Software Engineering and Scientific Computing

Barbara Paech, Hanna Remmel
Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

software engineering heidelberg

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Schedule Third Day

9:00	Quality Assurance and Testing
10:30	Break
11:00	Modeling Knowledge Management
12:00	Lunch
13:00 Incl. a short break	Tools, Exercises Branches and Tagging in Subversion IDE Wrap-Up, Feedback
16.00	End

Programming in a small team

What is
Ron doing?

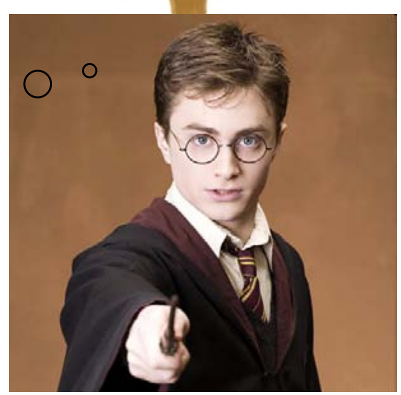
Project management
Issue Tracking

I want to explain
my ideas to Hermione

Modeling
Knowledge Management

I want to change
Ginnys code

Version management,
Build management



I want to check
Harrys changes

Quality assurance
Testing

[http://www.elec-intro.com]

Quality Assurance

What is Quality?

- Basic definition of quality: meeting the users' needs
  - needs, not wants
  - true functional needs are often unknowable

- There is a hierarchy of needs.
  - Do the required tasks.
  - Meet performance requirements.
  - Be usable and convenient.
  - Be economical and timely.
  - Be dependable and reliable.

[http://www.sei.cmu.edu/tsp/index.cfm]

Quality Focus

- To be useful, software must
  - install quickly and easily
  - run consistently
  - properly handle normal and abnormal cases
  - not do destructive or unexpected things
  - be essentially bug-free

- Defects are not important to users, as long as they do not
  - affect operations
  - cause inconvenience
  - cost time or money
  - cause loss of confidence in the program's results

Quality Assurance Goals

Maintainability

Accessibility

Installability

Performance

Correctness

Scalability

Compatibility

Concurrency

Localizability

Portability

Usability

Testability

Efficiency

Functionality

Reliability

Security

- Which Quality Assurance goals are important in your project?

Quality Assurance Methods

- **Proof** (complex theorem provers needed, only specific domains)
- **Test** (probe product with specific inputs)
- **Review** (systematic reading)
- **Metrics** (automated determination of characteristics, i.e. bugs per line of code, code coverage)

Reviews

Best Of for Scientific Software

- In a personal review
  - you privately review your product
  - your objective is to find and fix defects before test
- Reviews are most effective when they are structured and measured.
- Use reviews for requirements, designs, code, and everything else that you develop.
- Also continue to use inspections, compiling, and testing.

[http://www.sei.cmu.edu/tsp/index.cfm]

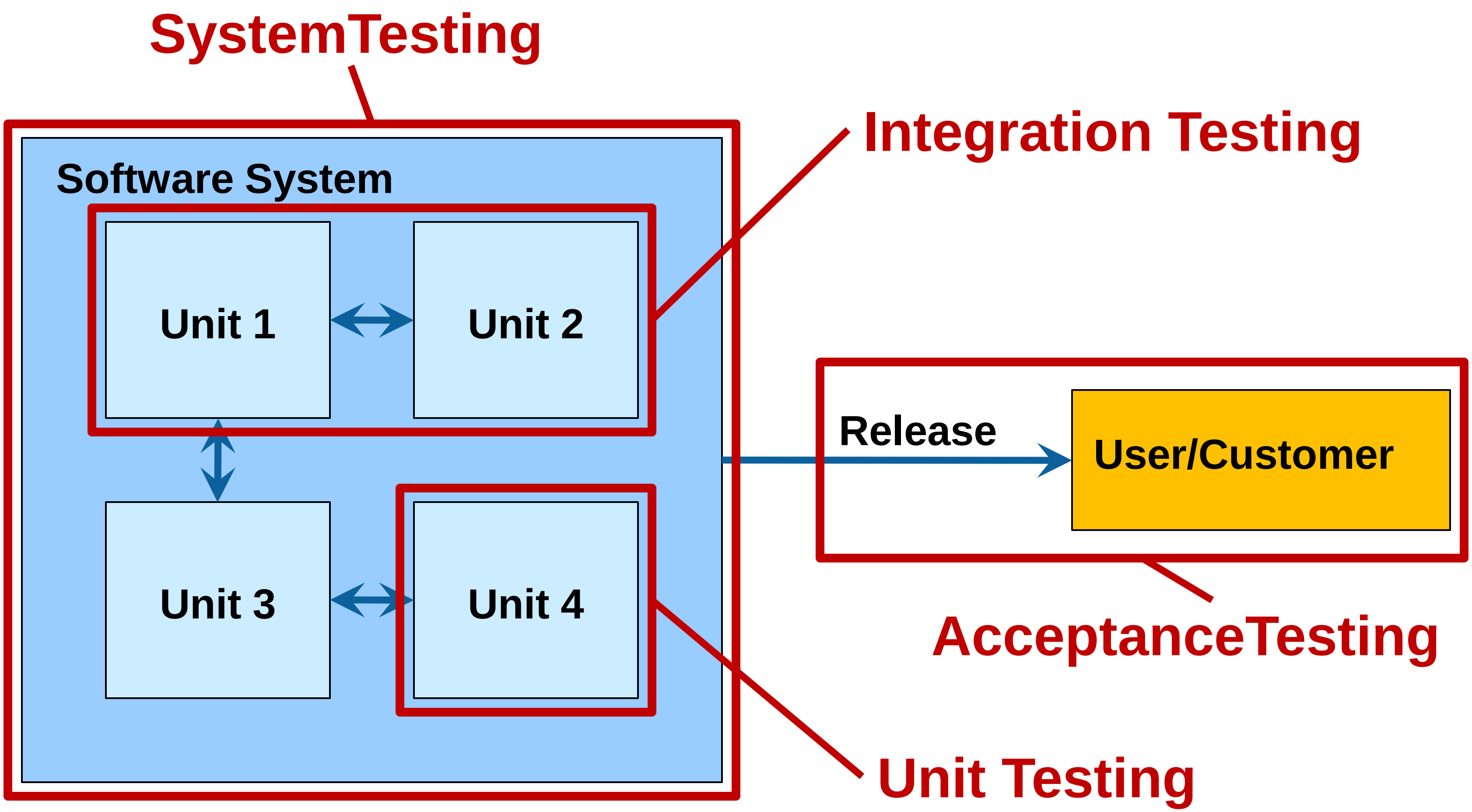
Why Reviews are Efficient

- In testing, you must
  - detect unusual behavior
  - figure out what the test program was doing
  - find where the problem is in the program
  - figure out which defect could cause such behavior
- This can take a lot of time.
- With reviews, you
  - follow your own logic
  - know where you are when you find a defect
  - know what the program should do, but did not
  - know why this is a defect
  - are in a better position to devise a correct fix

“Quality is free, but only to those who are willing to pay heavily for it.”
– T. DeMarco and T. Lister

Testing

Testing levels



Each testing level is important and should not be neclected!

Unit Testing

Best Of
for Scientific
Software

- A unit is the smallest testable part of software
  - method, function, class,…
- Several tools available
  - CppUnit (C++), CUnit (C), Junit (Java), googletest(C, C++), Check (C)
- Benefits
  - Unit testing increases confidence in changing/maintaining code
  - Codes are more reusable, since in order to make unit testing possible, codes need to be modular
  - The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels
- Tips
  - Isolate the development environment from the test environment
  - Use test data that is close to that of production
  - Before fixing a defect, write a test that exposes the defect
  - Aim at covering all paths through the unit
  - Perform unit tests continuously and frequently

Integration Testing

- The purpose of this level of testing is to expose faults in the interaction between integrated units.
- Tips
  - Ensure that the interactions between each unit are clearly defined
  - Make sure that each unit is first unit tested before you start Integration Testing
  - As far as possible, automate your tests

[http://softwaretestingfundamentals.com]

System Testing

- System Testing is a level of the software testing process where a complete, integrated system/software is tested.
- The purpose of this test is to evaluate the system's compliance with the specified requirements
- Benefits for the use in scientific software
  - Only at this level the interaction of mathematical model, numerical model and the implementation can be tested

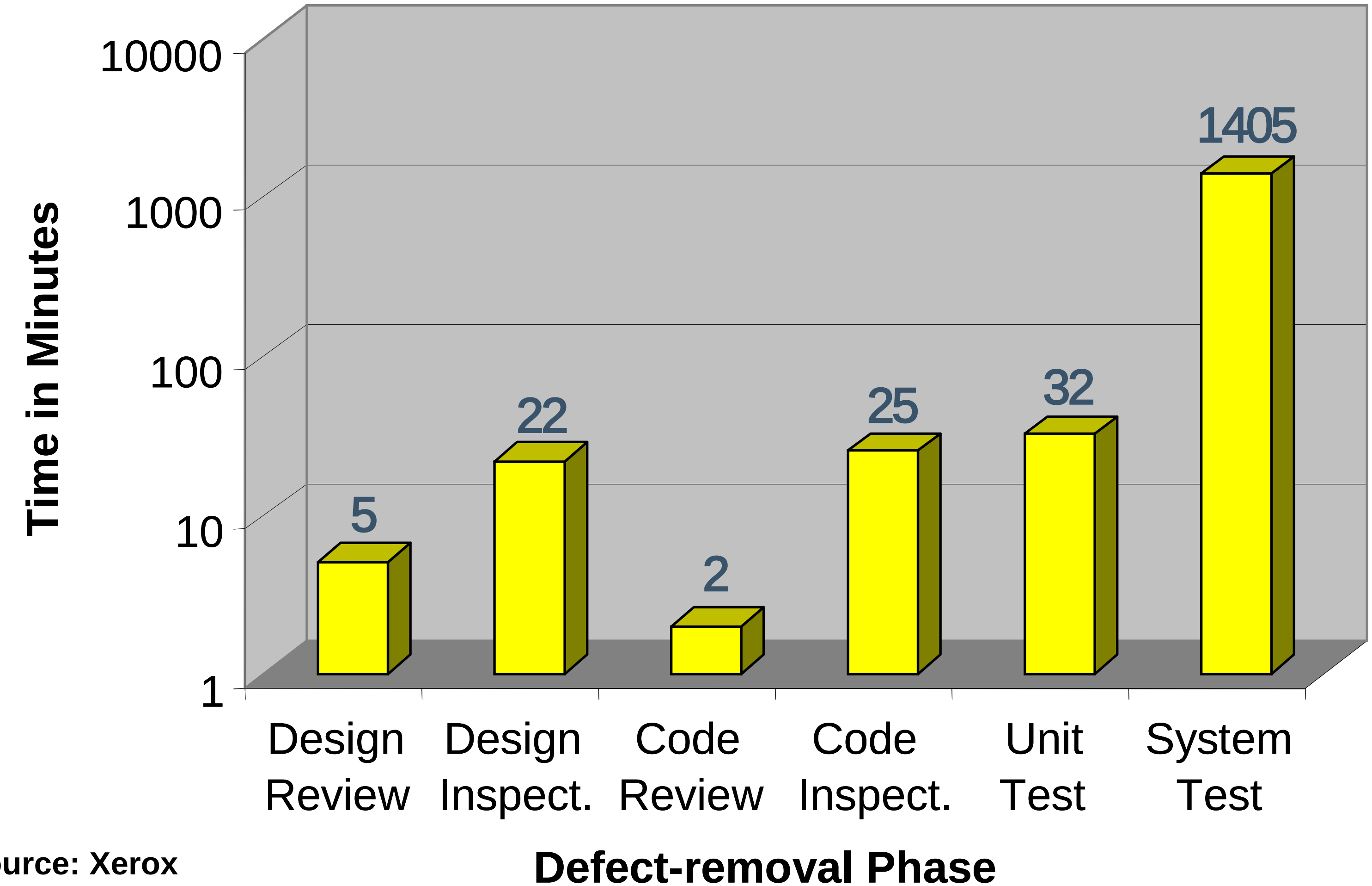
[http://softwaretestingfundamentals.com]

Acceptance Testing

- The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
- Types
  - Internal acceptance testing
  - External acceptance testing (customer, user)

[http://www.sei.cmu.edu/tsp/index.cfm]

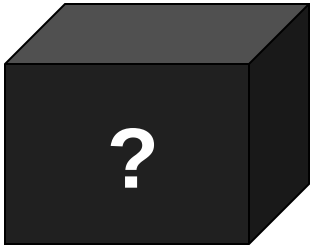
Defect-removal Times



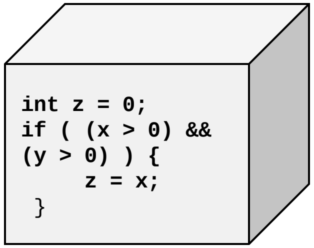
Source: Xerox

Test methods

- Black-Box:
  - Only use knowledge about the interface
  - Test the visible behaviour
  - No control of the test execution
  - Can not identify unnecessary code
  - Example: equivalence classes



- White-Box:
  - Use knowledge about the internal structure of the code
  - Control the test execution
  - Can not identify missing requirements
  - Example: code coverage testing



Test Types

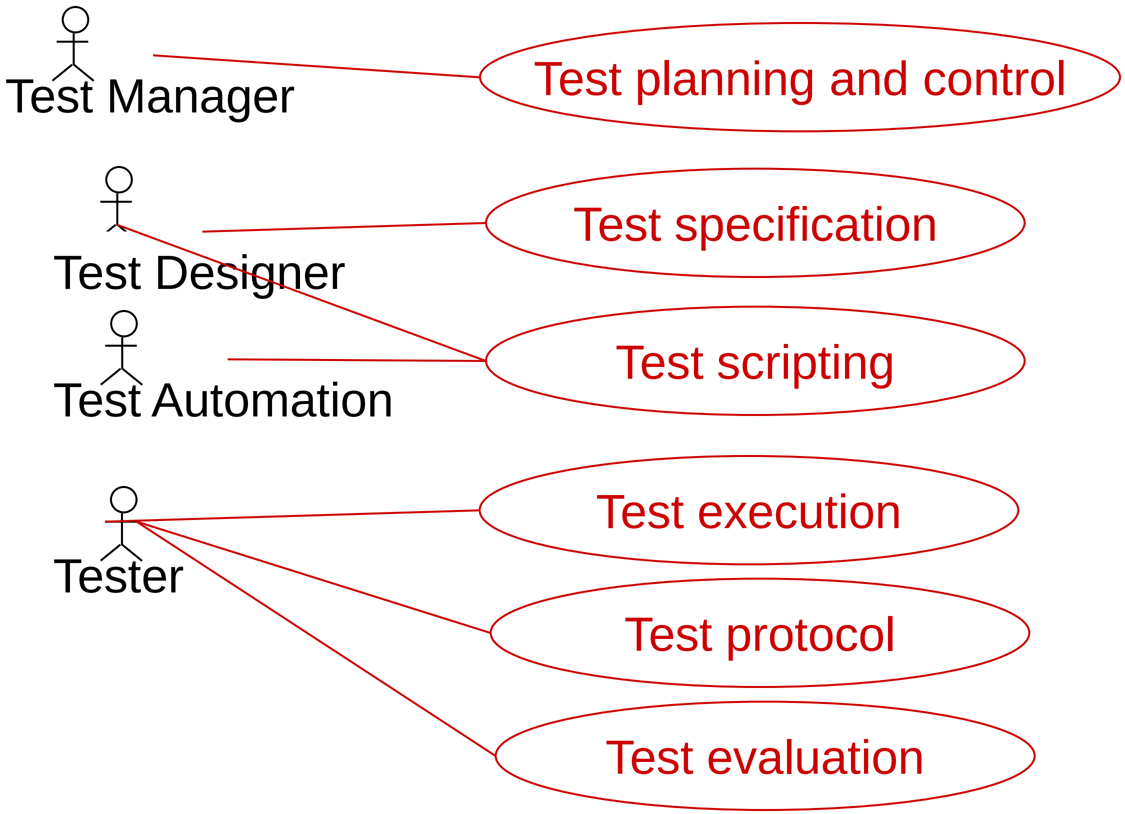
- **Smoke Testing:**
  - Covers most of the major functions of the software but none of them in depth
  - The result of this test is used to decide whether to proceed with further testing
    - If the smoke test passes, go ahead with further testing
    - If it fails, halt further tests and ask for a new build with the required fixes. If an application is badly broken, detailed testing might be a waste of time and effort.
- **Regression Testing:**
  - intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it
  - can be performed during any level of testing
  - Fix set of tests that run in a regular basis

Best Of for Scientific Software

Test specification

- Define
  - Test object (e.g. system, unit)
  - Test cases (black box or white box)
  - Test end criteria (e.g. how many % of the test cases must be successfully performed, at which defect density do you stop the testing)

Test process



Software Testing Exercise 1

Hold a pen.

Identify the types of testing you would perform on it to make sure that it is of the highest quality.

Software Testing Exercise 2

There is a simple program with the following items:

- Input Box A
- Input Box B
- Add button
- Result Text Box [=A+B]

Identify all the test cases for the program. [Example: press the Add button without entering anything in Input Box A and B]

[Ehmke 2011]

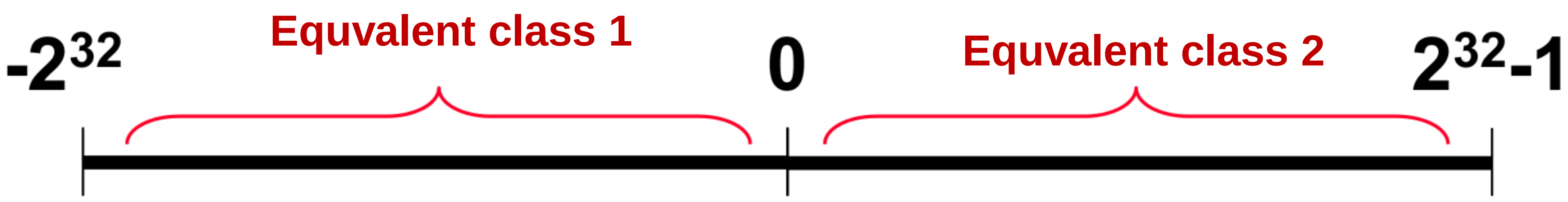
Test case selection

- Since it is impossible to test everything, how do i select a set of test cases?

```
class Trivial {
    static int sum( int a, int b) {
        return a + b;
    }
}
```

Equivalence classes

- Equivalence class: subset of all inputs which invoke similar program bevahiour
- A representative set of tests (sometimes only one) is taken from each class.
- Typical equivalence classes
  - Correct / incorrect inputs
  - Boundary values
- Gets more complicated with several input parameters



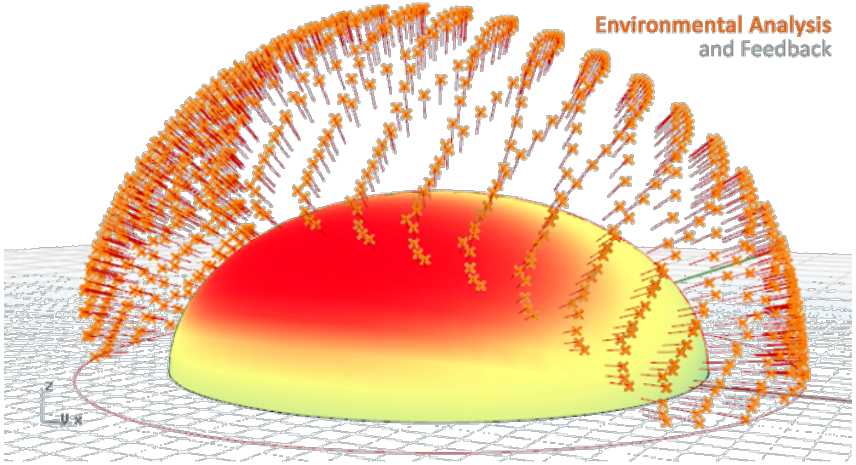
[Ehmke 2011]

Code Coverage

- Function coverage (foo(x,x))
- Statement coverage (foo(1,1))
- Decision coverage (foo(1,1), foo(1,0))
- Condition coverage (foo(1,1), foo(1,0), foo(0,0))

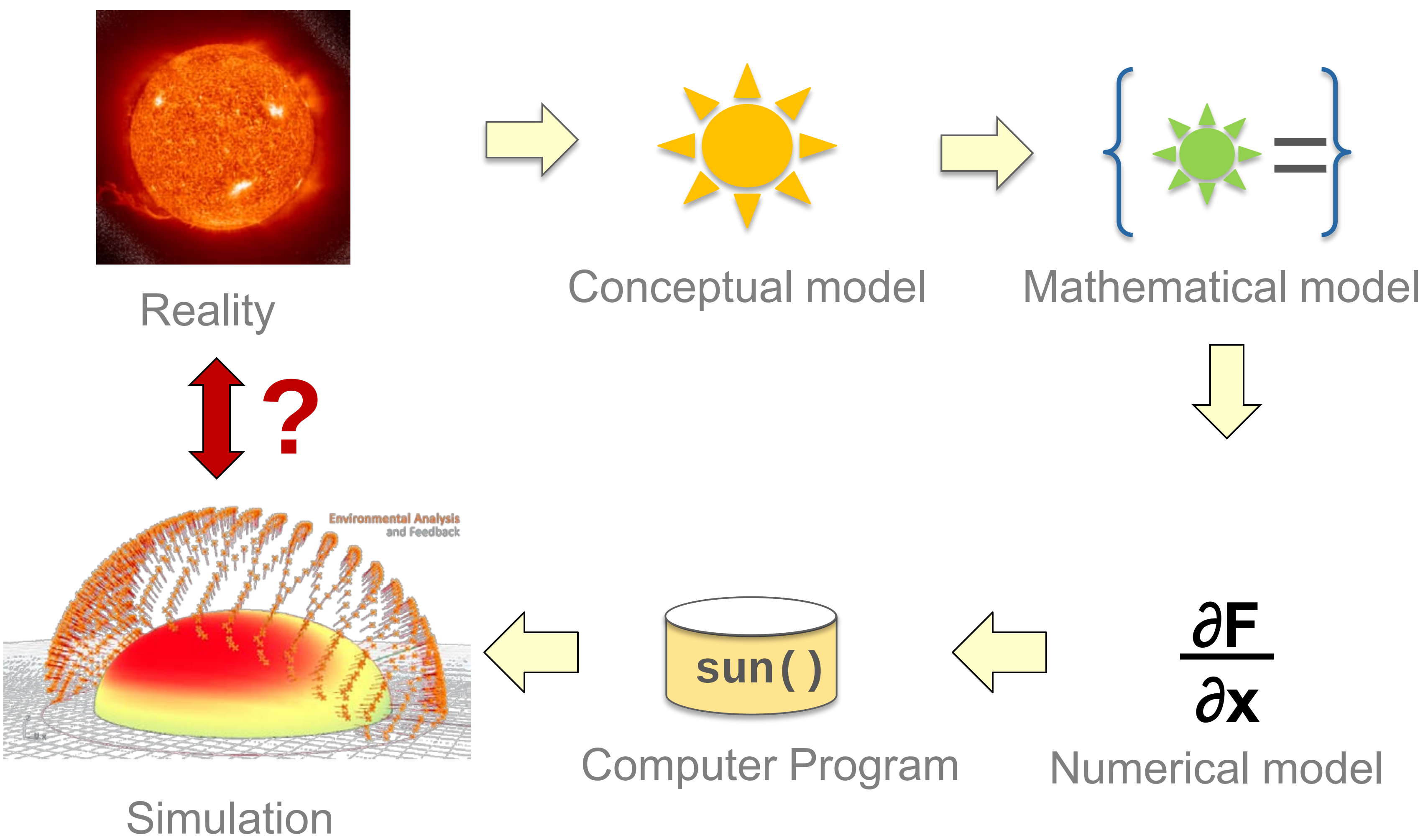
```
int foo ( int x, int y)
{
    int z = 0;
    if ( (x > 0) && (y > 0) ) {
        z = x;
    }
    return z ;
}
```

“All code is guilty, until proven innocent.”
– Anonymous



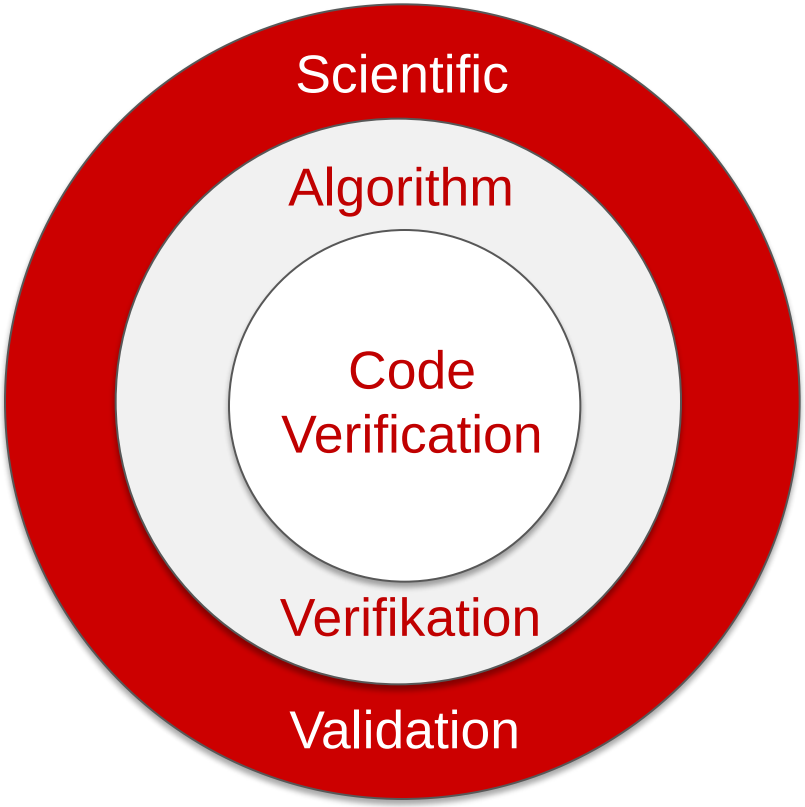
Testing Scientific Software

Possible Sources of Defects



Model for Testing Scientific Software

- 1. Code verification
  - check the program code for bugs
- 2. Algorithm verification
  - Is the implementation of the mathematical model correct?
- 3. Scientific Validation
  - Verify if the result is accurate



Reference: Hook & Kelly 2009

Example: DUNE system test environment

- Automatic regression test environment
  - Verify that development changes in the DUNE framework work in an expected way and do not break any other functionality
  - Tests run every night using the current development version of DUNE.
  - The results of the test run are published as a graphical overview on the projects web page.
  - Additionally, there is a mailing list accessible for all scientists developing DUNE informing about unsuccessful test runs.
- Supports algorithm verification and scientific validation

Code Verification

- First check the program code for bugs
  - Reviews
  - (Unit) Testing

Example DUNE:
Code Verification is done with Unit Testing

Algorithm Verification

- suitability of methods for algorithm verification strongly depends on the mathematical model used in the scientific software
  - i.e. grid convergence testing, symmetry and conservation tests
- If possible, the reference values for these mathematical quantities are determined analytically.
  - If this is not possible, like it typically isn’t for scientific software, the scientists set up these values from a scientifically validated run of the test application.

Example DUNE:

diffusionCube_dim2_level7_FEM_k2.log

```
FEM-Level=1 IT: 8
FEM-Level=1 rate of convergence: 2.6474705E-02
FEM-Level=1 gfs-globalsize: 25
FEM-Level=1 L2ERROR: 2.2732887E-03
```

Test program output

diffusionCube_dim2_level7_FEM_k2.ref

```
# Format:
# !T! <name of the value>: <value> +- <tolerance>
FEM-Level=1 IT: 8 +- 0
FEM-Level=1 rate of convergence: 2.6474705E-02 +- 1e-8
FEM-Level=1 gfs-globalsize: 25 +- 0
FEM-Level=1 L2ERROR: 2.2732888E-03 +- 1e-9
```

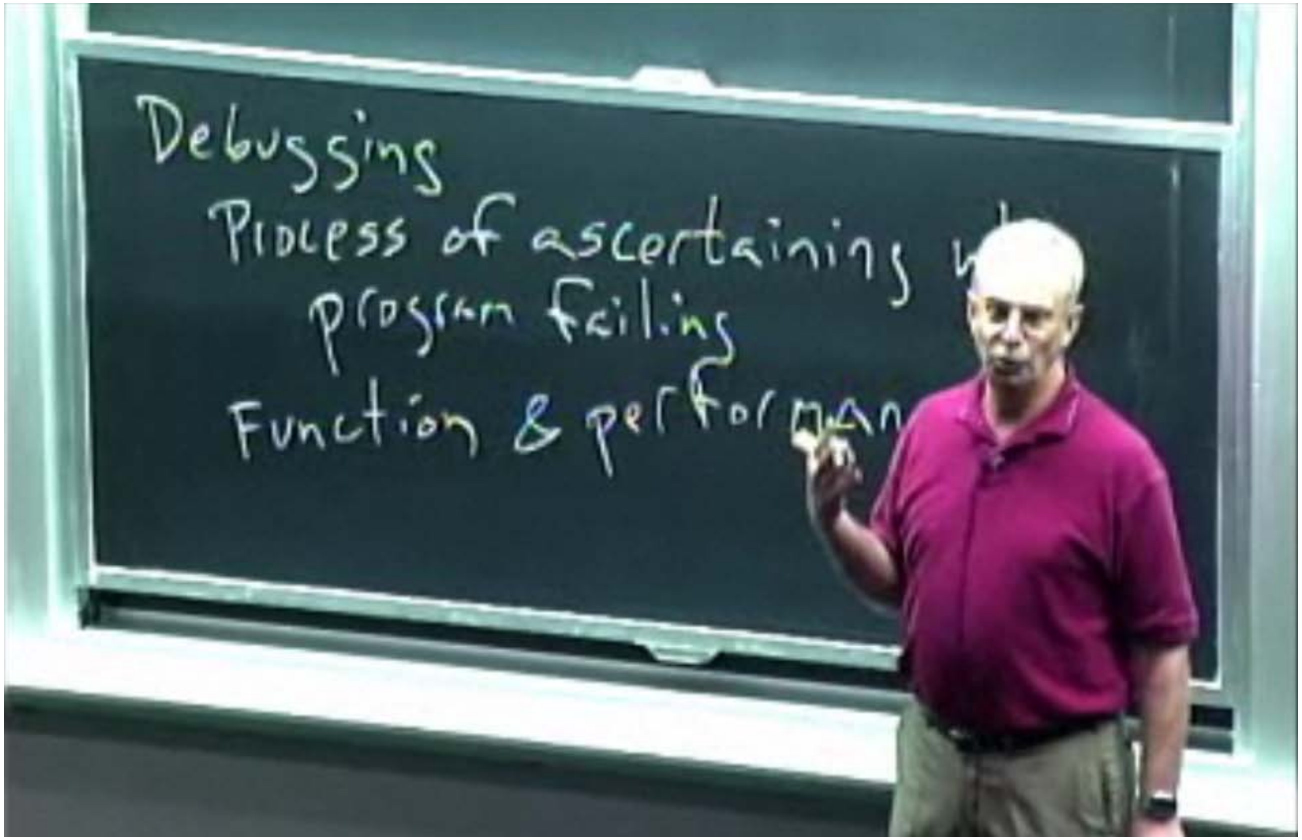
Reference file

Scientific Validation

- comparison of the graphical simulation output files
- the values in these output files are compared with according scientific validation reference files
  - Both, absolute and relative difference between the output file values and reference file values are tested.
- A change in these expected values always indicates a change in the test applications behavior.
  - Either there is a defect
  - Or the scientific software was changed in a way that a change in this specific test application was expected. In this case, the scientists can update the reference values for the test case.
- Changes in reference files always have to be scientifically justified and carefully documented.

Testing and Debugging

John Guttag, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, MIT



Testing principles (1)

- **Principle 1**
  Complete testing not possible
- **Principle 2**
  »Program testing can be used to show the presence of bugs, but never to show their absence!«? Edsger Dijkstra
- **Principle 3**
  Start early with testing (see defect removal times)
- **Principle 4**
  Defects are not evenly distributed in the code. If you have found many defects at one place, look for more.

Testing principles (2)

- **Principle 5**
  Test cases must be managed (evaluated, updated)
- **Principle 6**
  Test effort has to be adapted to the context (more for critical systems)
- **Principle 7**
  A defect-free system does not guarantee customer satisfaction

“Fast, good, cheap: pick any two.”
– Anonymous

References

- W.L. Oberkampf, T.G. Trucano, and C. Hirsch: Verification, Validation, and Predictive Capability in Computational Engineering and Physics, 2004
- www.swebok.com
- http://softwaretestingfundamentals.com/