THE UNIVERSITY OF QUEENSLAND

CSSE2003
Software Engineering Studio

Semester 2, 2009

5: Introduction to Software Testing

Lecture Summary

- Introduction
  - what is software testing?
  - systematic testing
  - levels of testing
  - testing tasks
- Test case selection
- Test case execution

What is Software Testing?

- A quality assurance activity
- Execution of software:
  - in a controlled environment,
  - with known inputs, and
  - where actual outputs are compared with expected outputs
- Necessary since no development process is perfect

Objectives of Software Testing

- Executing a program with the intent of finding an error
- Generating confidence in software (justified only if tests are effective)
- A good test case is one that has a high probability of finding an as-yet undiscovered error

Exhaustive Testing?

- Impossible for all realistic programs
  - consider adding two 32 bit integers:
  - at one test per microsecond, it would take half a million years!
- “Testing demonstrates the **presence** of bugs, not their **absence**” [Dijkstra]

Terminology

Error:

- mistake made by a developer

Fault:

- result of an error
- the difference between the incorrect program and a correct version

Failure:

- executing a program containing faults may lead to failure
- the difference in the results of the incorrect and correct programs

Alternatives to Testing

- Reviews involve human review of work products
  - can be applied to any human-readable work product (early in the life cycle)
  - finds faults, not failures (no debugging!)
  - covered in CSSE3002
- Formal methods apply mathematical reasoning to demonstrate program correctness
- Other techniques for increasing assurance
  - static analysis, e.g. data flow, model checking
  - language choice and appropriate design methods

Systematic Testing

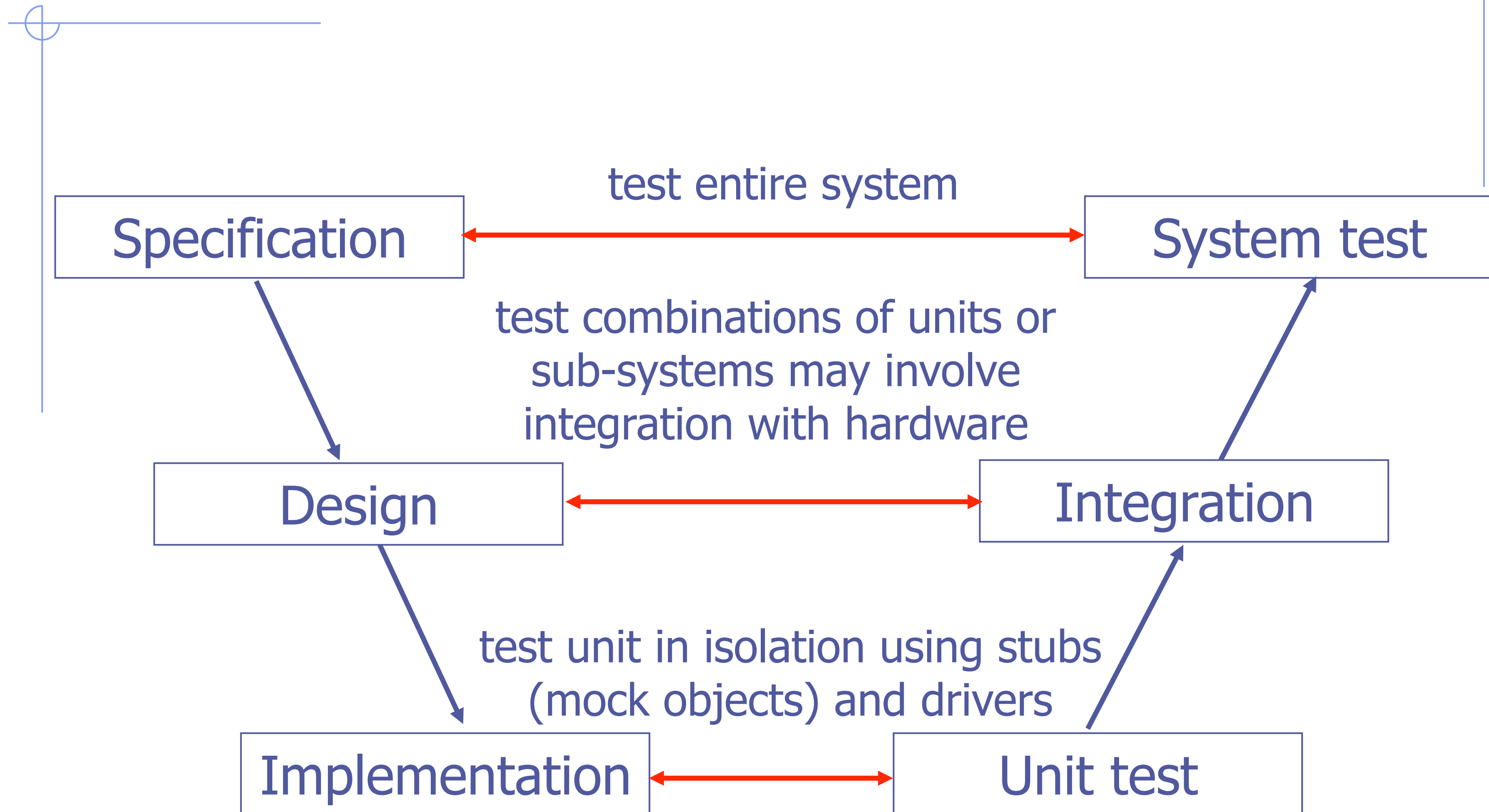
- Ad-hoc approach:
  - ineffective: no influence on design decisions
  - expensive: no re-use during maintenance
- Systematic approach:
  - planned: permits design for testability
  - documented: review/evaluate tests
  - maintained: retest after modifications
- Writing and maintaining unit tests is costly

  (e.g. minimizing public API and reducing cyclomatic complexity in the code to reduce this cost and make high-coverage test code faster to write and easier to maintain)

Testing is not Debugging

- Testing aims to identify failures,
  debugging aims to locate faults for correction.
- Testing can be planned, documented and maintained.
- Testing at the system level can be done without design knowledge.

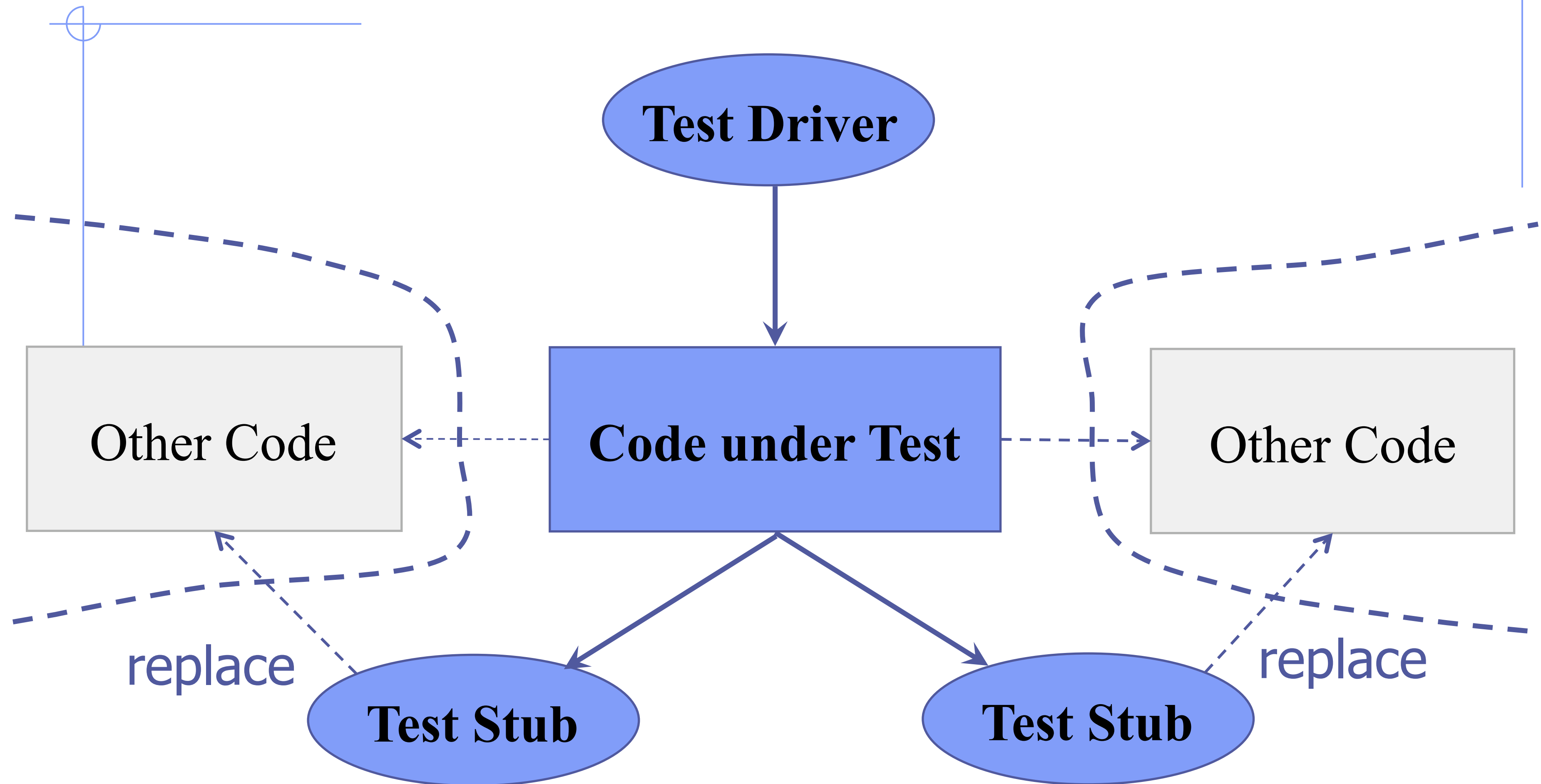
V Model and Testing



Levels of Testing

- Unit testing
  - test unit in isolation using stubs (mocks) and drivers
- Integration testing
  - test combinations of units or sub-systems
  - may involve integration with hardware
- System testing
  - test entire system
- Acceptance testing
  - user- or client-directed system testing, preferably in context

Test Scaffolding



Test Drivers

- Calls unit under test
  - initialises unit under test
  - provides inputs
  - captures outputs and exceptions
  - may compare actual to expected behaviour
- Benefits
  - isolates unit under test
  - provides independence from particular tests
  - allows rerunning of tests

Test Stubs (Mock Objects)

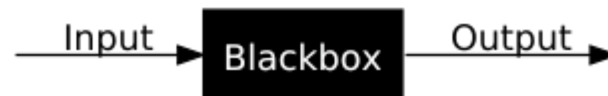
- Called by unit under test and replaces other units
  - simulates unit(s) it replaces
  - may accept or supply data
- Benefits
  - isolates unit under test
  - Can generate reproducible behaviour

Testing Tasks

- Planning
  - provide test scaffolding (test drivers and stubs)
  - create test cases: input and expected output
- Execution
  - execute test cases and capture actual output
- Evaluation
  - compare actual and expected output

Test Case Selection Strategies

- **Black-box** testing
  - also known as functional testing
  - test inputs are based on specification (checking a software component at its interface)
  - ignores internal details of implementation



- **White-box** testing
  - test inputs are based on implementation
  - ignores non-implemented behaviour

Black-Box Testing

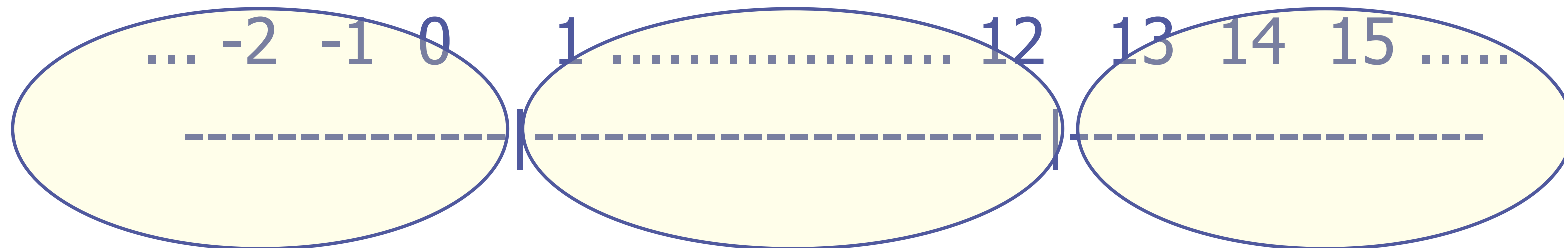
- Choose special values for state and parameters
  - both exceptional and normal case values
- Test combinations of special values
  - combinatorially explosive: testing all combinations may be infeasible
  - there may be dependencies
- Black box strategies:
  - to reduce the number of test cases to a necessary minimum
  - to select the right test cases to cover all possible scenarios.

Black Box Strategies: Equivalence partitioning

- Partition input values into equivalence classes
- Partitioning is derived from the specification of the unit's behavior under test
- An input has certain ranges which are valid and other ranges which are invalid
- Testing any input from an equivalence class intended to address entire class

Equivalence partitioning example

- A function which has a parameter "month" of a date



invalid partition 1 (<=0)

valid partition

invalid partition 2 (>=13)

- *if internally the function may have a differentiation of values between 1 and 6 and the values between 7 and 12, how do we test it?*

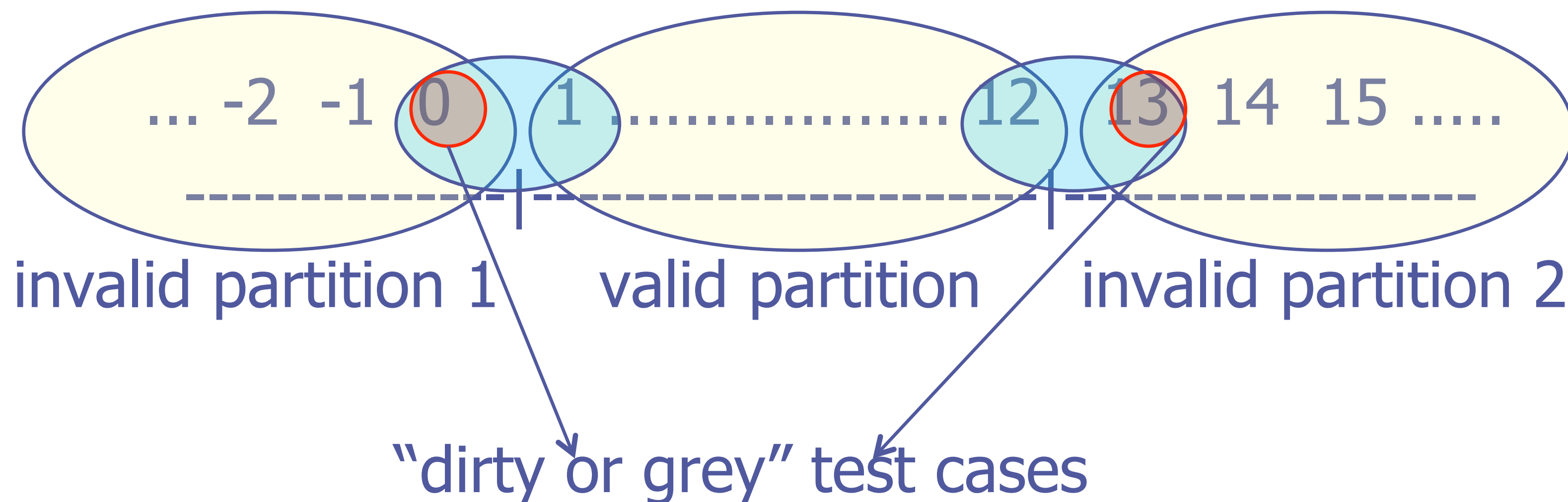
Black Box Strategies: Boundary-value analysis

- Special value testing, extreme value testing
- Input values at or near the boundaries (e.g, minimum and maximum) of allowable values
- The boundaries of input ranges to a software component are liable to defects
- Determine boundaries by applying the equivalence partitioning technique

Boundary-value analysis example

- Code checking months (a common programming error)

```
if (month >= 0 && month < 13)
```



Boundary-value analysis (cont.)

- A "dirty" test case should lead to a correct and specified input error treatment such as
  - the limiting of values
  - the usage of a substitute value
  - in case of a program with a user interface, it has to lead to warning and request to enter correct data
- The boundary value analysis **can** have 6 test cases: n, n-1, and n+1 for the upper limit; and n, n-1, and n+1 for the lower limit.

Other strategies

- The natural boundaries of the data types used in the program, examples,
  - a signed value, around zero (-1, 0, +1)
  - a sign problem when a value turns out to be negative, although the programmer always expected it to be positive
  - the natural lower and upper limit of the data type
    - an unsigned 8-bit (the range of 0 to 255)
    - check how the program reacts at an input of -1 and 0 as well as 255 and 256
- Error guessing: use intuition and experience to identify probable areas of potential errors
    - a division by zero

Black-Box – Example 1

- Function f(p1,p2)
  - p1 is integer restricted to the interval [0,100]
  - p2 belongs to enumerated type {red,green,blue}
- Special values for p1: the interval rule
  - normal case values: {0,50,100}
  - exceptional values: {-100,-1,101,1000}
- Special values for p2
  - normal case values: {red,green,blue}
  - exceptional values: none
- Test cases: cover all 21 combinations

Black-Box – Example 2

- Function `int Position (ItemType item, ListType list)`
  - *returns position of item in the list*
  - *return –1 if item is not in the list*

Black-Box – Example 2 (cont.)

- Example test cases
  - empty list
  - list with one element
    - item absent
    - item present
  - list with more than one element
    - item absent
    - item present: in first, middle, and last position
    - multiple occurrences of same item?

White-Box Testing

- Also known as glass-box or structural testing.
- Test inputs based on internal structure of the implementation
  - still need specification to determine expected output
- Interested in executing all the source code
  - statement coverage: execute every statement
  - Branch (or condition) coverage: exercise every branch
  - path coverage: execute every path

White-Box – Example

```
public void Demo (int x)
{
   if  (x > 0)
      pos = pos+1;
   if  (x % 2 == 0)
      even = even+1;
}
```

- statement coverage:
  x = 2
- branch coverage:
  x = 2 (T, T)
  x = -1 (F, F)
- path coverage:
  x = 2 (T, T)
  x = -1 (F, F)
  x = 1 (T, F)
  x = -2 (F, T)

If Statement

```
if (Cond1 && Cond2 ) {
                Statement1;
  } else {
                Statement2;
  }
```

- For branch coverage, two cases sufficient
  - **(***Cond1* **&&** *Cond2* **)** evaluate to true and to false
- For multiple-condition coverage, four cases necessary
  - all combinations of true and false for both *Cond1* and *Cond2*

While Statement

```
S1;
while (Cond) {
    S2;
}
S3;
```

Paths:

S1; S3

S1; S2; S3

S1; S2; S2; S3

…

Loop Coverage

- An infinite number of paths potentially exist for a loop.
- Loop coverage requires test cases that include 0, 1, and multiple iterations of the loop (not always feasible)
  - zero iterations: test S1; S3
  - one iteration: test S1; S2; S3
  - two iterations: test S1; S2; S2; S3
- Many loops have explicit limits
  - e.g., for (int i = 0; i < MAX; i++) ...

Systematic Unit Testing

- For effective testing, we need
  - **controllability**: being able to establish arbitrary testing conditions
  - **observability**: being able to observe effects of tests
- Difficult to test a unit in its production environment
  - due to poor controllability and observability
  - example: most applications never propagate exceptions from library classes

Example - IntSet

```
public class IntSet {
   public static final int MAX_SIZE = 100;
   public IntSet();

   public void insert (int x)
       throws DuplicateExc, FullExc;

   public void remove (int x)
       throws NotFoundExc;

   public Boolean isMember (int x);
   public int size();
}
```

Throwing and Catching Exceptions

- `IntSet` defines three programmer-defined exceptions

  `DuplicateExc, FullExc, NotFoundExc`

  ```
  public void remove (int x) {
     if (x is not in the set)
      throw new NotFoundExc();
     ...
  }
  ```

- Code that uses IntSet is responsible for catching exception so when testing, the test driver expects to catch exceptions

Test Drivers

- Careful driver implementation requires a lot of code
  - for try/catch blocks (exceptions)
  - for error messages and statistics
- Manual implementations hard to maintain
  - long, tedious, and repetitive
- Driver code can be partially generated

So, we need a testing framework that supports all these tasks such as JUnit