Software Architecture
Bertrand Meyer, Michela Pedroni

ETH Zurich, February-May 2010

Lecture 7:
Quality Assurance and testing
**Topics**

- Testing basics
- Partition testing
- Measure test quality
- Unit testing and test driven development
- GUI testing
- Test management
- Debugging
Testing Basics
Definition: software quality assurance (QA)

A set of policies and activities to:

- **Define** quality objectives
- **Help ensure** that software products and processes meet these objectives
- **Assess** to what extent they do
- **Improve** them over time
Software quality

Product quality (immediate):
  Correctness
  Robustness
  Security
  Ease of use
  Ease of learning
  Efficiency

Product quality (long-term):
  Extendibility
  Reusability
  Portability

Process quality:
  Timeliness
  Cost-effectiveness
  Self-improvement
Quality, defined negatively

Quality is the absence of “deficiencies” (or “bugs”).

More precise terminology (IEEE):

- **Mistakes**
- **Faults**
- **Failures**

result from **Failures**
caused by **Mistakes**

**Example: A Y2K issue**

Failure: person’s age appears as negative!
Fault: code for computing age yields negative value if birthdate is in 20th century and current date in 21st
Mistake: failed to account for dates beyond 20th century

Also: **Error**

In the case of a failure, extent of deviation from expected result
What is a failure?

For this discussion, a failure is any event of system execution that violates a stated quality objective.
Why does software contain faults?

We make mistakes:
- Unclear requirements
- Wrong assumptions
- Design errors
- Implementation errors

Some aspects of a system are hard to predict:
- For a large system, no one understands the whole
- Some behaviors are hard to predict
- Sheer complexity

Evidence (if any is needed!):
Widely accepted failure of “n-version programming”
The need for independent QA

Deep down, we want our software to succeed.

We are generally not in the best position to prevent or detect errors in our own products.
Definition: testing

To test a software system is to try to make it fail
“Testing can only show the presence of errors, never their absence”

(Edsger W. Dijkstra, in *Structured Programming*, 1970, and a few other places)

1. Gee, too bad, I hadn’t thought of this. I guess testing is useless, then?
2. Wow! Exciting! Where can I buy one?
Limits of testing

**Theoretical:** cannot test for termination

**Practical:** sheer number of cases

(Dijkstra’s example: multiplying two integers; today would mean $2^{128}$ combinations)
Consequences of the definition

- The purpose of testing is to find “bugs”
  (More precisely: to provoke failures, which generally reflect faults due to mistakes)

- We should really call a test “successful” if it fails
  (We don’t, but you get the idea)

- A test that passes tells us nothing about the reliability of the Unit Under Test (UUT)
  (except if it previously failed (regression testing))

- A thorough testing process must involve people other than developers
  (although it may involve them too)

- Testing stops at the identification of bugs
  (it does not include correcting them: that’s debugging)
Testing: the overall process

- Identify parts of the software to be tested
- Identify interesting input values
- Identify expected results (functional) and execution characteristics (non-functional)
- Run the software on the input values
- Compare results & execution characteristics to expectations
Testing, the ingredients: test definition

**Implementation Under Test (IUT)**

The software (& possibly hardware) elements to be tested

**Test case**

Precise specification of one execution intended to uncover a possible fault:

- Required state & environment of *IUT* before execution
- Inputs

**Test run**

One execution of a *test case*

**Test suite**

A collection of *test cases*
More ingredients: test assessment

Expected results (for a test case)
Precise specification of what the test is expected to yield in the absence of a fault:
- Returned values
- Messages
- Exceptions
- Resulting state of program & environment
- Non-functional characteristics (time, memory...)

Test oracle
A mechanism to determine whether a test run satisfies the expected results
- Output is generally just “pass” or “fail”.
**Classification: by scope**

**Unit test:** tests a module

**Integration test:** tests a complete subsystem

- Exercises interfaces between units, to assess whether they can operate together

**System test:** tests a complete, integrated application against the requirements

- May exercise characteristics present only at the level of the entire system

*We cannot connect our email client to their database driver because ours is written in Eiffel and theirs is written in Java.*
Classification: by intent

Functional testing
   Goal: evaluate the system's compliance with its specified requirements.

Fault-directed testing
   Goal: reveal faults through failures
   ➢ Unit and integration testing

Conformance-directed testing
   Goal: assess conformance to required capabilities
   ➢ System testing

Acceptance testing
   Goal: enable customer to decide whether to accept a product

Regression testing
   Goal: Retest previously tested element after changes, to assess whether they have re-introduced faults or uncovered new ones.

Mutation testing
   Goal: Introduce faults to assess test case quality
Alpha and beta testing

Alpha testing
The first test of newly developed hardware or software in a laboratory setting. When the first round of bugs has been fixed, the product goes into beta test with actual users.

Beta testing
A test of new or revised hardware or software that is performed by users at their facilities under normal operating conditions.
An interesting example: proportional testing of Gmail.
White-box testing

- To define test cases, source code of IUT is available

Alternative names:
implementation-based, structural, “glass box”, “clear box”

Black-box testing

- Properties of IUT available only through specification

Alternative names:
responsibility-based, functional
## A comparison

<table>
<thead>
<tr>
<th></th>
<th>White-box</th>
<th>Black-box</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IUT internals</strong></td>
<td>Knows internal structure &amp; implementation</td>
<td>No knowledge</td>
</tr>
<tr>
<td><strong>Focus</strong></td>
<td>Ensure coverage of many execution possibilities</td>
<td>Test conformance to specification</td>
</tr>
<tr>
<td><strong>Origin of test cases</strong></td>
<td>Source code analysis</td>
<td>Specification</td>
</tr>
<tr>
<td><strong>Typical use</strong></td>
<td>Unit testing</td>
<td>Integration &amp; system testing</td>
</tr>
<tr>
<td><strong>Who?</strong></td>
<td>Developer</td>
<td>Developers, testers, customers</td>
</tr>
</tbody>
</table>
Input Partitioning
Limits of testing - revisited

**Theoretical**: cannot test for termination

**Practical**: sheer number of cases

(Dijkstra’s example: multiplying two integers; today would mean $2^{128}$ combinations)

Problem: Exhaustive testing is impractical

Solution: Select representative input samples, but how?
We cannot test all inputs, but need realistic inputs

Idea of partition testing: select elements from a \textit{partition} of the input set, i.e. a set of subsets that is

\begin{itemize}
  \item \textit{Complete}: union of subsets covers entire domain
  \item \textit{Pairwise disjoint}: no two subsets intersect
\end{itemize}

Purpose (or hope!): 

\begin{itemize}
  \item For any input value that produces a failure, some other in the same subset produces a similar failure
\end{itemize}

Common abuse of language: “a partition” for “one of the subsets in the partition” (e.g. $A_2$)

\begin{itemize}
  \item Better called “equivalence class”
\end{itemize}
Examples of partitioning strategies

Ideas for equivalence classes:

- Set of values so that if any is processed correctly then any other will be processed correctly
- Set of values so that if any is processed incorrectly then any other in set will be processed incorrectly
- Values at the center of a range, e.g. 0, 1, -1 for integers
- Boundary values, e.g. MAXINT
- Values known to be particularly relevant
- Values that must trigger an error message ("invalid")
- Intervals dividing up range, e.g. for integers
- Objects: need notion of "object distance"
Example partitioning

Date-related program

- **Month**: 28, 29, 30, 31 days
- **Year**: leap, standard non-leap, special non-leap (x100), special leap (x1000)

All combinations: some do not make sense

From Wikipedia:

The [Gregorian calendar](https://en.wikipedia.org/wiki/Gregorian_calendar), the current standard calendar in most of the world, adds a 29th day to [February](https://en.wikipedia.org/wiki/February) in all years evenly divisible by four, except for centennial years (those ending in -00), which receive the extra day only if they are evenly divisible by 400. Thus 1600, 2000 and 2400 are leap years but 1700, 1800, 1900 and 2100 are not.
Boundary testing

Many errors occur on or near boundaries of input domain

Heuristics: in an equivalence class, select values at edge

Examples:
- Leap years
- Non-leap commonly mistaken as leap (1900)
- Leap years commonly mistaken as non-leap (2000)
- Invalid months: 0, 13
- For numbers in general: 0, very large, very small
Partition testing: assessment

Applicable to all levels of testing: unit, class, integration, system

Black-box: based only on input space, not the implementation

A natural and attractive idea, applied formally or by many testers, but lacks rigorous basis for assessing effectiveness.
Measure Test Quality
Coverage (white-box technique)

Idea: to assess the effectiveness of a test suite, 
Measure how much of the program it exercises.

Concretely:
- Choose a kind of program element, e.g. instructions (instruction coverage) or paths (path coverage)
- Count how many are executed at least once
- Report as percentage

A test suite that achieves 100% coverage achieves the chosen criterion. Example:
“*This test suite achieves instruction coverage for routine r*”

Means that for every instruction \(i\) in \(r\), at least one test executes \(i\).
Taking advantage of coverage measures

Coverage-guided test suite improvement:

- Perform coverage analysis for a given criterion
- If coverage < 100%, find unexercised code sections
- Create additional test cases to cover them

The process can be aided by a coverage analysis tool:

1. Instrument source code by inserting trace instructions
2. Run instrumented code, yielding a trace file
3. From the trace file, analyzer produces coverage report

High coverage /= high quality.
Coverage criteria

Instruction (or: statement) coverage:
  Measure instructions executed
Disadvantage: insensitive to some control structures

Branch coverage:
  Measure conditionals whose paths are both executed

Condition coverage:
  Count how many atomic boolean expressions evaluates
to both true and false

Path coverage:
  Count how many of the possible paths are taken
(Path: sequence of branches from routine entry to exit)
```java
public class Account {

    private int balance;

    public void withdraw(int sum) {
        if (balance >= sum) {
            balance = balance - sum;
            if (balance == 0)
                System.out.println("The account is now empty.");
        } else
            System.out.println("There are less than " + sum + "CHF in the account.");
    }

    ...
}
```
Example: instruction coverage

```java
public class Account {

    private int balance;

    public void withdraw(int sum) {
        if (balance >= sum) {
            balance = balance - sum;
            if (balance == 0)
                System.out.println("The account is now empty.");
        } else
            System.out.println("There are less than " + sum + "CHF in the account.");
    }
}

TC1:
    a = new Account();
a.setBalance(100);
a.withdraw(1000);

TC2:
    a = new Account();
a.setBalance(100);
a.withdraw(100);
```
Example: branch (condition, path) coverage

```java
class Account {
    private int balance;

    public void withdraw(int sum) {
        if (balance >= sum) {
            balance = balance - sum;
            if (balance == 0)
                System.out.println("The account is now empty.");
        } else
            System.out.println("There are less than " + sum + " CHF in the account.");
    }
}
```

**TC1:**
```java```
a = new Account();
a.setBalance(100);
a.withdraw(1000);
```java```

**TC2:**
```java```
a = new Account();
a.setBalance(100);
a.withdraw(100);
```java```

**TC3:**
```java```
a = new Account();
a.setBalance(100);
a.withdraw(99);
```java```
Specification coverage

**Predicate** = an expression that evaluates to a boolean value
- e.g.: \( a \lor b \lor (f(x) \land x > 0) \)

**Clause** = a predicate that does not contain any logical operator
- e.g.: \( x > 0 \)

If specification expressed as predicates on the state, specification coverage translates to **predicate coverage**.
A predicate is covered iff it evaluates to both true and false in 2 different runs of the system.

Example:

\[ a \lor b \lor (f(x) \land x > 0) \]

is covered by the following 2 test cases:

- \{a=true; b=false; f(x)=false; x=1\}
- \{a=false; b=false; f(x)=true; x=-1\}
Clause coverage (CC)

Satisfied if every clause of a certain predicate evaluates to both true and false.

Example:
\[ x > 0 \lor y < 0 \]

Clause coverage is achieved by:
- \{x=-1; y=-1\}
- \{x=1; y=1\}
Combinatorial coverage (CoC)

Every combination of evaluations for the clauses in a predicate must be achieved.

Example:

\[ ((A \lor B) \land C) \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>((A \lor B) \land C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Mutation testing

Idea: make small changes to the program source code (so that the modified versions still compile) and see if your test cases fail for the modified versions

Purpose: estimate the quality of your test suite
Faulty versions of the program = mutants
   ➢ We only consider mutants that are not equivalent to the original program!

A mutant is said to be killed if at least one test case detects the fault injected into the mutant

A mutant is said to be alive if no test case detects the injected fault
Mutation operators

**Mutation operator** = a rule that specifies a syntactic variation of the program text so that the modified program still compiles

**Mutant** = the result of an application of a mutation operator

The quality of the mutation operators determines the quality of the mutation testing process.

**Mutation operator coverage (MOC)**: For each mutation operator, create a mutant using that mutation operator.
Examples of mutants

Original program:

```plaintext
if (a < b)
    b := b - a;
else
    b := 0;
```

Mutants:

```plaintext
if (a < b)
    if (a <= b)
    if (a > b)
    if (c < b)
        b := b - a;
    b := b + a;
    b := x - a;
else
    b := 0;
    b := 1;
    a := 0;
```
Mutations operators (classical)

- Replace arithmetic operator by another
- Replace relational operator by another
- Replace logical operator by another
- Replace a variable by another
- Replace a variable (in use position) by a constant
- Replace number by absolute value
- Replace a constant by another
- Replace “while... do...” by “repeat... until...”
- Replace condition of test by negation
- Replace call to a routine by call to another
OO mutation operators

Visibility-related:

- **Access modifier change** - changes the visibility level of attributes and methods

Inheritance-related:

- **Hiding variable/method deletion** - deletes a declaration of an overriding or hiding variable/routine
- **Hiding variable insertion** - inserts a member variable to hide the parent's version
Polymorphism- and dynamic binding-related:

- Constructor call with child class type - changes the dynamic type with which an object is created

Various:

- Argument order change - changes the order of arguments in routine invocations (only if there exists an overloading routine that can accept the changed list of arguments)
- Reference assignment and content assignment replacement
  - example: list1 := list2 ->
    list1 := list2.clone()
Unit Testing
unit testing is a software verification and validation method in which a programmer tests if individual units of source code are fit for use. A unit is the smallest testable part of an application.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy.

Unit tests find problems early in the development cycle. Ideally, each test case is independent from the others.
Components of a test case

- Input generation
- Test case execution
- Result validation
xUnit – widely used testing frameworks

xUnit frameworks allow testing of different elements (units) of software, such as functions and classes. The main advantage of xUnit frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test.

Examples

- JUnit for Java
- N NUnit for .NET
- CppUnit for C++
JUnit: Overview

Provides a framework for running test cases

Test cases
- Written manually
- Normal classes, with annotated methods

Input values and expected results defined by the tester

Execution is the only automated step
How to use JUnit

Requires JDK 5

Annotations:
- `@Test` for every routine that represents a test case
- `@Before` for every routine that will be executed before every `@Test` routine
- `@After` for every routine that will be executed after every `@Test` routine

Every `@Test` routine must contain some check that the actual result matches the expected one - use `asserts` for this
- `assertTrue`, `assertFalse`, `assertEquals`, `assertNull`, `assertNotNull`, `assertSame`, `assertNotSame`
package unitests;

import org.junit.Test; // for the Test annotation
import org.junit.Assert; // for using asserts
import junit.framework.JUnit4TestAdapter; // for running

import ch.ethz.inf.se.bank.*;

public class AccountTest {

    // To declare a routine as a test case
    @Test
    public void initialBalance() {
        Account a = new Account("John Doe", 30, 1, 1000);
        Assert.assertEquals("Initial balance must be the one set through the constructor", 1000, a.getBalance());
    }

}

// To compare the actual result to the expected one
Example: set up and tear down

```java
package unittests;

import org.junit.Before; // for the Before annotation
import org.junit.After; // for the After annotation
// other imports as before...

public class AccountTestWithSetUpTearDown {
    private Account account;

    @Before
    public void setUp() {
        account = new Account("John Doe", 30, 1, 1000);
    }

    @After
    public void tearDown() {
        account = null;
    }

    @Test
    public void initialBalance() {
        Assert.assertEquals("Initial balance must be the one set through the constructor", 1000, account.getBalance());
    }
}
```

Must make account an attribute of the class now

To run this routine before any @Test routine

To run this routine after any @Test routine
@BeforeClass, @AfterClass

A routine annotated with @BeforeClass will be executed once, before any of the tests in that class is executed.

A routine annotated with @AfterClass will be executed once, after all of the tests in that class have been executed.

Can have several @Before and @After methods, but only one @BeforeClass and @AfterClass routine respectively.
Checking for exceptions

Pass a parameter to the `@Test` annotation stating the type of exception expected:

```java
@Test(expected=AmountNotAvailableException.class)
public void overdraft () throws AmountNotAvailableException {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.withdraw(1001);
}
```

The test will fail if a different exception is thrown or if no exception is thrown.
Pass a parameter to the `@Test` annotation setting a timeout period in milliseconds. The test fails if it takes longer than the given timeout.

```java
@Test(timeout=1000)
public void testTimeout () {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.infiniteLoop();
}
```
Test-driven development (TDD)

Software development methodology
One of the core practices of extreme programming (XP)
Write test, write code, refactor
More explicitly:

1. Write a small test.
2. Write enough code to make the test succeed.
3. Clean up the code.
4. Repeat.

Always used together with xUnit.
Test-Driven Development (TDD)

Evolutionary approach to development

Combines

- Test-first development
- Refactoring

Primarily a method of software design

- Not just method of testing
TDD 1: Test-First Development (TFD)
A change to the system that leaves its behavior unchanged, but enhances some non-functional quality:

- Simplicity
- Understandability
- Performance

Refactoring does not fix bugs or add new functionality.
Examples of refactoring

- Change the name of a variable, class, ...
- Convert local variable to attribute
- Generalize type
- Introduce argument
- Turn a block of code into a routine
- Replace a conditional with polymorphism
- Break down large routine
TDD = TFD + Refactoring

- Apply test-first development
- Refactor whenever you see fit (before next functional modification)
TDD and extreme programming (XP)

- Easy to give in and not write a test or skip a refactoring.
- Pair-programming partner can help keep you on track.
- Write testable code.
TDD: Kent Beck’s rules

• Write new business code only when a test fails
• Eliminate any duplication you find
TDD: consequences for the developer

- You design organically, running code provides feedback between decisions.
- You write your own tests, because you cannot wait.
- Development environment must provide rapid response to small changes.
- Your design must be consist of highly cohesive, loosely coupled components to make testing easier.
- Side effect: easier evolution and maintenance.
Developers must learn to write good unit tests:

- Run fast (short setup, run, and tear-down)
- Run in isolation (reordering is possible)
- Use data that makes test cases easy to read
- Use real data when needed
- Each test case is one step towards overall goal
TDD & traditional testing

TDD is a programming technique that ensures that source code is thoroughly unit tested.

Need remains for:

- Nonfunctional testing
- User acceptance testing
- System integration testing

XP suggests these tests should also occur early.
Failed test case is a success.

TDD guarantees complete statement coverage (per definition).

Traditional testing only recommends it.
Programmers often do not read documentation. Instead, they look for examples and play with them.

Good unit tests can serve as

- Examples
- Documentation
Bob Martin:

"The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function."

Contracts serve a very similar purpose.

Write header comment and contract before implementation.

Symbiosis:

- Tests make system run, execute assertions.
- Assertions provide additional tests.
TDD: pros and cons

Pros

- Reduce gap between decision and feedback.
- Encourage developers to write code that is easily tested.
- Creates a thorough test bed.

Drawbacks

- Time taken away from core development.
- Some code is difficult to test.
GUI Testing
Why is GUI testing hard?

- **GUI**
  - Graphics: easy for humans, hard for machines
  - Themable GUIs
  - Simple change to interface, big impact

- **Network & Databases**
  - Big effort to set up environment
    - Computers
    - Operating Systems
    - Applications
    - Data
    - Network
  - Reproducibility
Why is GUI testing hard?

- In the old days things were easy
  - Stdin / Stdout / Stderr
- Modern applications lack uniform interface
  - GUI
  - Network
  - Database
  - ...
Minimizing GUI code

- GUI code is hard to test
- Try to keep it minimal
- How?
Model-View-Controller

Views

Model

A = 50%
B = 30%
C = 20%
Model-View Controller

MODEL

MODEL represents VIEWi

VIEWi updates CONTROLLER

CONTROLLER interacts with GUI tools

User thinks in terms of MODEL

MVC structure
Model View Controller (2/2)

Model
- Encapsulates application state
- Exposes application functionality
- Notifies view of changes

View
- Renders the model
- Sends user gestures to controller
- Allows controller to select view

Controller
- Defines application behavior
- Maps user actions to model updates
- Selects view for response
- One for each functionality

Change Notification

State change

View selection

User gestures

Feature calls

Events
Example: Abstracting the GUI away

- **Algorithm needs to save file**
- **Algorithm queries Dialog for name**
- **Makes Algorithm hard to test**

**Solution:**
- Abstract interactivity away
- *Makes more of your software easy to test*
Capture and replay

Capture
Run GUI application manually, capture all the input events such as keystrokes, mouse moves and clicks.

Replay
Rerun the application automatically, spawn recorded events, check if the system responses as expected.

Problems
Fragile to changes, hard to define correctness.
Test management
Testing strategy

Planning & structuring the testing of a large program:

- Defining the process
  - Test plan
  - Input and output documents
- Who is testing?
  - Developers / special testing teams / customer
- What test levels do we need?
  - Unit, integration, system, acceptance, regression
- Order of tests
  - Top-down, bottom-up, combination
- Running the tests
  - Manually
  - Use of tools
  - Automatically
Who tests

Any significant project should have a separate QA team

Why: the almost infinite human propensity to self-delusion

Unit tests: the developers

- My suggestion: pair each developer with another who serves as “personal tester”

Integration test: developer or QA team
System test: QA team
Acceptance test: customer & QA team
Classifying reports: by severity

Classification must be defined in advance
Applied, in test assessment, to every reported failure
Analyzes each failure to determine whether it reflects a fault, and if so, how damaging

Example classification (from a real project):
- Not a fault
- Minor
- Serious
- Blocking
Classifying reports: by status

From a real project:

- Registered
- Open
- Re-opened
- Corrected
- Integrated
- Delivered
- Closed
- Irreproducible
- Cancelled
Assessment process (from real project)

- Registered
  - Open
    - Corrected
    - Integrated
  - Closed
- Reopened
- Cancelled
  - Irreproducible
Some responsibilities to be defined

Who runs each kind of test?

Who is responsible for assigning severity and status?

What is the procedure for disputing such an assignment?

What are the consequences on the project of a failure at each severity level?

(e.g. “the product shall be accepted when two successive rounds of testing, at least one week apart, have evidenced fewer than \textit{m} serious faults and no \textit{blocking} faults”).
Debugging
Debugging: topics and scope

What is Debugging?
Problem Management
How Failures Come to Be?
Scientific Debugging Techniques
  ➢ Delta Debugging
What is Debugging?
What Is Debugging?

Debugging is the work required to diagnose and correct a bug.

Testing is not debugging.

Debugging is not testing.

Debugging typically occurs after a failure has been observed.
Tracking problems

Large projects have many bugs reported.
Bugs are not always fixed immediately.

Need for Bug tracking system

- Bugzilla
- Origo
Classifying Problems

Severity
- Blocker
- Critical
- Major
- Normal
- Minor
- Trivial
- Enhancement

Priority
Identifier
Comments
Notifications
Testing and bug prevention

Three questions about each bug you find (Van Vleck):

- “Is this mistake somewhere else also?”
- “What next bug is hidden behind this one?”
- “What should I do to prevent bugs like this?”
How Failures Come to Be 1/3

variables

OK

time

?
How Failures Come to Be 2/3

variables

time

OK

X

X
How Failures Come to Be 3/3

variables

time
Observe failure.
Invent hypothesis, consistent with observation.
Use hypothesis to make prediction.
Test prediction by experiment or observation:

- If prediction satisfied, then refine hypothesis.
- Otherwise, create alternative hypothesis.
Debugging Techniques

Delta Debugging
Bug Example: Mozilla

<table>
<thead>
<tr>
<th>op sys</th>
<th>priority</th>
<th>bug severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>--</td>
<td>blocker</td>
</tr>
<tr>
<td>Windows 3.1</td>
<td>P1</td>
<td>critical</td>
</tr>
<tr>
<td>Windows 95</td>
<td>P2</td>
<td>major</td>
</tr>
<tr>
<td>Windows 98</td>
<td>P3</td>
<td>normal</td>
</tr>
<tr>
<td>Windows ME</td>
<td>P4</td>
<td>minor</td>
</tr>
<tr>
<td>Windows 2000</td>
<td>P5</td>
<td>trivial</td>
</tr>
<tr>
<td>Windows NT</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7.5</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7.6</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 8.0</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 9.x</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>MacOS X</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Linux</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>BSDI</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>NetBSD</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>AIX</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>BeOS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>MacOS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Solaris</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>SunOS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>openVMS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>OS/2</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>OSF/1</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>Solaris</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>SunOS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>openVMS</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>OS/2</td>
<td>other</td>
<td>enhancement</td>
</tr>
<tr>
<td>OSF/1</td>
<td>other</td>
<td>enhancement</td>
</tr>
</tbody>
</table>
Bug Example: Mozilla

Looking at the input it is hard to understand the real cause of the bug.

Can we simplify the input?
Delta Debugging: Characteristics

Simplification algorithm for bug reproducing examples.
Reduces size of input or program.
Easy to implement and customize.

Assumptions

- Input can be split into parts
- Working program
- Failing program
Delta Debugging: Example 2/5

Assume the following makes Mozilla crash:

```html
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Approach:

- Remove parts of input and see if it still crashes.
## Delta Debugging: Example 3/5

1. `<SELECT NAME="priority" MULTIPLE SIZE=7> F
2. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
3. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
4. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
5. `<SELECT NAME="priority" MULTIPLE SIZE=7> F
6. `<SELECT NAME="priority" MULTIPLE SIZE=7> F
7. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
8. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
9. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
10. `<SELECT NAME="priority" MULTIPLE SIZE=7> F
11. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
12. `<SELECT NAME="priority" MULTIPLE SIZE=7> P
13. `<SELECT NAME="priority" MULTIPLE SIZE=7> P

Bold parts remain in the input

- Fail
- Pass
Delta Debugging: Example 4/5

14  <SELECT NAME="priority" MULTIPLE SIZE=7> P
15  <SELECT NAME="priority" MULTIPLE SIZE=7> P
16  <SELECT NAME="priority" MULTIPLE SIZE=7> F
17  <SELECT NAME="priority" MULTIPLE SIZE=7> F
18  <SELECT NAME="priority" MULTIPLE SIZE=7> F
19  <SELECT NAME="priority" MULTIPLE SIZE=7> P
20  <SELECT NAME="priority" MULTIPLE SIZE=7> P
21  <SELECT NAME="priority" MULTIPLE SIZE=7> P
22  <SELECT NAME="priority" MULTIPLE SIZE=7> P
23  <SELECT NAME="priority" MULTIPLE SIZE=7> P
24  <SELECT NAME="priority" MULTIPLE SIZE=7> P
25  <SELECT NAME="priority" MULTIPLE SIZE=7> P
26  <SELECT NAME="priority" MULTIPLE SIZE=7> F
After 26 tries we found:

\[
\text{<SELECT>}
\]

causes Mozilla to crash.

Delta Debugging completely automates this process
Can this be applied only to input?
Delta Debugging: Limitations

Delta Debugging does not guarantee smallest possible example.

- It only guarantees an example where every line is relevant.

We need to be able to replay inputs.
We need to be able to split inputs.
Empty input must not trigger failure.
Debugging: conclusion

Debugging
Failures
Problem Management
Scientific Debugging
Techniques
  ➢ Delta Debugging