Software Engineering

Prof. Dr. Bertrand Meyer
Dr. Manuel Oriol
Dr. Bernd Schoeller

Introduction to QA and testing

(includes material adapted from Prof. Peter Müller)
Topics

Part 1: QA basics
Part 2: Testing basics & terminology
Part 3: Testing strategies
Part 4: Test automation
Part 5: Measuring test quality
Part 6: GUI testing
Part 7: Test management
Part 1:

QA 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 (reminder)

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**

Example: A Y2K issue

**Failure**: person’s age appears as negative!
**Fault**: code for computing age yields negative value if birthdate is in 20\textsuperscript{th} century and current date is in 21\textsuperscript{st}
**Mistake**: failed to account for dates beyond 20\textsuperscript{th} 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
What does QA target?

Everything!

Process:
- Timeliness
- Cost
- Goal achievement
- Self-improvement
- ...

Product:
- Correctness
- Robustness
- Efficiency (performance)
- ...

Software Engineering, lecture 9: Introduction to Testing
In this presentation...

... we concentrate on QA of product properties.

Mostly functional properties (correctness, robustness), but also some non-functional aspects.
When should QA be performed?

All the time!

A priori — build it right:
- Process (e.g. CMMI, PSP, Agile)
- Methodology (e.g. requirements, formal methods, Design by Contract, patterns…)
- Tools, languages

A posteriori — verify:
- Tests
- Other static and dynamic techniques (see next)
When should QA be performed?

All the time!

A priori — build it right:
- Process (e.g., CMMI, PSP, Agile)
- Methodology (e.g., requirements, formal methods, Design by Contract, patterns…)
- Tools, languages

A posteriori — verify:
- Tests
- Other static and dynamic techniques (see next)

Reagan to Gorbachev (1987): “My favorite Russian proverb: Trust but verify” (Доверай, но проверяй)

Gorbachev to Reagan: “You repeat this every time we meet!”
Levels

Fault avoidance

Fault detection (verification)

Fault tolerance
In this presentation...

... we concentrate on a posteriori (verification) techniques.
How should a posteriori verification be performed?

In many ways!

**Static (no execution):**
- Reviews (human)
- Type checking & enforcement of other reliability-friendly programming language traits
- Static analysis
- Proofs

**In-between but mostly static:**
- Model checking
- Abstract interpretation
- Symbolic execution

**Dynamic (must execute):**
- Tests
In this presentation...

... we concentrate on testing:

- Product (rather than process)
- A posteriori (rather than a priori)
- Dynamic (rather than static)

Later lectures will present static analysis, proofs (a glimpse) and model checking.
The obligatory quote

“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)
Definition: testing

To test a software system is to try to make it fail

Testing is none of:

- Ensuring software quality
- Assessing software quality
- Debugging

“Ich bin der Geist, der stets verneint”

Goethe, Faust, Act I

Fiodor Chaliapine as Mephistopheles
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)
V-shaped variant of the Waterfall

- FEASIBILITY STUDY
- REQUIREMENTS ANALYSIS
- GLOBAL DESIGN
- DETAILED DESIGN
- IMPLEMENTATION
- DISTRIBUTION
- SYSTEM VALIDATION
- SUBSYSTEM VALIDATION
- UNIT VALIDATION
- SUBSYSTEM VALIDATION
Part 2:

Testing basics & terminology
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
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”.

More ingredients: test execution

Test driver
A program, or program element (e.g. class), used to apply test cases to an IUT

Stub
A temporary implementation of a software element, replacing its actual implementation during testing of other elements relying on it.
Generally doesn't satisfy the element's full specification.
May serve as placeholder for:
- A software element that has not yet been written
- External software that cannot be run for the test (e.g. because it requires access to hardware or a live database)
- A software element that takes too much time or memory to run, and whose results can be simulated for testing purposes

Test harness
A setup, including test drivers and other necessary elements, permitting test execution
Test classification: by goal

- Functional test
- Performance test
- Stress (or “load”) test
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
Classification: by intent

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
Classification: by process phase

Unit testing: implementation
Integration testing: subsystem integration
System testing: system integration
Acceptance testing: deployment
Regression testing: maintenance
Classification: by available information

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><strong>White-box</strong></th>
<th><strong>Black-box</strong></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>
Part 3:

Testing strategies
Partition testing (black-box)

We cannot test all inputs, but need realistic inputs

Idea of partition testing: select elements from a partition of the input set, i.e. a set of subsets that is
- **Complete**: union of subsets covers entire domain
- **Pairwise disjoint**: no two subsets intersect

Purpose (or hope!):
- For any input value that produces a failure, some other in the same subset produces a similar failure

Common abuse of language: “a partition” for “one of the subsets in the partition” (e.g. $A_2$)
- Better called “equivalence class”
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"
Choosing values from equivalence classes

Each Choice (EC):
- For every equivalence class \( c \), at least one test case must use a value from \( c \)

All Combinations (AC):
- For every combination \( ec \) of equivalence classes, at least one test case must use a set of values from \( ec \)
- Obviously more extensive, but may be unrealistic
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) adds a 29th day to [February](https://en.wikipedia.org/wiki/February) in all years evenly divisible by four, except centennial years (those ending in -00), which only get it if they are evenly divisible by 400.

Thus 1600, 2000 and 2400 are leap years but not 1700, 1800, 1900.

*Slightly abridged*
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)
- Non-leap years commonly mistaken as non-leap (2000)
- Invalid months: 0, 13
- For numbers in general: 0, very large, very small
- Maximum positive integer, minimum negative integer
- Smallest representable floating-point number
- For interval types: middle and ends of interval
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 not by many testers, but lacks rigorous basis for assessing effectiveness
Coverage (white-box)

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

Details in part 5 (assessing test quality)
Part 4:

Test automation
Test automation

Testing is difficult and time consuming

So why not do it automatically?

What is most commonly meant by “automated testing” currently is automatic test execution

But actually...
What can we automate?

**Generation**
- Test inputs (values & objects used as targets & arguments of calls)
- Selection of test data
- Test driver code

**Execution**
- Running the test code
- Recovering from failures

**Evaluation**
- Oracle: classify pass/no pass
- Other info about results

**Test quality estimation**
- Coverage measures
- Other test quality measures
- Feedback to test data generator

**Management**
- Adaptation to user’s process, preferences
- Save tests for regression testing
Automated today (xunit)

**Generation**
- Test inputs (values & objects used as targets & arguments of calls)
- Selection of test data
- Test driver code

**Execution**
- Running the test code
- Recovering from failures

**Evaluation**
- Oracle: classify pass/no pass
- Other info about results

**Test quality estimation**
- Coverage measures
- Other test quality measures
- Feedback to test data generator

**Management**
- Adaptation to user’s process, preferences
- Save tests for regression testing
The trickiest parts to automate

**Generation**
- Test inputs (values & objects used as targets & arguments of calls)
- Selection of test data
- Test driver code

**Execution**
- Running the test code
- Recovering from failures

**Evaluation**
- Oracle: classify pass/no pass
- Other info about results

**Test quality estimation**
- Coverage measures
- Other test quality measures
- Feedback to test data generator

**Management**
- Adaptation to user’s process, preferences
- Save tests for regression testing
The generic name for a number of current test automation frameworks for unit testing

Goal: to provides all needed mechanisms to run tests, so test writer must only provide test-specific logic

Implemented in all the major programming languages:

- JUnit – for Java
- cppunit – for C++
- SUnit – for Smalltalk (the first one)
- PyUnit – for Python
- vbUnit – for Visual Basic
- EiffelTest – for Eiffel
Hands-on!

Unit Testing:

A session with JUnit
Hands-on with JUnit: resources

Unit testing framework for Java
  Erich Gamma & Kent Beck
Open source (CPL 1.0), hosted on SourceForge
Current version: 4.3
Available at: www.junit.org

Intro to JUnit 3.8: Erich Gamma, Kent Beck, JUnit Test

JUnit 4.0: Erich Gamma, Kent Beck, JUnit Cookbook,
  http://junit.sourceforge.net/doc/cookbook/cookbook.htm
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
Example: basics

```java
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 {
    @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());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTest.class);
    }
}
```

To declare a routine as a test case

To compare the actual result to the expected one

Required to run JUnit4 tests with the old JUnit runner
Example: set up and tear down

```java
package unitests;

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());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTestWithSetUpTearDown.class);
    }
}
```

- To run this routine before any `@Test` routine
- To run this routine after any `@Test` routine
- Must make account an attribute of the class now
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.

```
@Test(timeout=1000) public void testTimeout () {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.infiniteLoop();
}
```
Automated today (xunit)

**Generation**
- Test inputs (values & objects used as targets & arguments of calls)
- Selection of test data
- Test driver code

**Execution**
- Running the test code
- Recovering from failures

**Evaluation**
- Oracle: classify pass/no pass
- Other info about results

**Test quality estimation**
- Coverage measures
- Other test quality measures
- Feedback to test data generator

**Management**
- Adaptation to user’s process, preferences
- Save tests for regression testing
The trickiest parts to automate

**Generation**
- Test inputs (values & objects used as targets & arguments of calls)
- Selection of test data
- Test driver code

**Execution**
- Running the test code
- Recovering from failures

**Evaluation**
- Oracle: classify pass/no pass
- Other info about results

**Test quality estimation**
- Coverage measures
- Other test quality measures
- Feedback to test data generator

**Management**
- Adaptation to user’s process, preferences
- Save tests for regression testing
Push-button testing: AutoTest

Goal: never write a test case, a test suite, a test oracle, or a test driver

IUTs: contracted classes, written in Eiffel

Automatically generate

- Objects
- Feature calls
- Evaluation and saving of results

User only specifies which classes to test; the tool does the rest: test generation, execution and result evaluation
Master/Slave Design

Separation of
- **Driver (Master)**
- **Interpreter (Slave)**

Robust testing
Keep objects around
Dynamic test case creation & execution
AutoTest as a framework

[Diagram showing the components and flow of AutoTest as a framework, including System under test, Master, strategy, proxy, oracle, results, and Interpreter.]

Software Engineering, lecture 9: Introduction to Testing  61
AutoTest principles

- Input is set of classes, and testing time
- AutoTest generates instances and calls features with automatically selected arguments
- Oracles are contracts:
  - Precondition violations: skip
  - Postcondition/invariant violation: bingo!
- Manual tests can be added explicitly
- Any test (manual or automated) that fails becomes part of the test suite
Automated testing and slicing

\[\texttt{auto\_test sys.ace -t 120 BANK\_ACCOUNT}\]

**STRING**

create \{STRING\} v1
v1.wipe\_out
v1.append\_character (\'c\')
v1.append\_double (2.45)
create \{STRING\} v2
v1.append\_string (v2)
v2.fill (\'g\', 254343)
...
create \{BANK\_ACCOUNT\} v3.make (v2)
v3.deposit (15)
v3.deposit (100)
v3.deposit (-8901)
...

```plaintext
class BANK\_ACCOUNT
create
make feature
make (n : STRING)
require
\ n /= Void
do
name := n
balance := 0
ensure
name = n
end
invariant
name /= Void
balance >= 0
end
```

```plaintext
name : STRING
balance : INTEGER
deposit (v : INTEGER)
do
\ balance ::= balance + v
ensure
balance =
\ old balance + v
end
```

Software Engineering, lecture 9: Introduction to Testing
## Some results (random strategy)

<table>
<thead>
<tr>
<th>Library</th>
<th>TESTS</th>
<th></th>
<th></th>
<th>ROUTINES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Failed</td>
<td>Total</td>
<td>Failed</td>
<td></td>
</tr>
<tr>
<td>EiffelBase</td>
<td>40,000</td>
<td>3%</td>
<td>2000</td>
<td>6%</td>
<td></td>
</tr>
<tr>
<td>(Sep 2005)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gobo Math</td>
<td>1500</td>
<td>1%</td>
<td>140</td>
<td>6%</td>
<td></td>
</tr>
</tbody>
</table>
Handson!

Push-button testing:

A session with AutoTest
Part 5:

Measuring 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.
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)
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
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)
Example: source code

```java
class ACCOUNT feature
    balance : INTEGER

withdraw (sum : INTEGER)
    do
        if balance >= sum then
            balance := balance - sum
            if balance = 0 then
                io.put_string("Account empty%N")
            end
        else
            io.put_string("Less than ")
            io.put_integer(sum)
            io.put_string(" CHF in account%N")
        end
    end
```

Start

```
balance >= sum
```

| True | balance = balance - sum |
| False | print(...) |

balance = 0

| True | print(...) |
```

Software Engineering, lecture 9: Introduction to Testing
Instruction coverage

class ACCOUNT feature

    balance : INTEGER

withdraw (sum : INTEGER)
do
    if balance >= sum then
        balance := balance - sum
    if balance = 0 then
        io.put_string ("Account empty\n")
    end
else
    io.put_string ("Less than ")
    io.put_integer (sum )
    io.put_string (" CHF in account\n")
end
end

-- TC1:  create a  a.set_balance(100)  a.withdraw(1000)
-- TC2:  create a  a.set_balance(100)  a.withdraw(100)
class ACCOUNT feature

balance : INTEGER

withdraw (sum : INTEGER)
  do
    if balance >= sum then
      balance := balance - sum
    if balance = 0 then
      io.put_string ("Account empty\n")
    end
  else
    io.put_string ("Less than ")
    io.put_integer (sum )
    io.put_string (" CHF in account\n")
  end
end

-- TC1: create a
-- TC2: create a
-- TC3: create a
create a
a.set_balance (100)
a.withdraw (1000)

create a
a.set_balance (100)
a.withdraw (100)

create a
a.set_balance (100)
a.withdraw (99)
Code coverage tools

Emma
- Java
- Open-source
- http://emma.sourceforge.net/

JCoverage
- Java
- Commercial tool
- http://www.jcoverage.com/

NCover
- C#
- Open-source
- http://ncover.sourceforge.net/

Clover, Clover.NET
- Java, C#
- Commercial tools
- http://www.cenqua.com/clover/
Dataflow-oriented testing

Focuses on how variables are defined, modified, and accessed throughout the run of the program

Goal: to execute certain paths between a definition of a variable in the code and certain uses of that variable
Access-related bugs

- Using an uninitialized variable
- Assigning to a variable more than once without an intermediate access
- Deallocating a variable before it is initialized
- Deallocating a variable before it is used
- Modifying an object more than once without accessing it
Types of access to variables

Definition (\texttt{def}): changing the value of a variable
   Creation instruction, assignment

Use: reading the value of a variable without changing it
   - Computational use (c-use): use variable for computation
   - Predicative use (p-use): use in a test

Kill: any operation that causes the value to be deallocated, undefined, no longer usable

Examples:
   - \texttt{a := b * c} \quad \text{c-use of } b \ ; \ c-use of \ c \ ; \ \text{def of } a
   - \texttt{if } x > 0 \ \text{then...} \quad \text{p-use of } x
Data flow graph

Measures of dataflow coverage can be defined in terms of the data flow graph

A sub-path is a sequence of consecutive nodes on a path
Characterizing paths in a dataflow graph

For a path or sub-path \( p \) and a variable \( v \):

**Def-clear** for \( v \):
- No definition of \( v \) occurs in \( p \)

**Du-path** for \( v \):
- \( p \) starts with a definition of \( v \)
- Except for this first node, \( p \) is def-clear for \( v \)
- \( v \) encounters either a c-use in the last node or a p-use along the last edge of \( p \)
Example: control flow graph for **withdraw**

```plaintext
class ACCOUNT feature
  balance : INTEGER
  withdraw (sum : INTEGER)
    do
      if balance >= sum then
        balance := balance - sum
        if balance = 0 then
          io.put_string ("Account empty\%N")
        end
      else
        io.put_string ("Less than ")
        io.put_integer (sum )
        io.put_string (" CHF in account\%N")
      end
    end
end
```

Definition of sum
- True
  - balance>= sum
  - balance:= balance - sum
- False
  - print(sum)
  - print(sum)
  - False
Data flow graph for **sum in withdraw**

**Definition of sum**

(0) 

if 

balance >= sum 

(1) 

if 

balance := balance – sum 

(2) 

True 

False 

if 

balance = 0 

(3) 

if 

true 

True 

False 

print(sum) 

(4) 

print(sum) 

(5) 

def 

(0) 

def 

p-use 

(1) 

def 

c-use 

(2) 

c-use 

(3) 

c-use 

(4) 

c-use 

(5) 

Software Engineering, lecture 9: Introduction to Testing  81
Data flow graph for **balance** in **withdraw**

1. **Definition of sum**
   - (0)

2. **if** balance >= sum
   - (1)
   - True: balance := balance – sum
     - (2)
   - False: print(sum)
     - (4)

3. **if** balance = 0
   - (3)
   - True: print(sum)
     - (5)
   - False:

4. **p-use**
   - (5)
   - True:
   - False:
Dataflow coverage criteria

**all-defs**: execute **at least one** def-clear sub-path between every definition of every variable and **at least one** reachable use of that variable.

**all-p-uses**: execute **at least one** def-clear sub-path from every definition of every variable to **every** reachable p-use of that variable.

**all-c-uses**: execute **at least one** def-clear sub-path from every definition of every variable to **every** reachable c-use of the respective variable.
Dataflow coverage criteria (continued)

all-c-uses/some-p-uses: apply all-c-uses; then if any definition of a variable is not covered, use p-use

all-p-uses/some-c-uses: symmetrical to all-c-uses/some-p-uses

all-uses: execute at least one def-clear sub-path from every definition of every variable to every reachable use of that variable
Dataflow coverage criteria for \textit{sum}

\begin{itemize}
  \item \textbf{all-defs:} at least one def-clear sub-path between every definition and at least one reachable use \((0,1)\)
  \item \textbf{all-p-uses:} at least one def-clear sub-path from every definition to every reachable p-use \((0,1)\)
  \item \textbf{all-c-uses:} at least one def-clear sub-path from every definition to every reachable c-use \((0,1,2); (0,1,2,3,4); (0,1,5)\)
\end{itemize}
Dataflow coverage criteria for \textit{sum}

- **all-c-uses/some-p-uses**: apply all-c-uses; then if any definition of a variable is not covered, use p-use $(0,1,2); (0,1,2,3,4); (0,1,5)$

- **all-p-uses/some-c-uses**: symmetrical to all-c-uses/some-p-uses $(0,1)$

- **all-uses**: at least one def-clear sub-path from every definition to every reachable use $(0,1); (0,1,2); (0,1,2,3,4); (0,1,5)$
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.: \( x > 0 \)

Notation:
  ➢ \( P = \) set of predicates
  ➢ \( C_p = \) set of clauses of predicate \( p \)

If specification expressed as predicates on the state, specification coverage translates to predicate coverage.
Predicate coverage (PC)

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 (fault injection)

How do you count the Egli in the Zürichsee?
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
Who tests the tester?

- Program: tested by test suite
- Test suite: tested by ?

- Good test suite: finds failures
- Problem: if program perfect, no good test case
- Solution: introduce bugs in program, then test
  - If bugs are found, test suite good
  - If no bugs are found, test suite bad
Fault injection terminology

Faulty versions of the program = mutants

- We only consider mutants that are not equivalent to the original program

A mutant is

- Killed if at least one test case detects the fault injected into the mutant
- Alive otherwise

A mutation score (MS) is associated to the test set to measure its effectiveness
Mutation operators

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

A mutant is 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:

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

Mutants:

```
if (a < b)
if (a <= b)
if (a > b)
if (c < b)
    b := b - a;
else
    b := b + a;
    b := x - a;
else
    b := 0;
    b := 1;
    a := 0;
```
Mutation 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.twin`
System test quality (STQ)

$S$ - system composed of $n$ components denoted $C_i$

$d_i$ - number of killed mutants after applying the unit test sequence to $C_i$

$m_i$ - total number of mutants

The mutation score $MS$ for $C_i$ being given a unit test sequence $T_i$:

$$MS (C_i, T_i) = \frac{d_i}{m_i}$$

$$\sum_{i=1}^{n} \frac{d_i}{m_i}$$

$$STQ (S) = \sum_{i=1}^{n} m_i$$

In general, STQ is a measure of test suite quality.

If contracts are used as oracles, STQ is a combined measure of test suite quality and contract quality.
Mutation tools

muJava - [http://ise.gmu.edu/~ofut/mujaava/](http://ise.gmu.edu/~ofut/mujaava/)
Part 6:
GUI Testing
# Console vs. GUI Applications

<table>
<thead>
<tr>
<th></th>
<th>Human</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Console Application</strong></td>
<td>Hard to use</td>
<td>Hard to process</td>
</tr>
<tr>
<td><strong>GUI Application</strong></td>
<td>Easy to use</td>
<td>Easy to process</td>
</tr>
</tbody>
</table>
Why is GUI testing hard?

- **GUI**
  - Bitmaps
  - Themable GUIs
  - Simple change to interface, big impact
  - Platform details, e.g. resolution

- **Network & Databases**
  - Complicated set up
    - Computers
    - Operating Systems
    - Applications
    - Data
    - Network
  - Reproducibility
Why is GUI testing hard?

- In the CLI 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?
  - class LIST_VIEW
  - class SORTED_LIST_VIEW
Model-View-Controller

**Views**

- [Table with values: A=50%, B=30%, C=20%]
- [Bar chart: A=50%, B=30%, C=20%]
- [Pie chart: A=50%, B=30%, C=20%]

**Model**

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

**Events**
- Feature calls
- User gestures
- Change Notification
- State change
- View selection
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 / Replay: principle

- **Phase 1: Capture**
  - Run application, record inputs and outputs

- **Phase 2:**
  - Replay recorded inputs to application
  - Compare new outputs to recorded outputs

- **Potential issues: Performance**
Capture / Replay: operating system approach

- **Capture at OS level**
  - Must change OS
    - Per interface
  - Works for all applications
  - Depends on operating system
  - Fragile wrt theme changes
Capture at library level

- Must change each library
- Must not talk to system directly
- Works for all operating systems
Capture / Replay: language approach

- **Capture at the language level**
  - **Must change compiler or VM**
  - **Works on all operating systems**
  - **Works on all interfaces**
  - **Easy to change what is captured**

- But, capturing everything is too costly...
Hands-on!

GUI capture/replay:
The Scarpe example
Scarpe: A capture/replay tool

Capture: Input → Rest of the Application → Captured Subsystem → Libraries → Event Log

Replay: Event Log → Output

Joshi, Orso 2006
Scarpe: A capture/replay tool
Scarpe: events

- **Routines**
  - Out-call / Out-call-return
  - In-call / In-call-return
- **Fields**
  - Out-read
  - Out-write
  - In-Write
- **Constructors**
  - ...
- **Exceptions**
  - ...
Scarpe: capture phase

Joshi, Orso 2006
Scarpe: replay phase

- Replays are sandbox automatically

Joshi, Orso 2006

Software Engineering, lecture 9: Introduction to Testing 120
Scarpe: typical use case

- Developer selects boundary for recording
- Application at client side records by default
- In case of failure
  - Minimize failure at client side
  - Send it to developer
GUI testing: conclusions

- Write testable code
  - Minimize GUI code
  - Separate GUI code from non-GUI code
  - MVC pattern
- Capture / Replay
  - Operating System level
  - Library level
  - Programming language level
Part 7:

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
- Not retained
- Irreproducible
- Cancelled
Assessment process (from real project)

```
Customer
Registered

Developer
Open
Corrected
Integration

Project/Customer

Customer
Closed

Project
Cancelled
Irreproducible

Project
Reopened

Developer

Project
```
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 $m$ serious faults and no blocking faults”).
Test planning: IEEE 829


Can be found at: [http://tinyurl.com/35pcp6](http://tinyurl.com/35pcp6)

Specifies a set of test documents and their form

For an overview, see the Wikipedia entry
IEEE-829-conformant test elements

Test plan:
- "Prescribes scope, approach, resources, & schedule of testing. Identifies items & features to test, tasks to perform, personnel responsible for each task, and risks associated with plan"*

Test specification documents:
- Test design specification: identifies features to be covered by tests, constraints on test process
- Test case specification: describes the test suite
- Test procedure specification: defines testing steps

Test reporting documents:
- Test item transmittal report
- Test log
- Test incident report
- Test summary report

*Citation slightly abridged
IEEE 829: Test plan structure

a) Test plan identifier
b) Introduction
c) Test items
d) Features to be tested
e) Features not to be tested
f) Approach
g) Item pass/fail criteria
h) Suspension criteria and resumption requirements
i) Test deliverables
j) Testing tasks
k) Environmental needs
l) Responsibilities
m) Staffing and training needs
n) Schedule
o) Risks and contingencies
p) Approvals
Test Case Specification: an example

Part 1: Identification

S01. Name
S02. Code
S03. Source of test: one of
  □ Devised by tester in QA process
  □ EiffelWeasel
  □ Internal bug report
  □ User bug report
  □ Automatic, e.g. AutoTest
S04. Original author, date
  ______________________
S05. Revisions (author, date)
  ______________________
S07. Product or products affected
S08. Purpose

S06. Other references (zero or more)
  □ Bug database entry: _______
  □ Email message
    from __ to ___, date: ___
  □ Minutes of meeting: reference
    _______
  □ ISO/ECMA 367: section, page:
    _______
  □ URL: ____________________
  □ Other document: _________
    Section, page: ____
  □ Other:___________________
Test case specification: an example

Part 2: Details

S09. Nature: one of
- Functional correctness
- Performance: time
- Performance: memory
- Performance, other: ________
- Usability

S10. Context: one of
- Normal usage
- Stress/boundary
- Platform compatibility with ___

S11. Severity if test fails
- Minor, doesn't prevent release
- Serious, requires management decision to approve release
- Blocking, prevents release

S12. Relations to other tests

S13. Scope: one of
- Feature: ____ (fill "class")
- Class: ______ (fill "cluster")
- Cluster/subsystem: ______
- Collaboration test
  - Other elements involved:
    - ____________________________
      - System test
      - Eiffel language mechanism
        - ____________________________
        - Other language mechanism

S14. Release where it must succeed

S15. Platform requirements

S16. Initial conditions

S17. Expected results

S18. Any test scripts used
Test case specification: an example

Part 3: Test execution

S19. Test procedure (how to run the test)

S20. Status of last test run: one of

- Passed
- Failed
  Test Run Report id: ___________________

S21. Regression status: one of

- Some past test runs have failed
- Some past test runs have passed
### Test Run Report: an example

<table>
<thead>
<tr>
<th>R01. TCS id (refers to S02)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R02. Test run id (unique, automatically generated)</td>
</tr>
<tr>
<td>R03. Date and time run</td>
</tr>
<tr>
<td>R04. Precise identification</td>
</tr>
<tr>
<td>Platform ___________________</td>
</tr>
<tr>
<td>Software versions involved</td>
</tr>
<tr>
<td>(SUT plus any others needed):</td>
</tr>
<tr>
<td>___________________________</td>
</tr>
<tr>
<td>Other info on test run:</td>
</tr>
<tr>
<td>___________________________</td>
</tr>
<tr>
<td>R04. Name of tester</td>
</tr>
<tr>
<td>R05. Testing tool used</td>
</tr>
<tr>
<td>R05. Result as assessed by tester:</td>
</tr>
<tr>
<td>☐ Pass</td>
</tr>
<tr>
<td>☐ Fail</td>
</tr>
<tr>
<td>R06. More detailed description of test run if necessary and any other relevant details describing test run</td>
</tr>
<tr>
<td>R07. Other test run data, e.g. performance figures (time, memory)</td>
</tr>
<tr>
<td>R07. Caused update of TCS?</td>
</tr>
<tr>
<td>☐ Yes -- what was changed?</td>
</tr>
<tr>
<td>☐ No</td>
</tr>
</tbody>
</table>

Software Engineering, lecture 9: Introduction to Testing  136
When to stop testing?

You don’t know, but in practice:

- Keep a precise log of bugs and bug numbers
- Compare to previous projects
- Extrapolate

See Belady and Lehmann work on OS 360 releases