Lecture 3

Testing Fundamentals

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

1. **Verification & Validation**
   - Introduction
   - Definitions
   - The V Model
   - Black Box vs. White Box
   - Unit Testing
   - Equivalence Classes
   - Testing Strategies

2. **Notes and Further Reading**

Two key SQA concerns

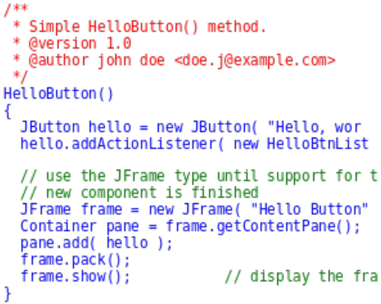
Software verification:

- “Are we building the product right?”

Software validation:

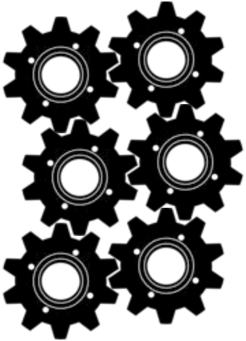
- “Are we building the right product?”

Verification and Validation (V&V) techniques come in two varieties



```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();            // display the fra
}
```



Static V&V:

- Analyses of the code

Dynamic V&V:

- Executing the code

What is testing?

Testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units, or an entire software system are examined by **running the programs** on a computer. All the associated tests are performed according to approved test procedures on approved test cases.

Basic testing definitions

Errors, faults, failures, and incidents:

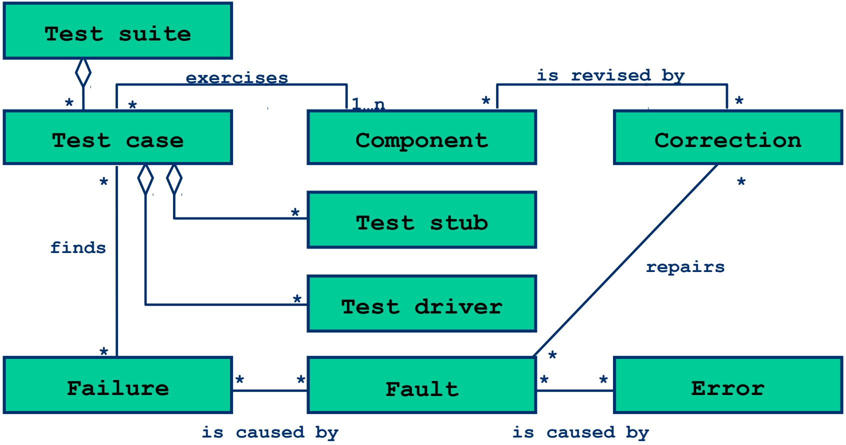
- **Errors** are mistakes that are made by people
- **Faults** are the result of an error in software artifacts
- **Failures** occur when a fault executes
- **Incidents** are consequences of failures
- **Software testing** exercises the software with test cases to find faults or gain confidence in the system (execution based on test cases)

Basic testing definitions

Test cases, stubs, and drivers:

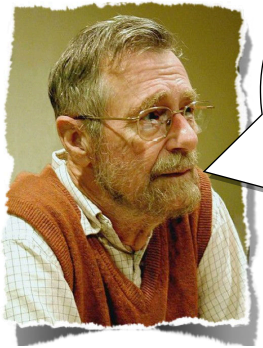
- **Test cases** provide a set of inputs and list a set of expected outputs
- **Test stubs** are partial implementations (simulation) of components on which a component under test depends
- **Test drivers** are partial implementations of components that exercise and depend on the tested component

An overview of testing definitions



What does he mean?

E. Dijkstra



“Program testing can be a very **effective** way to show the *presence of bugs*, but it is hopelessly inadequate for showing their **absence**.”

The ugly truth about software testing

Bugs are inescapable:

- It is impractical to test all operating conditions
- Testing, although incomplete, increases our confidence that software has its desired behaviour

The impracticality of complete testing:

- Large input space
- Large output space
- Large state space
- Large number of possible execution paths
- Subjective requirements

The ugly truth about testing: Large input and state spaces

Exercise: How many tests do we need to exhaustively test:

```
int testMe(int x, int y)
{ ... }
```

Exhaustive testing:

- Testing all possible input combinations is often infeasible

The ugly truth about testing: Large input and state spaces

Exhaustive testing:

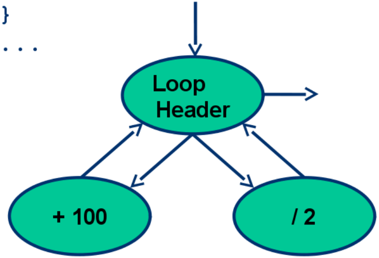
- It gets worse!
- Imagine that you need to test the Java compiler
- Exhaustive testing implies that you would need to try all possible Java programs!
- At best, exhaustive testing is impractical, at worst it’s impossible!

Exercise: How many execution paths?

```
...
for (int i = 0; i < n; ++i) {
    if (a.get(i) == b.get(i))
        x[i] = x[i] + 100;
    else
        x[i] = x[i]/2;
}
...
```

The ugly truth about testing: Large number of possible execution paths

```
...
for (int i = 0; i < n; ++i) {
    if (a.get(i) == b.get(i))
        x[i] = x[i] + 100;
    else
        x[i] = x[i]/2;
}
...
```

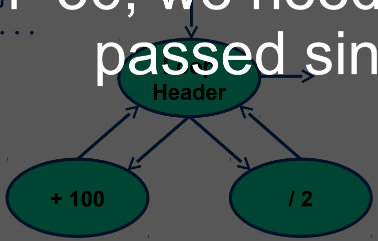


n	Number of paths
1	2
2	4
3	8
10	1024
20	1048576
60	1.15 x 10^{18}

Number of Paths = 2^n

The ugly truth about testing: Large number of possible execution paths

```
...
for (int i = 0; i < n; ++i) {
    if (a.get(i) == b.get(i))
       x[i] = x[i]+100;
    else
       x[i] = x[i]/2;
}
...
```



n	Number of paths
1	2
2	4
3	8
10	1024
20	1048576
60	1.15 x 10^{18}

Number of Paths = 2^n

If each test takes 10^{-3} seconds, with n=36, we need more time than has passed since the big bang!

Quality assurance is a risk mitigation exercise

Striking a balance:

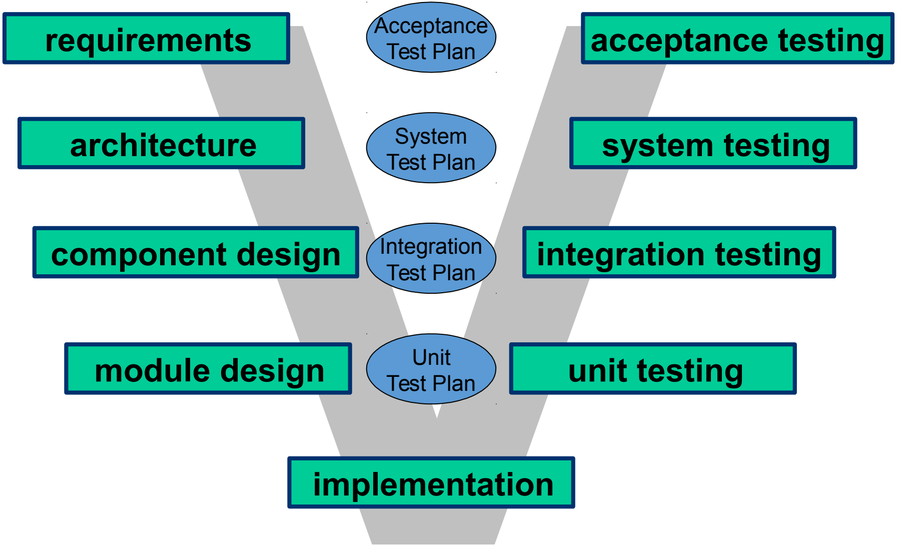
- A balance must be reached between the cost of testing and the risk of missing bugs

The ugly truth about testing: Bugs, bugs, bugs

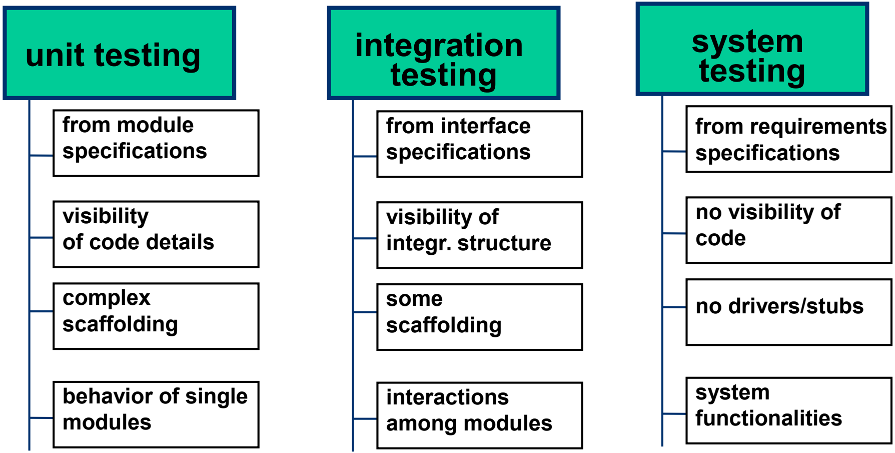
The truth about bugs:

- Not all known bugs are fixed
  - Not enough time
  - Too risky to fix (might create more problems)
- When is a bug *really* a bug?
  - Time pressure may lead to downgrading bugs to feature requests
- Since many people struggle to process criticism, QA personnel are not the most popular people ㅆ

The V model of development



Differences between the testing



unit testing

- from module specifications
- visibility of code details
- complex scaffolding
- behavior of single modules

integration testing

- from interface specifications
- visibility of integr. structure
- some scaffolding
- interactions among modules

system testing

- from requirements specifications
- no visibility of code
- no drivers/stubs
- system functionalities

Testing visibility specifiers

Black-box testing:

- Tests that are written without considering how the code is written/structured (based on specs)
- Treats the system like an opaque box that accepts inputs and checks outputs

White-box testing:

- Tests that are written with an understanding of the structure of the code
- Tests the inner workings of the system itself

Black-box or white-box?

(1) **Unit tests?** White box

(2) **System tests?** Black-box

(3) **Stress tests?** Black-box

(4) **Integration tests?** White-box

(5) **Acceptance tests?** Black-box

Black-box vs. White-box

Black-box:

+ Check conformance with spec

+ Scales up

- Depends on detail in spec

- Does not tell how much of the system is being tested (coverage)

White-box:

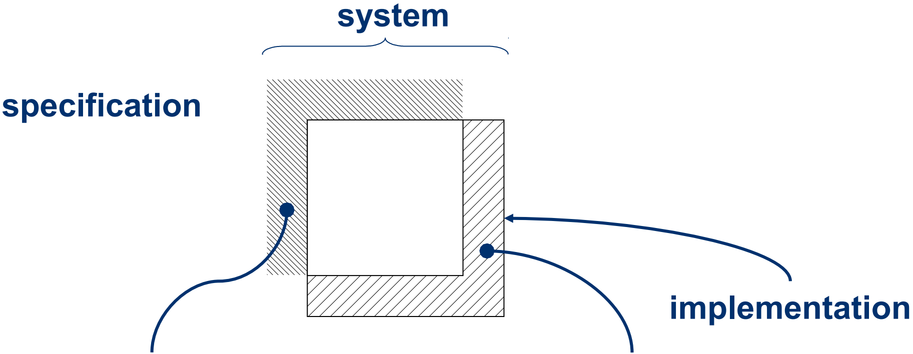
+ Allows for measuring "test coverage"

+ Based on control or data flow coverage

- Doesn't scale up well

- Cannot reveal missing functionality!

Do we need both types of tests?



- Missing functionality cannot be revealed by white-box techniques
- Unexpected functionality cannot be revealed by black-box techniques

Which input set will find the bug?

```
...
if x > y then
    Max := x;
else
    Max := x;
end if;
...
```

Input sets:

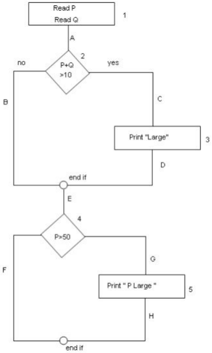
{x=3, y=2; x=2, y=3}

{x=3, y=2; x=4, y=3; x=5, y=1}

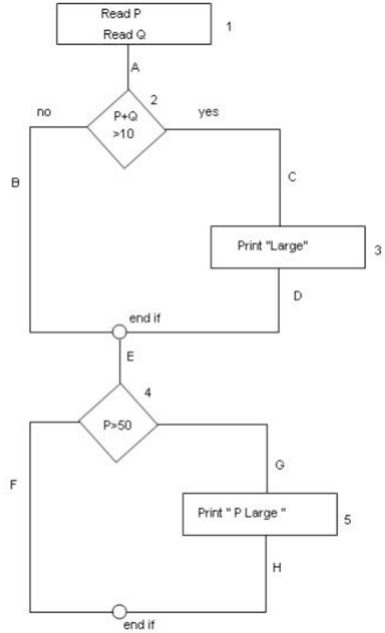
Flow graph for white-box testing

- To help the programmer to systematically test the code
  - Each branch in the code (such as if and while statements) creates a node in the graph

```
Read P
Read Q
IF P+Q > 100 THEN
      Print "Large"
ENDIF
If P > 50 THEN
      Print "P Large"
ENDIF
```



```
Read P
Read Q
IF P+Q > 100 THEN
        Print “Large”
ENDIF
If P > 50 THEN
        Print “P Large”
ENDIF
```

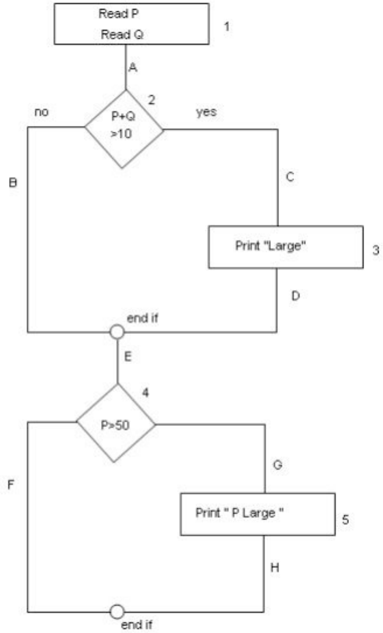


Flow graph for white-box testing

- The testing strategy has to reach a targeted coverage of statements and branches; the objective can be to:
  - cover all possible statements
  - cover all possible branches
  - cover all possible pathes

Cover all possible statements

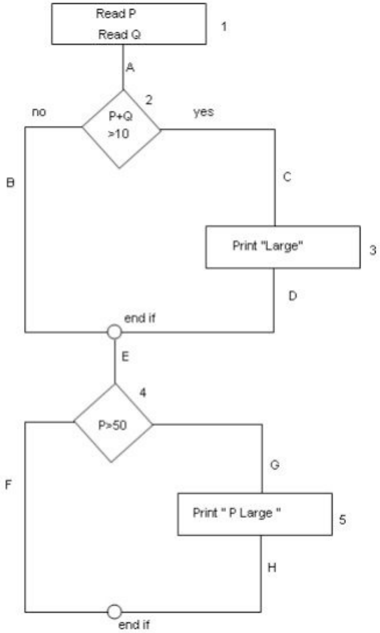
```
Read P
Read Q
IF P+Q > 100 THEN
    Print "Large"
ENDIF
If P > 50 THEN
    Print "P Large"
ENDIF
```



1A-2C-3D-E-4G-5H

Cover all possible branches

```
Read P
Read Q
IF P+Q > 100 THEN
        Print “Large”
ENDIF
If P > 50 THEN
        Print “P Large”
ENDIF
```

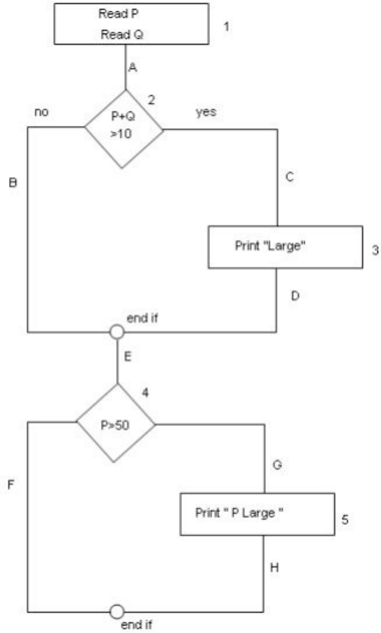


1A-2C-3D-E-4G-5H

1A-2B-E-4F

Cover all possible paths

```
Read P
Read Q
IF P+Q > 100 THEN
    Print “Large”
ENDIF
If P > 50 THEN
    Print “P Large”
ENDIF
```



1A-2B-E-4F
1A-2B-E-4G-5H
1A-2C-3D-E-4G-5H
1A-2C-3D-E-4F

Testing stages

Development testing:

- Tests that are typically written by system designers and programmers

Release testing:

- Separate QA team produces tests to check the system before it is released

User testing:

- Tests run to ensure that users will be satisfied with the product

Development testing Granularities

Unit testing:

- Target: particular program unit (class)
- Focuses on testing the functionality of an object or a method

Development testing
Unit testing

Unit testing:

- The name of the game is to maximize coverage of the system while minimizing execution time
- We want our test cases to cover all of the possible paths through our methods
- However, we want our tests to finish quickly so that we don't delay the release process!

Development testing
Unit testing

Key phases:

- Setup part: Prepares the system for executing the test case
- Call part: Call the method under test
- Assertion part: Check that the system state is in its expected form
- Teardown part: Reset the system to its pre-setup state

Development testing
Unit testing goals

Goals in designing unit tests:

(1) Show that (when used as expected) the unit does what it is supposed to do

(2) If there are defects in a unit, the test cases should reveal them!

Need two types of unit tests to do this!

Development testing
Unit testing strategies

Partition testing:

- Identify groups of inputs that have common characteristics (should be processed the same way)

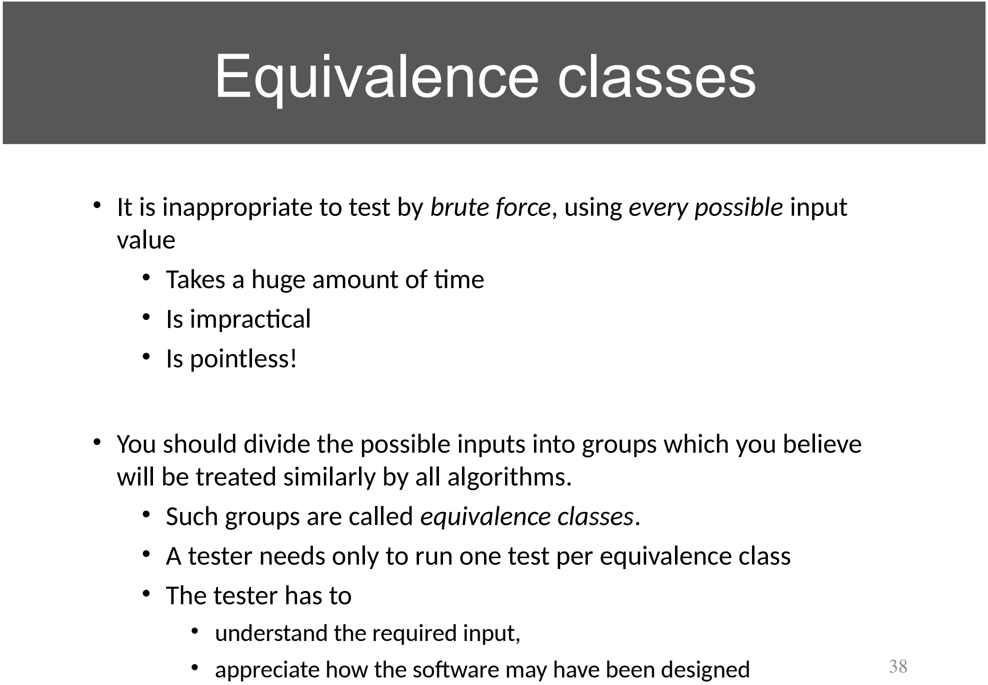
Guideline-based testing:

- Use guidelines based on previous experience of the kinds of errors that programmers often make
- This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur

Development testing
Partition testing

Equivalence partitioning:

- Inputs and outputs can be thought of as members of sets with common characteristics
- Rule of thumb
  - Select test cases on the edge of the partitions
  - Select a test case near the middle as well

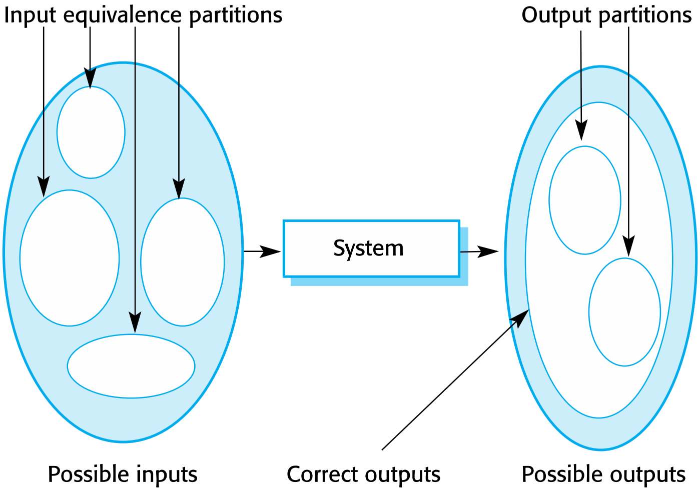


Equivalence classes

- It is inappropriate to test by *brute force*, using *every possible* input value
  - Takes a huge amount of time
  - Is impractical
  - Is pointless!

- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - Such groups are called *equivalence classes*.
  - A tester needs only to run one test per equivalence class
  - The tester has to
    - understand the required input,
    - appreciate how the software may have been designed

Development testing
Partition testing

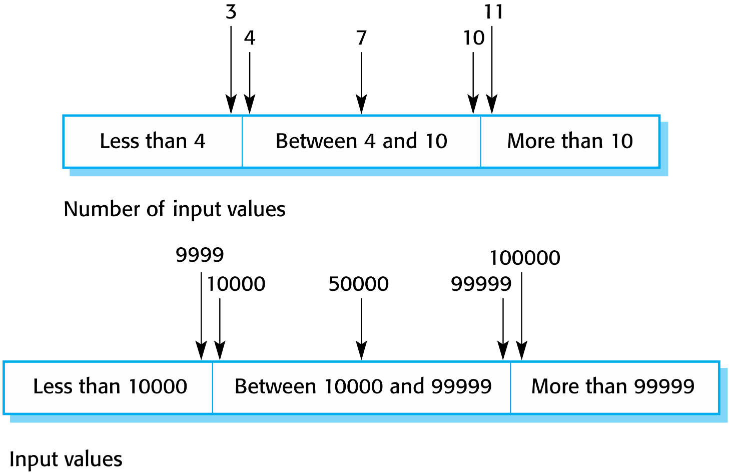


Examples of equivalence classes

- Valid input is a month number (1-12)
  - Equivalence classes are: [-∞..0], [1..12], [13.. ∞]
- Valid input is one of ten strings representing a type of fuel
  - Equivalence classes are
    - 10 classes, one for each string
    - A class representing all other strings

Development testing
Identifying partitions

"the program accepts four to ten inputs which are five-digit integers greater than 10,000."



Combinations of equivalence classes

- Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
  - If there are 4 inputs with 5 possible values there are 5^4 (i.e. 625) possible system-wide equivalence classes.
- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to *affect the interpretation* of another.
- You should test a few other random combinations of equivalence classes.

Example equivalence class combinations

- One valid input is either ‘Metric’ or ‘US/Imperial’
  - Equivalence classes are:
    - Metric, US/Imperial, Other
- Another valid input is maximum speed: 1 to 750 km/h or 1 to 500 mph
  - Validity depends on whether metric or US/imperial
  - Equivalence classes are:
    - [-∞..0], [1..500], [501..750], [751.. ∞]
- Some test combinations
    - Metric, [1..500] valid
    - US/Imperial, [1..500] valid
    - Metric, [501..750] valid
    - US/Imperial, [501..750] invalid

Testing at boundaries of equivalence classes

- More errors in software occur at the boundaries of equivalence classes
- The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
  - E.g. The number 0 often causes problems
- *E.g.*: If the valid input is a month number (1-12)
  - Test equivalence classes as before
  - Test 0, 1, 12 and 13 as well as very large positive and negative values

Development testing
Guideline-based testing

Guidelines for sequences, arrays, or lists:

(1) Test software with sequences that have only one value

- Common mistake! Assume sequences have multiple values!

(2) Use different sequences of different sizes in different tests (increases coverage!)

(3) Derive tests so that the first, middle, and last elements of the sequence are accessed (reveals partition boundaries!)

Development testing
Guideline-based testing

More guidelines:

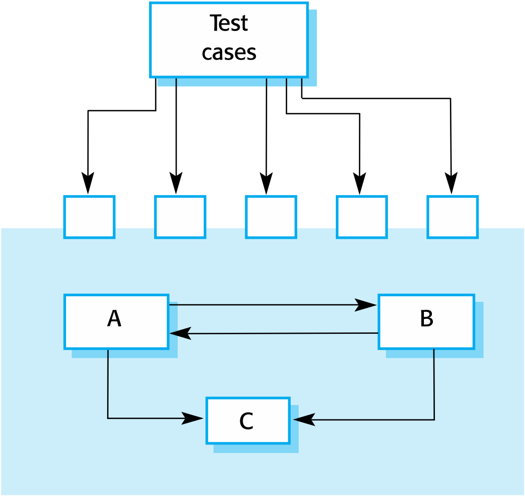
- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small

Development testing Granularities

Component (integration) testing:

- Target: the combination of units that make up a component
- Focuses on the interface that a component exposes to other components

Development testing
Integration testing



Development testing
Types of component interfaces

Parameter:

- Passing parameters through method calls

Shared memory:

- Accessing a shared program element

Procedural:

- Interfaces in the typical (OO) sense

Message passing:

- A message is sent to request some service (e.g., client-server applications)

Development testing Types of interface errors

Misuse:

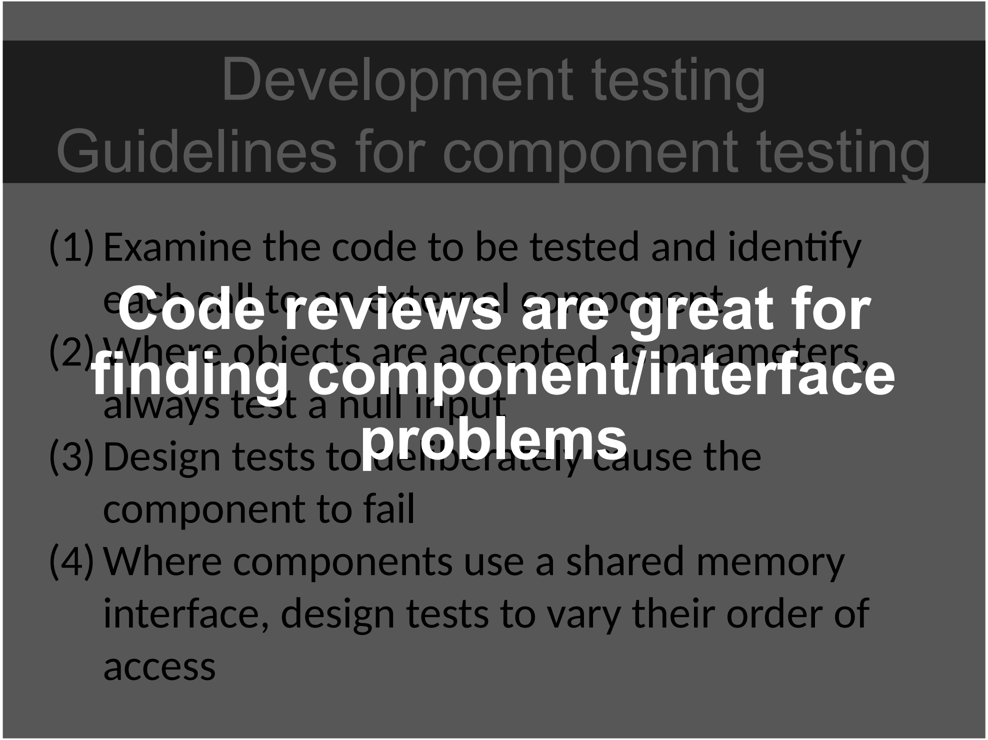
- Caller makes an error in the use of an interface

Misunderstanding:

- Caller makes an invalid assumption about the use of an interface

Timing:

- Some interfaces are time-sensitive
- Even if a caller uses an interface correctly, it may be too early/late



Development testing
Guidelines for component testing

(1) Examine the code to be tested and identify each call to an external component

(2) Where objects are accepted as parameters, always test a null input

(3) Design tests to deliberately cause the component to fail

(4) Where components use a shared memory interface, design tests to vary their order of access

Code reviews are great for finding component/interface problems

Development testing Granularities

System testing:

- Target: the combination of components that make up a subsystem or the entire system
- Focuses on testing the interactions between components

Development testing
System testing

How is it different from component testing?

- All components are in the testing loop
- Crosses team boundaries

Development testing
Writing good system tests

Use cases:

- Essentially, use cases describe expected system behaviour
- They make for an ideal place to start with your system tests

Sequence diagrams can help to establish the expected system test behaviour!

Development testing How much system testing is enough?

Policies:

- Each project should outline testing policies to reach the level of risk that is comfortable

Some example guidelines:

(1) **All system functions** that are accessed through menus should be tested

(2) Combinations of functions that are accessed through the same menu must be tested

(3) Where user input is provided, all functions must be tested with **correct and invalid** input

Release testing

What is release testing?

- Similar to system testing, release testing is performed on the entire system and is based on use cases
- Unlike system tests, release tests:
  - Must be written by (and executed by) a team that does **not** report to development
  - Check that the system meets requirements (system tests are out to find bugs)

Release testing
How does it differ from system testing?

Black-box vs. white box:

- Technically speaking, some development organizations write white-box system tests
- Release tests are never white-box tests, they are always black-box!

User testing

What are user tests?

- User tests are performed by customers to ensure that the software meets their expectations
- Useful approach for ***validation***

User testing
Types of user tests

1. Alpha tests:

- A select group of users works closely with the development team to test early releases
- Defects are expected, but alpha users get access to “bleeding edge” features!

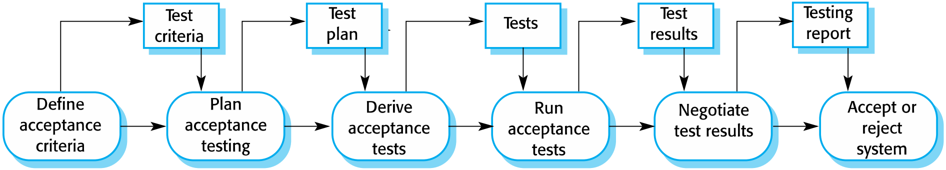
2. Beta tests:

- Beta testing involves delivering a system to a number of potential customers who agree to use that system.
- A larger group of users is allowed to experiment with a release

User testing
Types of user tests

3. Acceptance tests:

- Customers test a software system to make sure it meets their expectations and can be adopted in their environment



Strategies for Testing Large Systems

- Big bang testing versus integration testing
  - In *big bang* testing, you take the entire system and test it as a unit
  - A better strategy in most cases is *incremental testing*:
    - You test each individual subsystem in isolation
    - Continue testing as you add more and more subsystems to the final product
    - Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture
      - Horizontal testing can be used when the system is divided into separate sub-applications

Top down testing

- Start by testing just the user interface.
- The underlying functionality are simulated by *stubs*.
  - Pieces of code that have the same interface as the lower level functionality.
  - Do not perform any real computations or manipulate any real data.
- Then you work downwards, integrating lower and lower layers.
- The big drawback to top down testing is the cost of writing the stubs.

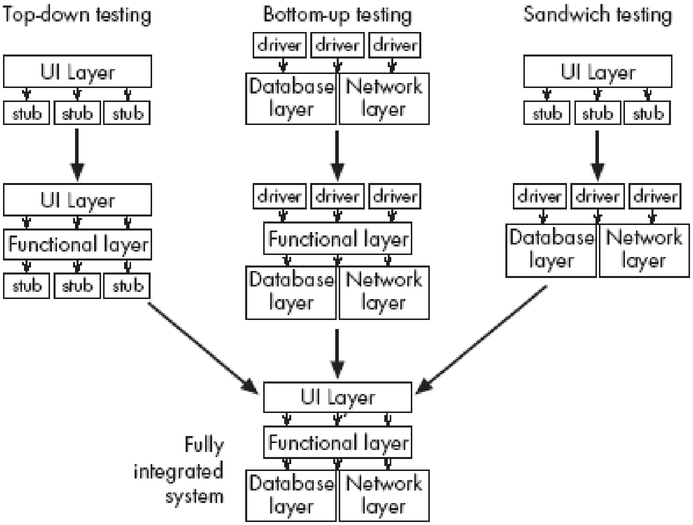
Bottom-up testing

- Start by testing the very lowest levels of the software.
- You needs *drivers* to test the lower layers of software.
  - Drivers are simple programs designed specifically for testing that make calls to the lower layers.
- Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

Sandwich testing

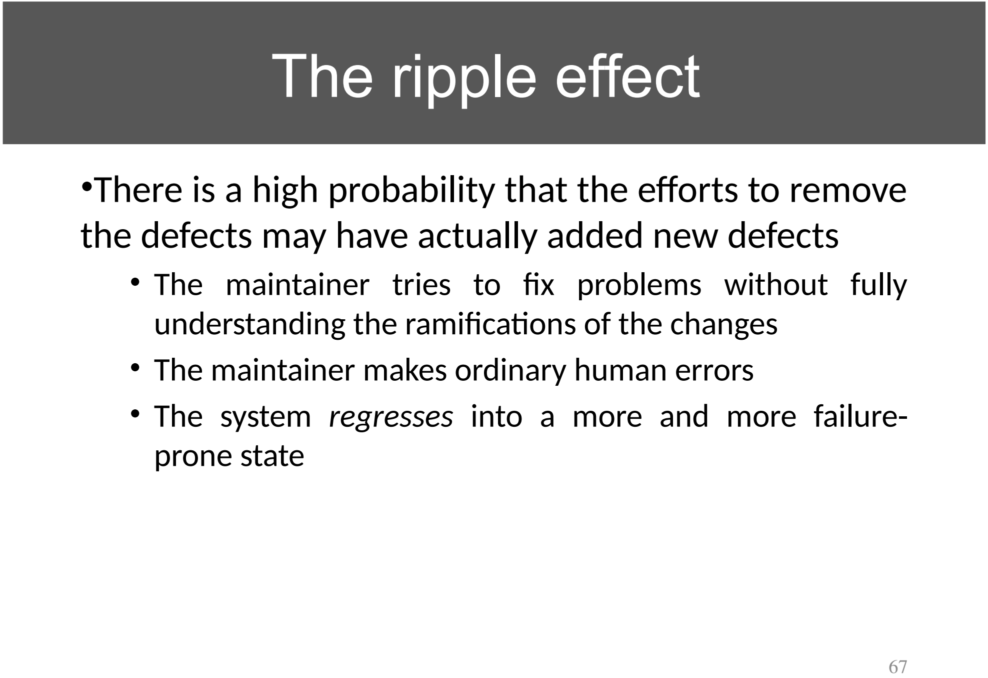
- Sandwich testing is a hybrid between bottom-up and top down testing.
- Test the user interface in isolation, using stubs.
- Test the very lowest level functions, using drivers.
- When the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

Vertical strategies for incremental integration testing



The test-fix-test cycle

- When a failure occurs during testing:
  - Each failure report is entered into a failure tracking system.
  - It is then screened and assigned a priority.
  - Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
  - Some failure reports might be merged if they appear to result from the same defects.
  - Somebody is assigned to investigate a failure.
  - That person tracks down the defect and fixes it.
  - Finally a new version of the system is created, ready to be tested again.



The ripple effect

- There is a high probability that the efforts to remove the defects may have actually added new defects
  - The maintainer tries to fix problems without fully understanding the ramifications of the changes
  - The maintainer makes ordinary human errors
  - The system *regresses* into a more and more failure-prone state

Regression testing

- It tends to be far too expensive to re-run every single test case every time a change is made to software.
- Hence only a subset of the previously-successful test cases is actually re-run.
- This process is called *regression testing*.
  - The tests that are re-run are called regression tests.
- Regression test cases are carefully selected to cover as much of the system as possible.

- The “law of conservation of bugs”:
  - ***The number of bugs remaining in a large system is proportional to the number of bugs already fixed***

Outline

1. Verification & Validation
   - Introduction
   - Definitions
   - The V Model
   - Black Box vs. White Box
   - Unit Testing
   - Equivalence Classes
   - Testing Strategies

2. Notes and Further Reading

Reading Material

Supplemental

- [Som16, Chapter 8] (Software testing)

References

[Som16] Ian Sommerville.
Software Engineering.
Pearson, 10th edition, 2016.
http://software-engineering-book.com.