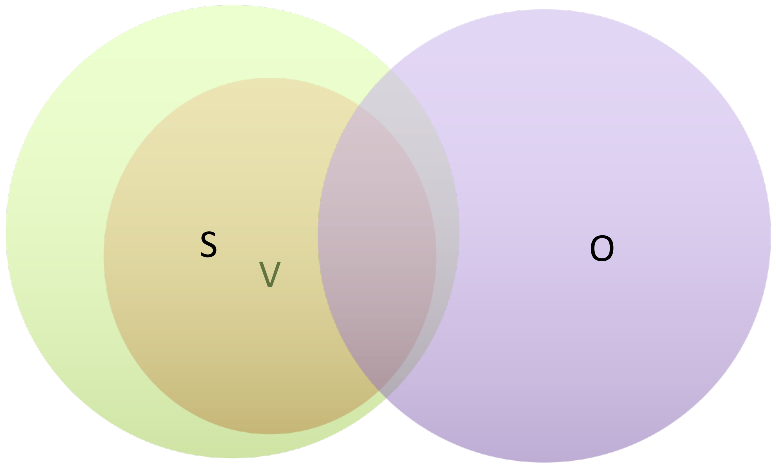
Specification (expected behavior)



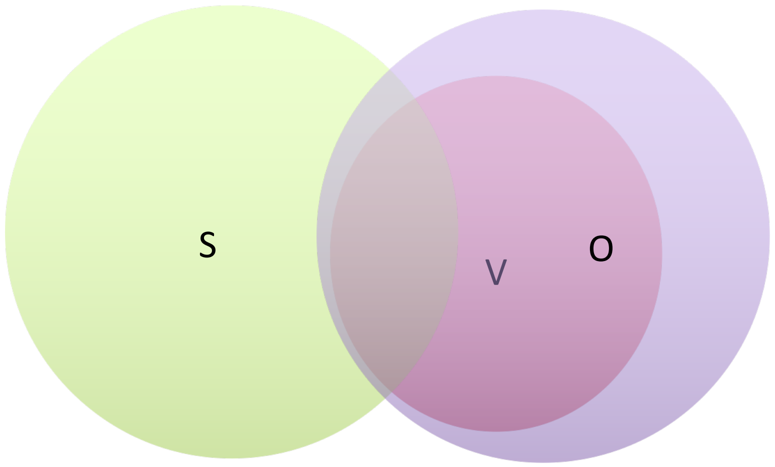
Program (observed behavior)

Test Cased (verified)

Functional vs structural testing



Functional test cases



Structural test cases

Discussion time:

Why do we test software?
What do we want to accomplish?

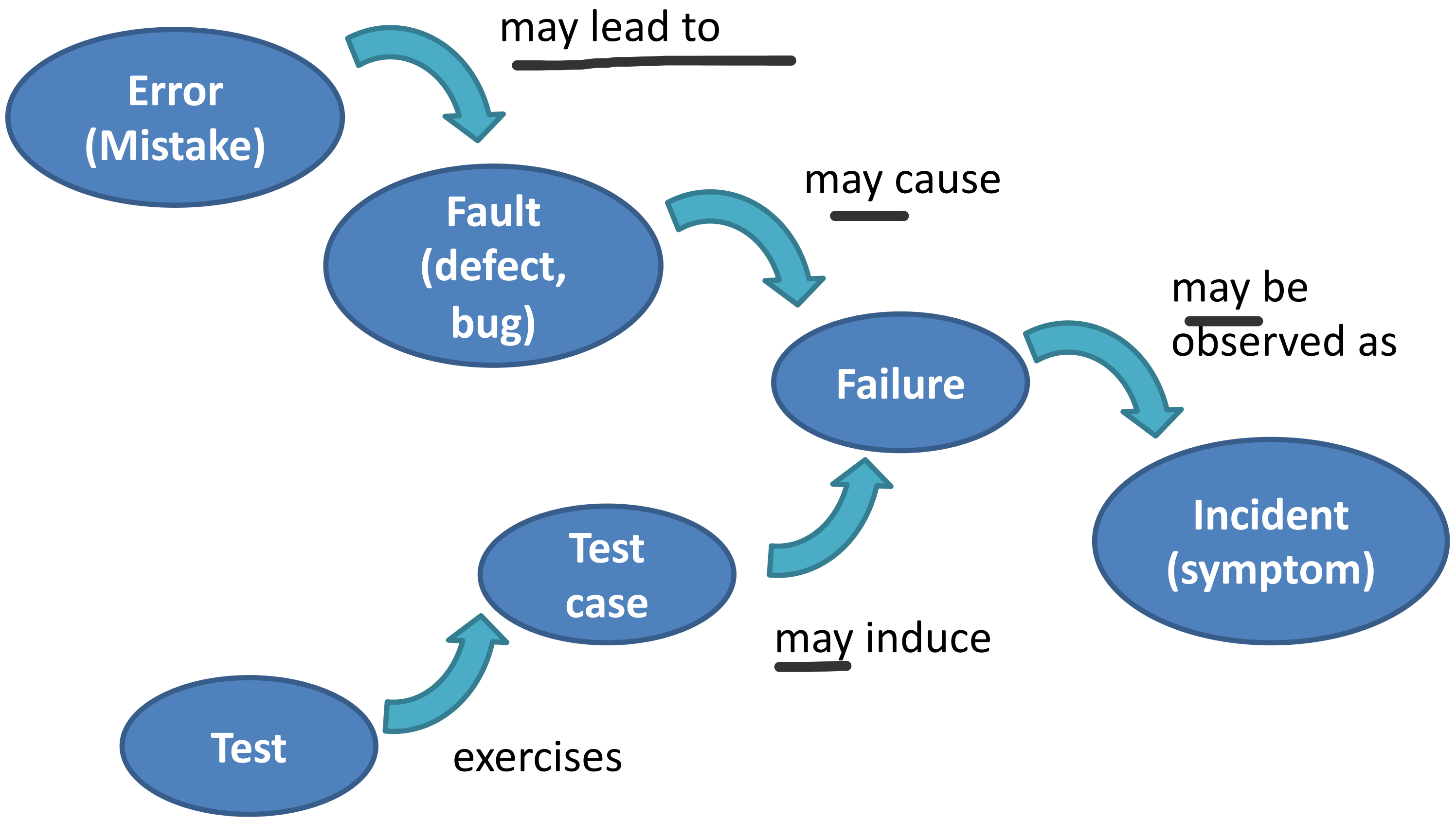
What is the goal when testing?

Some common answers:

- Do we want to isolate and fix bugs in the program?
- Do we want to demonstrate that the program works?
- Do we want to demonstrate that the program **doesn't** work?
- Do we want to reduce the risk involved in using the program?
- Do we want to have a methodology for producing better quality software?

These goals correspond to 5 levels of "test process maturity" as defined by Beizer

Testers language



Definitions (IEEE)

- **Error:** people make errors. A good synonym is mistake. When people make mistakes while coding, we call these mistakes bugs. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.
- **Fault:** a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. We might speak of faults of commission and faults of omission. **A fault of commission** occurs when we enter something into a representation that is incorrect. **Faults of omission** occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Fault classification

Software defect taxonomies: what kind is it?

- Useful to guide test planning (e.g. have we covered all kinds of faults)
- Beizer (1984): Four-level classification
- Kaner et al. (1999): 400 different classifications

Severity classification: how bad is it?

- Important to **define** what each level means
- **Severity does not equal priority**
- Beizer (1984): mild, moderate, annoying, disturbing, serious, very serious, extreme, intolerable, catastrophic, infectious.
- ITIL (one possibility): severity 1, severity 2

Definitions (IEEE)

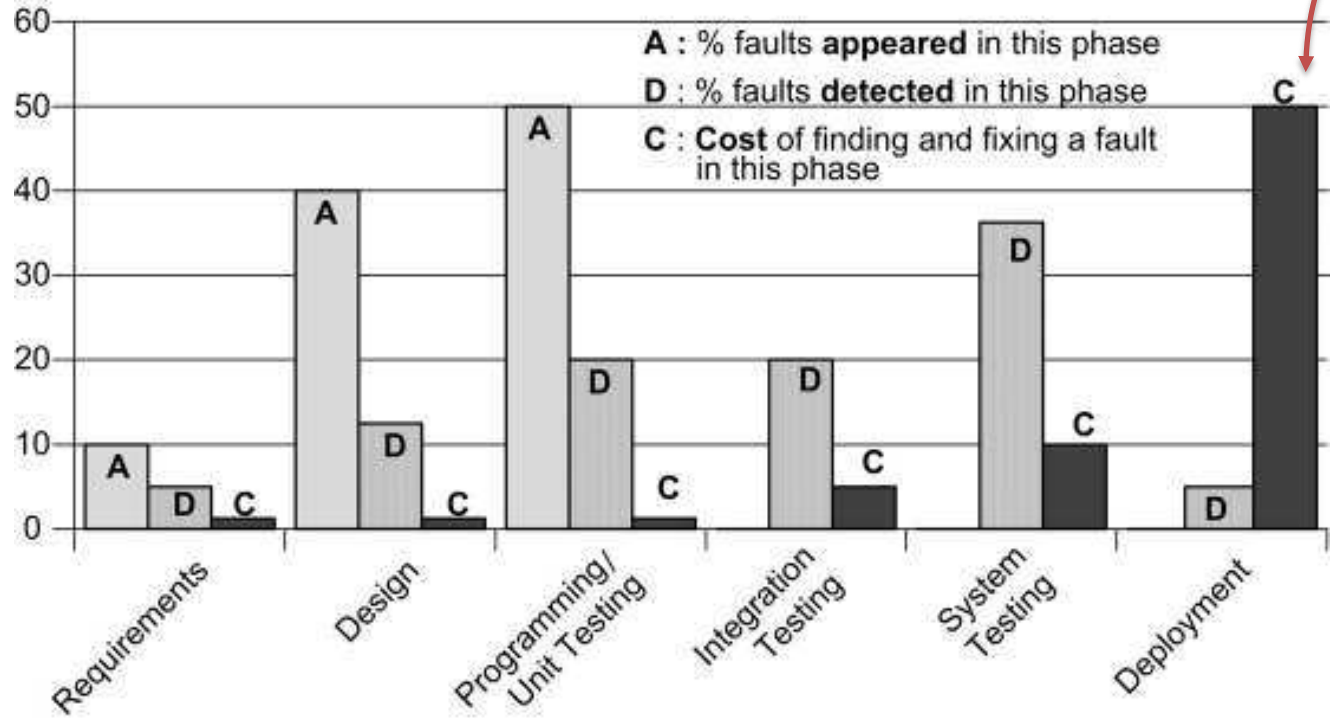
- **Failure:** a failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission?
- **Incident:** when a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

Definitions (IEEE)

- **Test:** testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.
- **Test Case:** test case has an identity and is associated with a program behavior. A test case also has a set of inputs and a list of expected outputs.

Cost of testing late

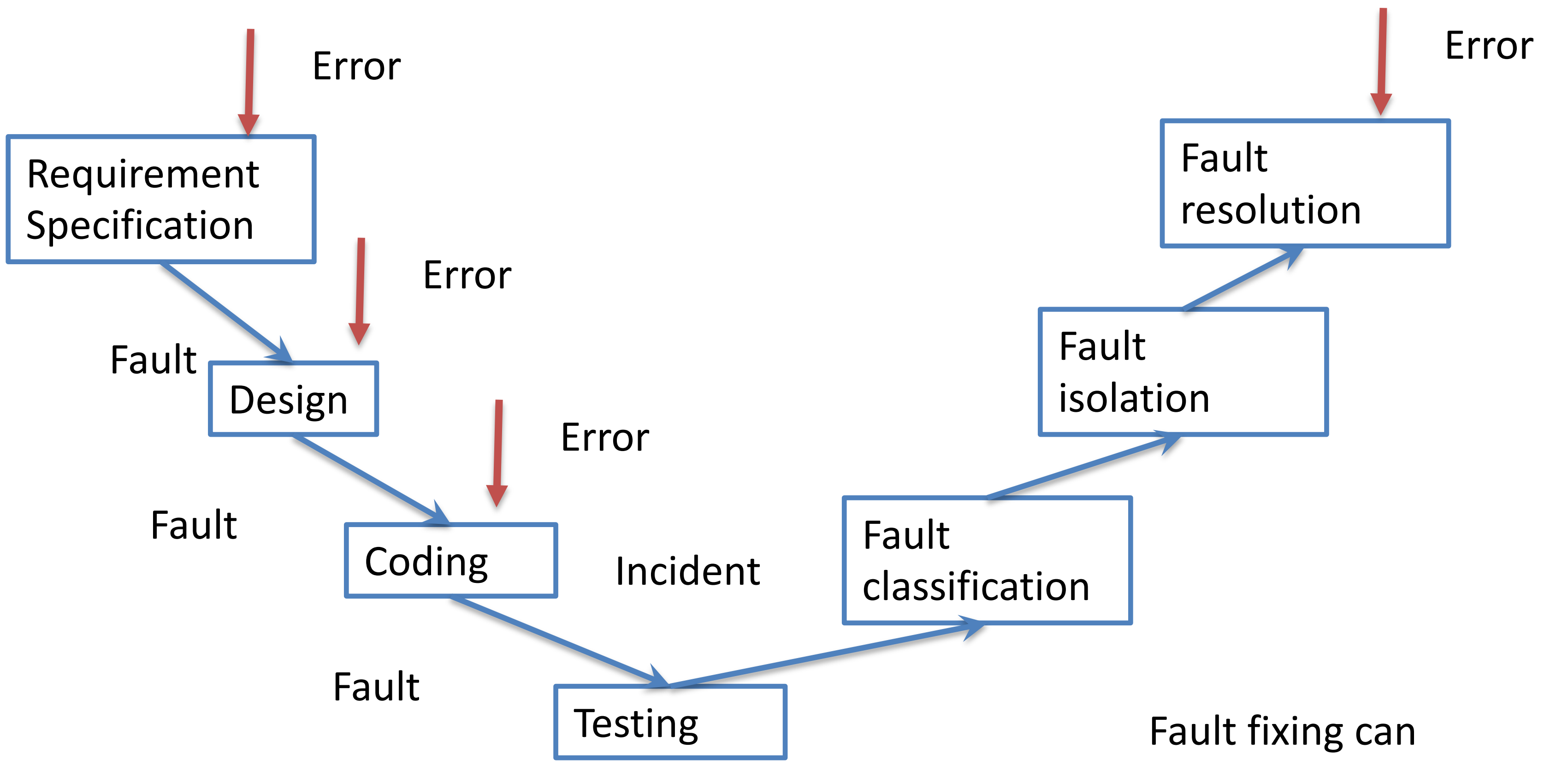
50 times more expensive to fix a fault at this stage!



Ariane 5 – a spectacular failure

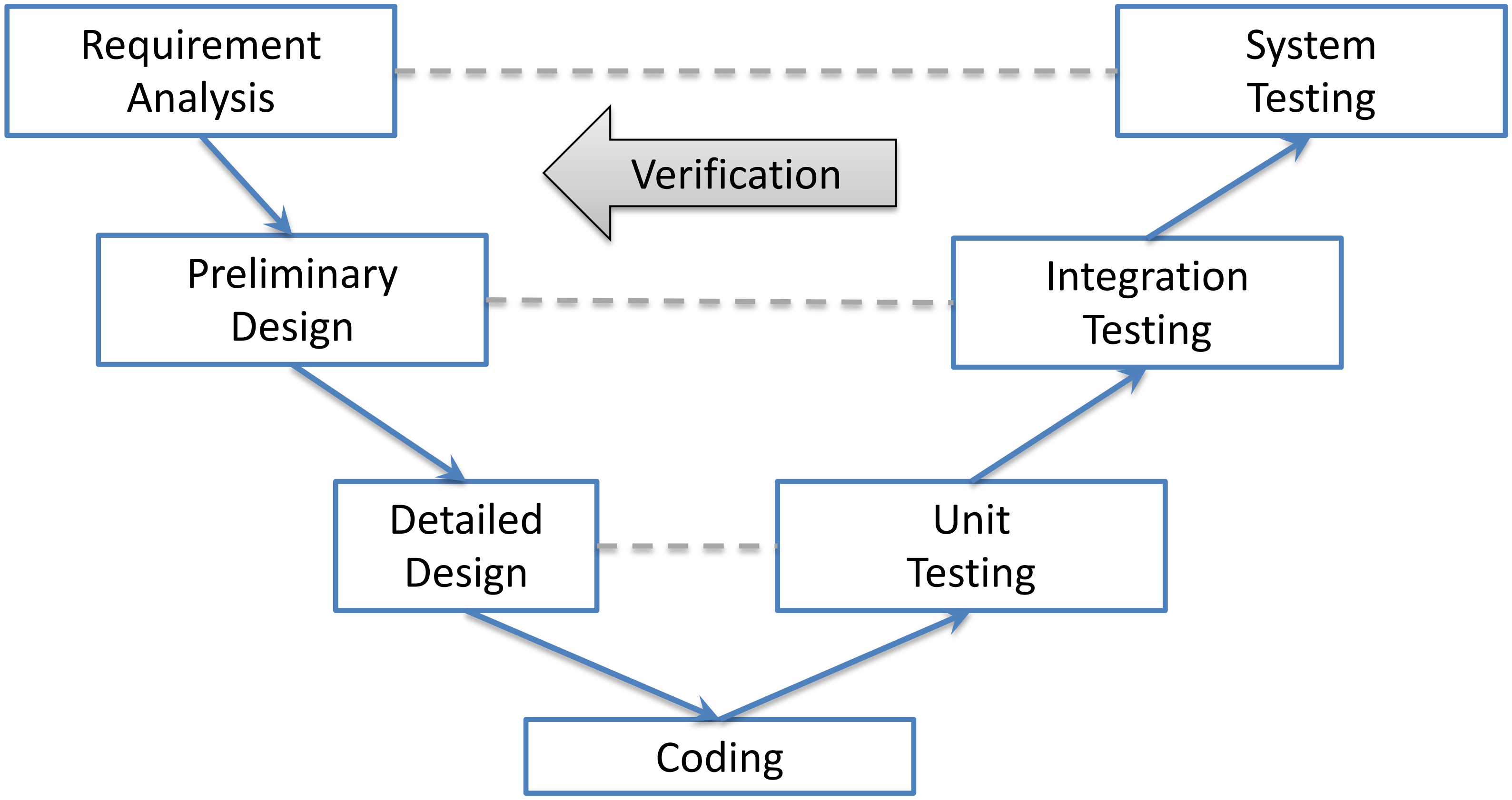
- 10 years and $7 billion to produce
- < 1 min to explode
- the error came from a piece of the software that was *not* needed during the crash
- programmers thought that this particular value would never become large enough to cause trouble
- removed the test present in Ariane 4 software
- 1 bug = 1 crash

Software testing life-cycle



Fault fixing can introduce more errors!

Testing in the waterfall model



How would you test a ballpoint pen?

- Does the pen write?
- Does it work upside down?
- Does it write in the correct color?
- Do the lines have the correct thickness?
- Does the click-mechanism work? Does it work after 100,000 clicks?
- Is it safe to chew on the pen?
- Is the logo on the pen according to company standards?
- Does the pen write in -40 degree temperature?
- Does the pen write underwater?
- Does the pen write after being run over by a car?
- **Which are relevant? Which are not relevant? Why (not)?**

To ponder…

- **Discuss:** Who should write tests? Developers? The person who wrote the code? An independent tester? The customer? The user? Someone else?
- **Discuss:** When should tests be written? Before the code? After the code? Why?
- **We will return to these issues!**

Discussion time:

What should a test case contain?

Test case structure

- Identifier – persistent unique identifier of the test case
- Setup/environment/preconditions
- How to perform the test – including input data
- Expected output data – including how to evaluate it
- Purpose – what this test case is supposed to show
- Link to requirements/user story/use case/design/model
- Related test cases
- Author, date, revision history …

Limitations of testing

An example proposed by Robert Binder to show limitations of testing

```
int scale(int j) {
    j = j - 1; // should be j = j + 1
    j = j / 30000;
    return j;}
```

Input (j)	Expected Result	Actual Result
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

Limitations of testing

An example proposed by Robert Binder to show limitations of testing

```
int scale(int j) {
    j = j - 1; // should be j = j + 1
    j = j / 30000;
    return j;}
```

For a 16 bit encoding of integers, out of 65,530 testable values only 6 will detect the bug:

-30001, -30000, -1, 0, 29999, and 30000.

Limitations of testing

- **Testing cannot prove correctness**
- Testing **can** demonstrate the presence of failures
- Testing **cannot** prove the absence of failures
- **Discuss:** What does it mean when testing does not detect any failures? **Discuss:** Is correctness important? Why? Why not? What is most important? **Discuss:** Would it be possible to prove correctness? Any limitations?
- **Testing doesn't make software better**
- Testing must be combined with fault resolution to improve software **Discuss:** Why test at all then?

Next lecture

- Read on Black-box testing techniques
- Check the exercise on black-box testing on the lectures page

Thank you!

Questions?

LINKÖPING UNIVERSITY