Software Engineering
Software Quality

Klaus Ostermann

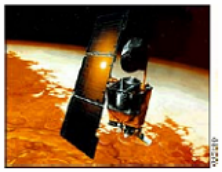
Testing

Some slides by C. Kästner, T. Ball and J. Aldrich

Why test?

Mars Climate Orbiter

- Purpose: to relay signals from the Mars Polar Lander once it reached the surface of the planet
- Disaster: smashed into the planet instead of reaching a safe orbit
- Why: Software bug - failure to convert English measures to metric values
- $165M

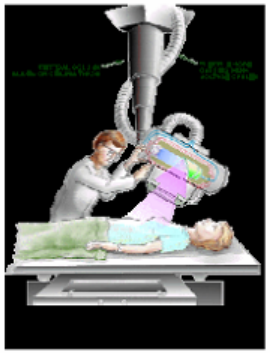


Shooting Down of Airbus 320

- 1988
- US Vicennes shot down Airbus 320
- Mistook airbus 320 for a F-14
- 290 people dead
- Why: Software bug - cryptic and misleading output displayed by the tracking software

THERAC-25 Radiation Therapy

- THERAC-25, a computer-controlled radiation-therapy machine
- 1986: two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdoses
- Why: Software bug - mishandled race condition (i.e., miscoordination between concurrent tasks)



Testing: Challenges

- Testing is a huge cost of product development
- Test effectiveness and software quality hard to measure
- Incomplete, informal and changing specifications
- Downstream cost of bugs is enormous
- Lack of spec and implementation testing tools
- Integration testing across product groups
- Patching nightmare
- Versions exploding

Example: Testing MS Word

- inputs
  - keyboard
  - mouse/pen
  - .doc, .htm, .xml, …
- outputs (WYSIWYG)
  - Printers
  - displays
  - doc, .htm, .xml, …
- variables
  - fonts
  - templates
  - languages
  - dictionaries
  - styles
- Interoperability
  - Access
  - Excel
  - COM
  - VB
  - SharePoint
- Other features
  - 34 toolbars
  - 100s of commands
  - ? dialogs

From Microsoft Office EULA…

11. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, **IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) **ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT,** THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

From GPL

- **11.** BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. **THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU**. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
- **12. IN NO EVENT** UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING **WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM** AS PERMITTED ABOVE, **BE LIABLE TO YOU FOR DAMAGES**, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES **ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The goals of testing

- Not-quite-right answers
  - Make sure it doesn't crash
  - Regression testing –no new bugs
  - Make sure you meet the spec
  - Make sure you don't have harmful side effects
- Actual goals
  - Reveal faults
  - Establish confidence
  - Clarify or represent the specification
  - No absolute certainty!

THE limitation of testing

> Testing can only show the presence of errors, not their absence
>
> - E.W. Dijkstra

Black-box Testing

- Verify each piece of functionality of the system
  - Black-box: don't look at the code
- Systematic testing
  - Test each use case
  - Test combinations of functionality (bold + italic + font + size)
    - Generally have to sample due to combinatorial explosion
  - Test incorrect user input
  - Test each "equivalence class"(similar input/output)
  - Test uncommon cases
    - Generating all error messages
    - Using uncommon functionality
  - Test borderline cases
  - Edges of ranges, overflow inputs, array of size 0 or 1

Example: Black-box Testing of Binary Search

- in/not in the array
- array with duplicate elements
- empty array, 1-element array
- even vs. odd array sizes
- unsorted/sorted array
  - Spec says array must be sorted
- Smaller or greater every element in array

White-box Testing

- Look at the code (white-box) and **try to systematically cause it to fail**
- Coverage criteria: a way to be systematic
  - Function coverage
    - Has each function been executed?
  - Statement coverage
    - Has each statement in the program been executed?
  - Edge coverage
    - Have both/all sides of each branch been taken?
  - Condition coverage
    - Has each boolean subexpression evaluated to both true and false?

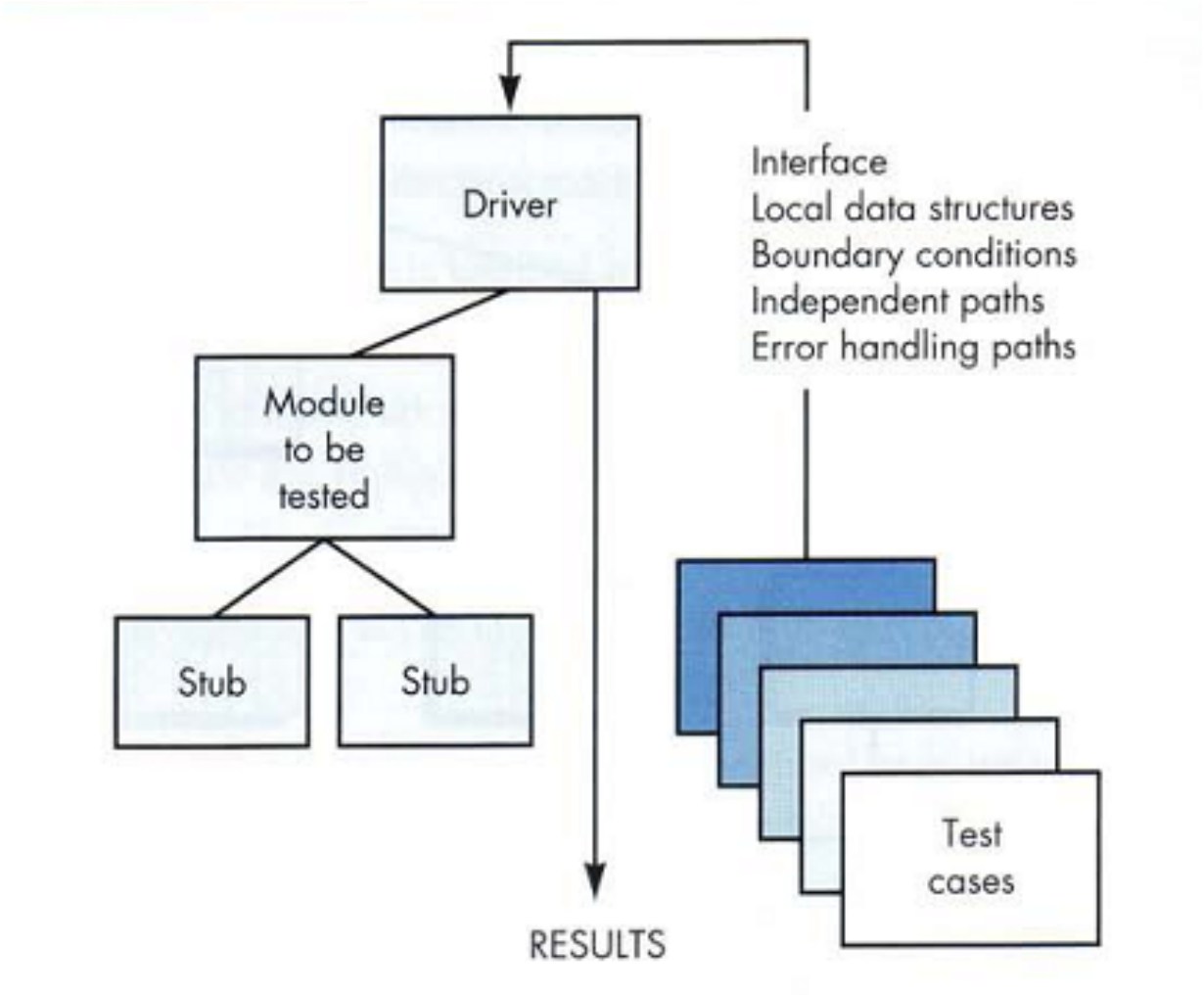
White-Box Testing

- Coverage criteria: a way to be systematic (continued)
  - Path coverage
    - Has each possible route through the code been executed?
    - Note: infinite number of paths!
    - Typical compromise: 0-1-many loop iterations
  - Exercise data structures
    - Each conceptual state or sequence of states
- Typically cannot reach 100% coverage
  - Especially true of paths, conditions
- Many tools exist to measure and visualize code coverage of tests
- Even though coverage criteria can be applied systematically, no definite conclusion about the quality or lack of bugs can be drawn from 100% XYZ-coverage
  - Dijkstra's verdict still holds

Unit Tests

- Focus on one function or module at a time
  - May need to call other functions for setup
- Usually automated
- Stubs or mock objects serve to replace modules used by the module to be tested
- A driver initializes the test environment
  - Driver and stubs/mock objects together are often called **test fixture**
- Unit tests often specified by developer
  - Always in Extreme Programming

Unit Tests


Driver
Interface
Local data structures
Boundary conditions
Independent paths
Error handling paths
Module to be tested
Stub
Stub
Test cases
RESULTS

Example Unit Test using JUnit

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Unit Tests

- The style of testing on the previous slide uses **state verification**
  - We determine whether the exercised method worked correctly by examining the state of the system under test and its collaborators after the method was exercised.
- **Mock objects** enable a different approach to testing
  - Mocks use **behavior verification**
    - check if the order made the correct calls on the warehouse.
    - Do this by telling the mock what to expect during setup and asking the mock to verify itself during verification.

Unit Tests using Mock Objects (1/2)

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";
  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());  } …
```

Unit Tests using Mock Objects (2/2)

```
public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    warehouseMock.verify();
    assertFalse(order.isFilled());
  }
```

Integration Testing (IT)

- IT is the phase in software testing in which individual software modules are combined and tested as a group
- It occurs after unit testing and before system testing
- Purpose: verify functional, performance, and reliability requirements placed on major design items
- IT uses black-box testing
- IT often structured as top-down IT or bottom-up IT
  - Top-down needs stubs, bottom-up doesn't
  - With top-down, major control functions can be tested early

Integration Testing – Top Down Approach

- Integration process is performed in a series of steps

1. Main control module is used as test driver, stubs are substituted for all components directly subordinate to main control module
2. Subordinate stubs are replaced one at a time with actual components
3. Tests are conducted as each component is integrated
4. On completion of each set of tests, another stub is replaced with the real component

Integration Testing – Bottom-up Approach

- Steps
  1. Low-level components are combined into clusters that perform a specific subfunction
  2. A driver is written to coordinate test case input and output
  3. The cluster is tested
  4. Drivers are removed and clusters are combined moving upward in the program structure

System Test

- Test entire end-to-end system functionality in black-box style
- Often organized by use cases
- Often driven by separate testing team
  - Customer / customer representative in XP
- Many different forms of system tests
  - GUI testing, Usability testing, Performance testing, Accessibility testing, Stress testing, …

Acceptance Tests

- Functional tests that the customer uses to evaluate the quality of the system

Design for Testing

- Ensure components can be tested in isolation
  - Minimize dependences on other components
  - Provide constructors to set up objects for testing
- Design techniques exist to ease testability
  - Use interfaces to allow usage of mock objects or stubs
  - "Dependency Injection"
- Some PLs provide support for testing
  - AspectJ is frequently used for testing

Test-driven Development (TDD)

- **Goal**:
  - have enough unit tests
  - check they're effective

Design for testing: TDD

- **Method**: to develop a program fragment
  1. Write a test
  2. Stub the functionality
  3. Ensure that the test actually fails – if not, the test is not restrictive, fix it!
  4. Implement enough functionality for the test to start passing, but **no more**
  5. Iterate by adding more tests
  6. Stop when tests force the desired behavior to be implemented

Design for testing: TDD

Result:

- we get more confidence that
  - all functionality is tested, because we don't implement anything which is not tested!
  - tests actually check what they should!
- tests are a form of specification (especially in BDD, a variant of TDD)
- More test code, thus also more code to maintain
  - There are techniques to ease maintenance

- **But** again, no absolute guarantee

Design by Contract

- General meaning
  - Specify a contract between client and implementation of a module
  - Using pre- and post-conditions
  - System works if both parties fulfill their contract
- Specific setting of testing
  - Verify pre-and post-conditions while running
  - Assign blame based on which one fails
  - Turns a system execution into a set of unit tests

Example: Design by Contract using the Java Modeling Language (JML)

```
/*@
   @ public normal_behavior
   @   requires ! isEmpty();
   @   ensures
   @     elementsInQueue.equals(((JMLObjectBag)
   @          \old(elementsInQueue))
   @                          .remove(\result)) &&
   @     \result.equals(\old(peek()));
   @*/
Object pop() throws NoSuchElementException;
```

Contracts are checked dynamically if the code is compiled with the JML compiler

Regression Testing

- A suite of tests is run every time the system changes
- Goal: to catch any (?) new bugs introduced by change
  - Need to add tests for new functionality
  - But still test the old functionality also!
  - Note: in some cases, old test cases should return a different result, depending on the change that was made

Nightly Builds

- Building a release of a large project every night
  - Catches integration problems where a change "breaks the build"
  - Breaking the build is a BIG deal—may result in midnight calls to the responsible engineer
- Typically, run regression test after building
  - Plot progress on tests over time

> "Treat the daily build as the heartbeat of the project. If there is no heartbeat, the project is dead." - Jim McCarthy

Add tests for each defect fixed!

- If existing tests don't already cover the defect
  - e.g., it was not found through tests.
- **Goal:**
  - To check that the defect is actually fixed
  - To prevent the defect from being reintroduced

When are you done testing?

- Most common
  - Run out of time or money
- Can try to use statistical models
  - Only as good as your characterization of the input
  - Which is often quite bad
  - Exception: stable systems for which you have empirical data (telephones)
  - Exception: good mathematical model (avionics)
- Can seed faults
  - Halt when an “adequate” percentage is found
  - Implication: same percentage of unknown errors found
  - But is this really true?
- Rule of thumb: when error detection rate drops

Testing Quality Attributes

- Throughput
  - Increase load steadily through a series of tests until performance is unacceptable
  - Load profile should match actual operation profile of system
  - “Stress testing” tests the system beyond intended design limits
  - Look at failure behavior
  - Identify defects related to heavy load

Testing Quality Attributes

- Reliability
  - Run for a period of time against operational profile, estimate reliability metric
  - Challenges:
    - Hard to know correct profile
    - Expensive to generate profile
    - Need large test cases to generate statistical confidence
    - Which is irrelevant anyway if the profile is off
  - Basically no good way to do this
  - Alternative: stress testing, again

Testing Quality Attributes

- Fault tolerance
  - Programmatically cause a fault and test that the system can recover
- Security
  - Attack team
- Usability
  - Measure user performance on some task
- Portability
  - Test against multiple platforms
- Evolvability
  - Design extension

Defect Tracking

- Organized handling of defects
  - Defect description
  - Problem analysis
  - Product and version affected
  - Originator, Owner
  - Status: open, confirmed, closed
  - Severity
  - Date reported, fixed
- Widely used in open source, industry
  - Tools like Bugzilla

Test Plan

- Strategy
  - Unit? Functional? White/Black box? Design by contract?
  - During requirements? Before coding? During test phase?
  - Quality attribute testing?
  - Nightly builds?
  - Completeness criterion?
- Document acceptance tests
  - Trace each requirement to one or more acceptance tests
- Tools
  - Generation? Regression? Selection? Coverage? Defect tracking?
- People
  - Developer or dedicated testers?

Code Reviews

Reviews and Inspections

- A family of techniques
  - Pair Programming
  - Walkthroughs
  - Inspections
  - Personal reviews
  - Formal technical reviews
- Review / inspect
  - To examine closely
  - With an eye toward correction or appraisal
- People (peers) are the examiners

Why do code reviews?

- Catching errors
  - Sooner
  - More and different
- Improving communication
  - Crossing organization boundaries
- Providing education
- Making software visible

Results

- Catching most errors before test
- Review plus test is much cheaper than just test
  - Sample results:
  - 10x reduction in errors reaching test
  - 50 -80 % total cost reduction
- Fewer defects after release
- Substantial cost savings in maintenance
  - Supported by study at HP (R. Grady)
  - Testing efficiency (defects found / hour)
    - System use 0.21
    - Black box 0.282
    - White box 0.322
    - Reading/inspect 1.057

Personal Review

- Features
  - Informal
  - Done by the producer
- Implications
  - Not as objective
  - Available to any developer
  - Different mindset limits screening efficiency
    - Need for review
    - Product completion

Pair Programming

- Features
  - Two programmers work together at one work station
  - One types in code while the other reviews each line of code as it is typed
  - These two roles are switched frequently
- Implications
  - Knowledge passes between programmers – with "promiscuous" pairing through the whole team
  - Studies found that pair programming decreases defects and improves discipline and productivity
  - No preparation required, default way of coding in Extreme Programming

Walkthroughs

- Features
  - Less formal
  - Producer presents or provides information
- Implications
  - Larger groups can attend (education)
  - More material per meeting
  - Less preparation time
  - Harder to separate explanation and justification, product and presenter
- IEEE 1028 recommends three specialist roles:
  - The **Author** - presents the software product in step-by-step manner at the walk-through meeting, and is probably responsible for completing most action items;
  - The **Walkthrough Leader** - conducts the walkthrough, handles administrative tasks, and ensures orderly conduct (and who is often the Author)
  - The **Recorder** - notes all anomalies (potential defects), decisions, and action items identified during the walkthrough meetings.

Inspections

- Features
  - Team reviews materials separately
  - Team and producers meet to discuss
  - May review selected product aspects only
- Implications
  - Focus on important issues
  - If you know what they are
  - More material per meeting
  - Less preparation time

Review before merging

- Each change must be reviewed before acceptance
- Pros: higher-quality changes
  - More defects found
  - The author is more careful
  - and documents the code better
- Cons:
  - slower development (?)
  - risk of ego problems (to manage)
- Used for instance at Google and in good Open Source projects

Formal Technical Review

- Features
  - Formal
  - Scheduled event
  - Defined procedure
  - Reported result
  - Technical
    - Not schedule
    - Not budget
  - Independent review team
  - Producers not present

Formal Technical Review

- Implications
  - More preparation time
  - Less material per meeting
  - Product must stand or fall on its own

Review Report

- Purpose
  - Tell managers the outcome
  - Early warning system for major problems
  - Provide historical record
    - For process improvement
    - For tracking people involved with projects
- Contents
  - Summary
  - Product issues
  - Other related issues

Summary

- Code Reviews are a highly effective technique to improve software quality
  - And many other beneficial side effects
- Not used nearly enough
- Do it!
  - Personal reviews, Pair programming are applicable in almost every context
  - Walkthroughs for student projects