Part I
Introduction, Testing Process

Part II
Unit Testing:

A Software Life-cycle Model

Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation of Units (classes, procedures, functions)

Unit testing

Acceptance Test
(Release testing)

System Testing
(Integration testing of modules)

Module Testing
(Integration testing of units)

Validate Requirements, Verify Specification

Verify System Design

Verify Module Design

Verify Implementation

Acceptance Test

System Testing

Module Testing

Maintenance

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education
Part I
Introduction, Testing Process

Part II
Unit Testing:
Part I
Introduction, Testing Process
Testing a ballpoint pen

 Does the pen write in the right color, with the right line thickness?
 Is the logo on the pen according to company standards?
 Is it safe to chew on the pen?
 Does the click-mechanism still work after 100 000 clicks?
 Does it still write after a car has run over it?

What is expected from this pen?

Intended use!!
Testing software

Part I
Introduction, Testing Process

Part II
Unit Testing:

- Are the functions giving correct output?
  - Are the integrated modules giving correct output?
  - Is the entire system giving correct output when used?
- Is the correct output given in reasonable time?
- Is the output presented in an understandable way?
- Was this what we really expected?

- Software testing is an activity in which a program is executed under specified conditions, the results are observed, and an evaluation is made of the program.
Other methods for Validation & Verification

- Formal verification – read about Z in the book
- Model checking
- Prototyping
- Simulation
- Software reviews – lecture to come
"Testing shows the presence, not the absence of bugs" 
(Edsger Wybe Dijkstra)

...but you might use experience and statistics to make some kind of assessment.
The terminology here is taken from standards developed by the institute of Electronics and Electrical Engineers (IEEE) computer Society.

- **Error**: people make **errors**. A good synonym is **mistake**. When people make mistakes while coding, we call these mistakes **bugs**. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

- **Fault**: a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. **Defect** is a good synonym for fault, as is **bug**. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. We might speak of faults of commission and faults of omission. A **fault of commission** occurs when we enter something into a representation that is incorrect. **Faults of omission** occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

- **Failure**: a failure occurs when a **fault executes**. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission?
Part I
Introduction, Testing Process

Part II
Unit Testing:

Error, Fault, Failure

Human error (Mistake, Bug)

Can lead to

Fault (Defect, Bug)

Can lead to

Failure

Kristian.Sandahl@liu.se
A Testing Life Cycle

Part I
Introduction,
Testing Process

Part II
Unit Testing:

- Requirements Specification
- Design
- Coding
- Testing
- Fault Resolution
- Fault Isolation
- Fault Classification

Putting Bugs IN
Development phases

Finding Bugs
Testing phase

Getting Bugs OUT

Error
Fault
Error
Fault
Error
Fault
Incident
Fix
Error
Program Behaviors

Part I
Introduction, Testing Process

Part II
Unit Testing:

Specification (expected)

Program (observed)

Missing Functionality (sins of omission)

Extra Functionality (sins of commission)

"Correct" Portion
Basic Approaches

Part I
Introduction,
Testing Process

Part II
Unit Testing:

Specification

Program

Functional (Black Box) establishes confidence

Structural (White Box) seeks faults
Types of Faults

Discuss in pairs:

What types of faults do you typically find in your software?

How can you test to find them?
Types of Faults
(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req.
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other.
Faults classified by severity
(Beizer, 1984)

1. Mild
   Misspelled word

2. Moderate
   Misleading or redundant information

3. Annoying
   Truncated names, bill for $0.00

4. Disturbing
   Some transaction(s) not processed

5. Serious
   Lose a transaction

6. Very serious
   Incorrect transaction execution

7. Extreme
   Frequent ”very serious” errors

8. Intolerable
   Database corruption

9. Catastrophic
   System shutdown

10. Infectious
    Shutdown that spreads to others
- **Black-box**: a strategy in which testing is based on requirements and specifications.

- **White-box**: a strategy in which testing is based on internal paths, structure, and implementation.

- **Gray-box**: peek into the “box” with some understanding of how it has been implemented. Parameters, configuration, model.
"Boilerplate": author, date, purpose, **test case ID**
Pre-conditions (including environment)

**Inputs**

**Expected Outputs**

Observed Outputs

Pass/Fail
Part I
Introduction, Testing Process

Part II
Unit Testing:

Testing levels

- **Requirements**
- **System Design** (Architecture, High-level Design)
- **Module Design** (Program Design, Detailed Design)
- **Implementation** of Units (classes, procedures, functions)
- **Unit testing**
- **System Testing** (Integration testing of modules)
- **Module Testing** (Integration testing of units)
- **Verify Implementation**
- **Verify Module Design**
- **Verify System Design**
- **Acceptance Test** (Release testing)
- **Validate Requirements, Verify Specification**

Diagrams illustrate the flow of testing through the different levels.

Kristian.Sandahl@liu.se
## Test table

<table>
<thead>
<tr>
<th>Id</th>
<th>Advanced credits in Computer Science</th>
<th>Advanced credits in total</th>
<th>Masters thesis in subject</th>
<th>Total number of credits</th>
<th>M.Sc., Computer Science</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>120</td>
<td>Computer sc.</td>
<td>120</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>90</td>
<td>Computer sc.</td>
<td>120</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>90</td>
<td>Physics</td>
<td>120</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Part II
Unit Testing:
Objective: to ensure that code implemented the design properly.
The oracle problem

Part I
Introduction,
Testing Process

Part II
Unit Testing:

Input

Test Object

Output

Failure?

Oracle

The oracle problem

Kristian.Sandahl@liu.se
Two Types of Oracles

- **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.

- **Automated**: a system capable of performing the above task.
Black-box Testing

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
1. Exhaustive testing

- **Definition**: testing with every member of the input value space.
- **Input value space**: the set of all possible input values to the program.
  - Sum of two 16 bit integers: $2^{32}$ combinations
  - One test per ms takes about 50 days.
2. Equivalence Class Testing

- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.

- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is equivalent, in terms of testing, to any other value.
Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - Inclusive range of values $x$: 1-50

- Valid equivalence class
  - $? \leq x \leq ?$

- Invalid equivalence classes
  - $x < ?$
  - $x > ?$
Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- **Input condition**
  - range of values $x$: 1-50

- **Valid equivalence class**
  - $1 \leq x \leq 50$

- **Invalid equivalence classes**
  - $x < 1$
  - $x > 50$
Two-variable example

- Validate loan application forms against the rule:
  - *If you are 18 years and older, you can borrow maximally 100.000, but not less than 10.000.*
- Variable: age
  - EC1: age < 18
  - EC2: age >= 18
- Variable: sum
  - EC3: sum < 10.000
  - EC4: 10.000 <= sum <= 100.000
  - EC5: sum > 100.000
## Two-variable example, test-cases

<table>
<thead>
<tr>
<th>Test-case id</th>
<th>Age</th>
<th>Sum</th>
<th>Valid form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>55.300</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>72.650</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>44</td>
<td>9.875</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>60.000</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>87</td>
<td>103.800</td>
<td>No</td>
</tr>
</tbody>
</table>

Arbitrary, valid sums

Arbitrary, valid ages
Guidelines

1. If an input condition specifies a range of values; identify one valid EC and two invalid EC.

2. If an input condition specifies the number (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).

3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.

4. If an input condition specifies a “must be” situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter).

5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.
Identifying the Test Cases

1. Assign a unique number to each EC.

2. Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.

3. Until all invalid ECs have been covered by test cases, write a test case that cover one, and only one, of the uncovered invalid ECs.
Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.

- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.

- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system’s requirements.
3. Boundary Value Testing

Boundary value testing focuses on the boundaries simply because that is where so many defects hide. The defects can be in the requirements or in the code.

The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham’s book).
1. Identify the ECs.

2. Identify the boundaries of each EC.

3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.
Specification: the program accepts four to eight inputs which are 5 digit integers greater than or equal to 10000.

<table>
<thead>
<tr>
<th>Input values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 10000</td>
</tr>
<tr>
<td>Between 10000 and 99999</td>
</tr>
<tr>
<td>More than 99999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of input values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 4</td>
</tr>
<tr>
<td>Between 4 and 8</td>
</tr>
<tr>
<td>More than 8</td>
</tr>
</tbody>
</table>
Boundary value analysis

Part I
Introduction, Testing Process

Part II
Unit Testing:

<table>
<thead>
<tr>
<th>Less than 10000</th>
<th>Between 10000 and 99999</th>
<th>More than 99999</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999</td>
<td>10001</td>
<td>99998</td>
</tr>
<tr>
<td>99999</td>
<td>100000</td>
<td>999999</td>
</tr>
</tbody>
</table>
Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system’s requirements.
xUnit is a set of tools for regression testing
- x denotes a programming language
- JUnit, for Java is one of the earliest and most popular
- TDDC88 has a lab – do that

Recommended primer:

http://www.it-c.dk/~lthorup/JUnitPrimer.html
Object Oriented Framework Development
by Marcus Eduardo Markiewicz and Carlos J.P. Lucena
JUnit interface

Part I
Introduction, Testing Process

Part II
Unit Testing:

Source: Plantir dev zone
Part I
Introduction, Testing Process

Part II
Unit Testing:

Test-Driven Development (TDD)

The mantra of Test-Driven Development (TDD) is “red, green, refactor.”

source: Redmond Developer
Summary - What have we learned today?

Part I: Introduction, Testing process

Part II: Unit Testing:
- Black-box Testing
  1. Exhaustive testing
  2. Equivalence class testing (Equivalence Partitioning)
  3. Boundary value analysis
- xUnit regression testing framework
- Test-Driven Development (TDD)