COMP 2710
Software Construction

Chapter2: Intro to Unit Test

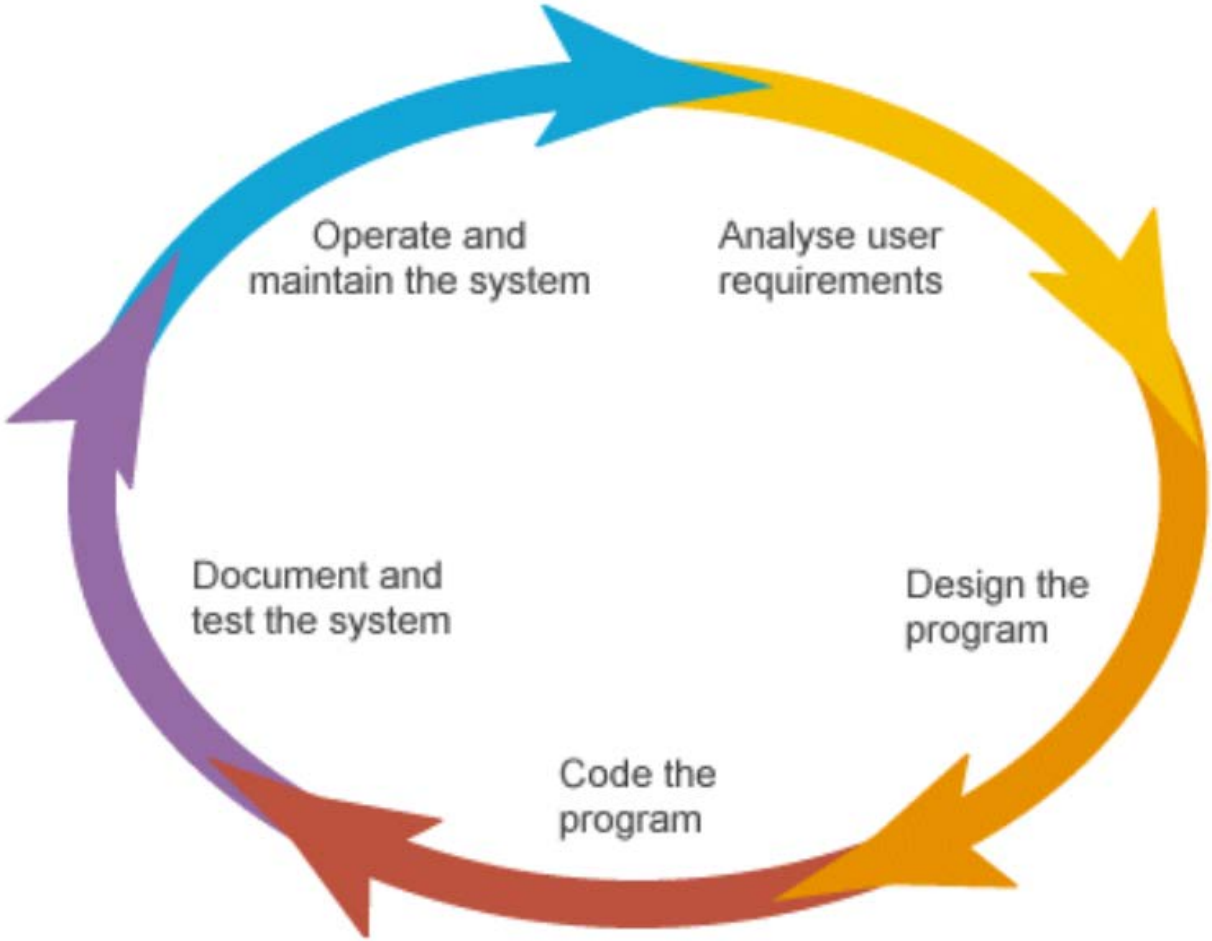
Dr. Xuechao Li

AUBURN
UNIVERSITY

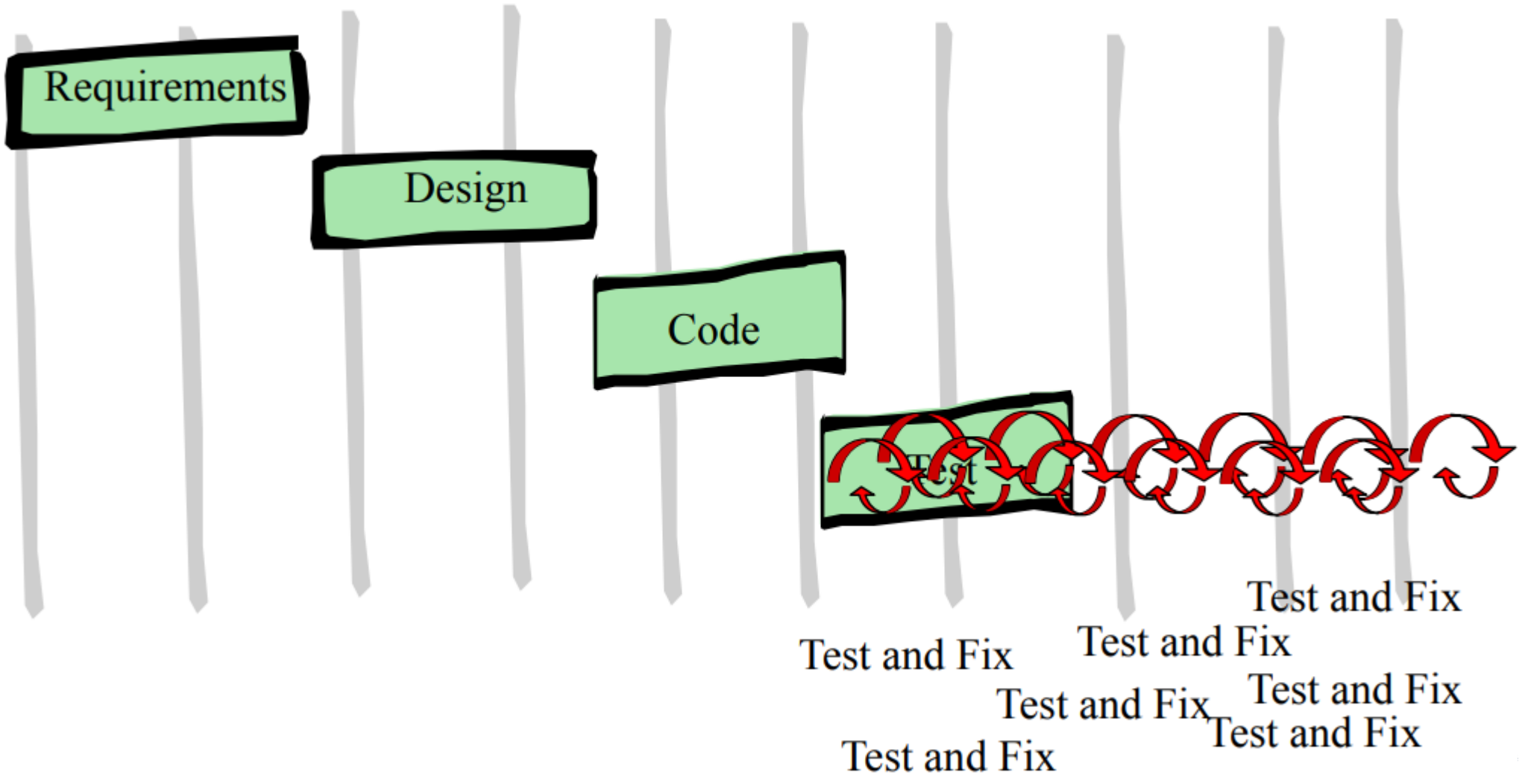
SAMUEL GINN
COLLEGE OF ENGINEERING

Partial source: http://indico.ictp.it/event/a12226/session/67/contribution/41/material/0/1.pdf
https://wingman-sw.com/slides/TDD-Embedded-C++v1.1.ppt.pdf

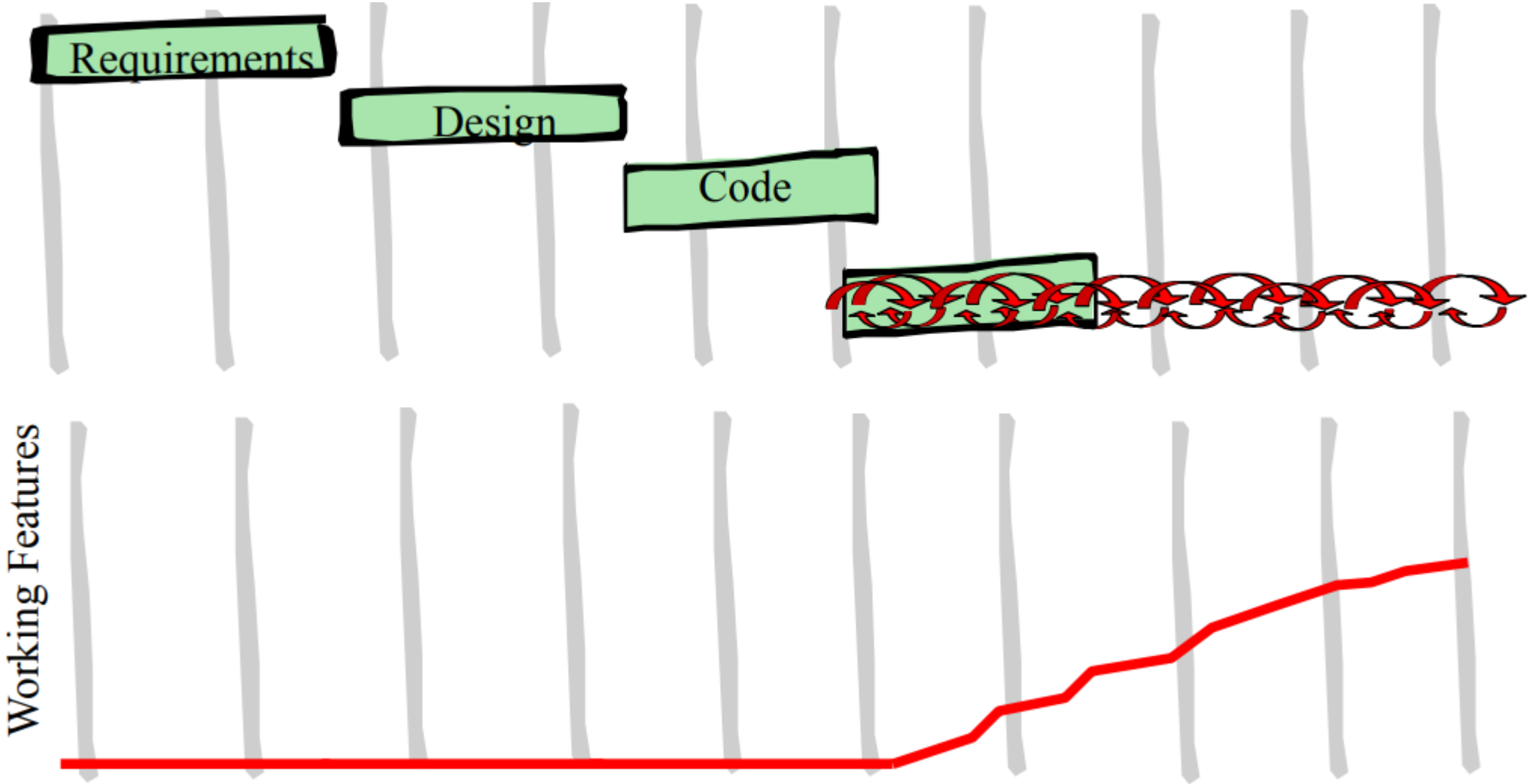
Software Development Lifecycle



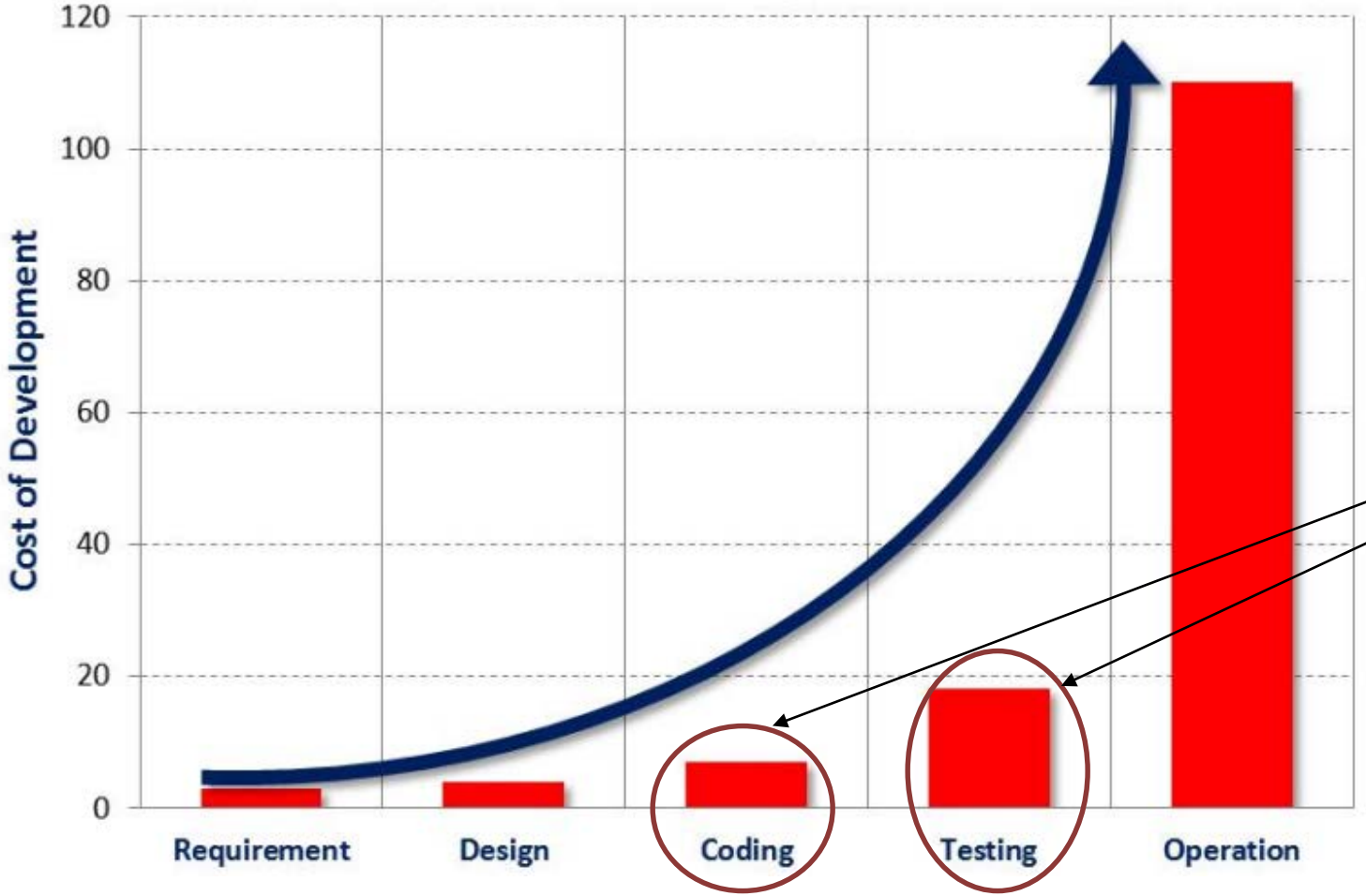
Software Development Flow

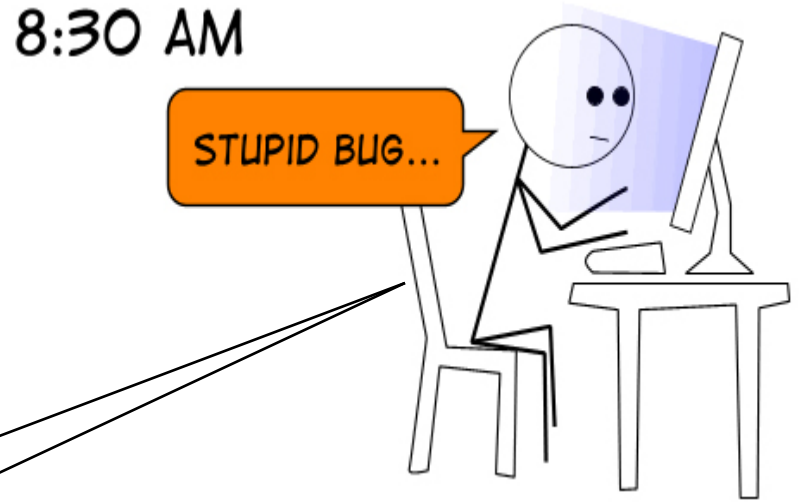


Software Development Workload



Where are we?





What is a Unit?

- Smallest testable part of an application
- Definition differs depending on the type of programming under discussion
  - Procedural Programming: an individual function or procedure
  - Object-Oriented Programming: an interface such as a class

Unit Testing

- **Static Testing**
  - ➢Focuses on prevention
  - ➢Done at compile time
  - ➢Tests code only not output of the code
  - ➢Can find: syntax errors, ANSI violations, code that does not conform to coding standards, etc.
  - ➢100% statement coverage in short time

Unit Testing

- **Dynamic Testing**
  - Focuses on elimination of logical errors
  - Performed during run time
  - Two kinds: White and black box testing
  - Finds bugs in executed pieces of software
  - Limited statement coverage with long run time

Traditional Testing

- Test the system as a whole
  - Higher level of complexity
  - Individual components are rarely tested
  - Isolating errors is problematic
- Testing Strategies
  - Print Statements
  - Use of Debugger
  - Test Scripts

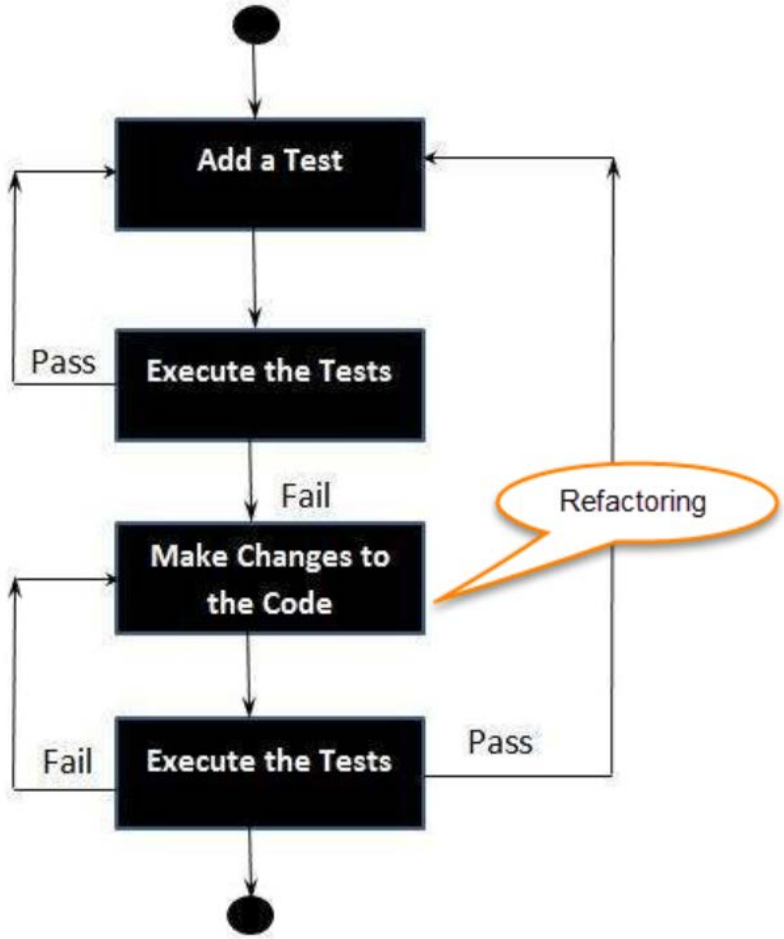
What test we learn

- Unit Testing (Developers)
  - ➢Individual components (class or subsystem)
- Integration Testing (Developers)
  - ➢Aggregates of subsystems
- System Testing (Developers)
  - ➢Complete integrated system
  - ➢Evaluates system's compliance with specified requirements
- Acceptance Testing (Client)
  - ➢Evaluates system delivered by developers

Test Driven Development(TDD)

- Starts with designing and developing tests for every small functionality of an application
  - ➢clearer
  - ➢simple
  - ➢bug-free (hopefully)

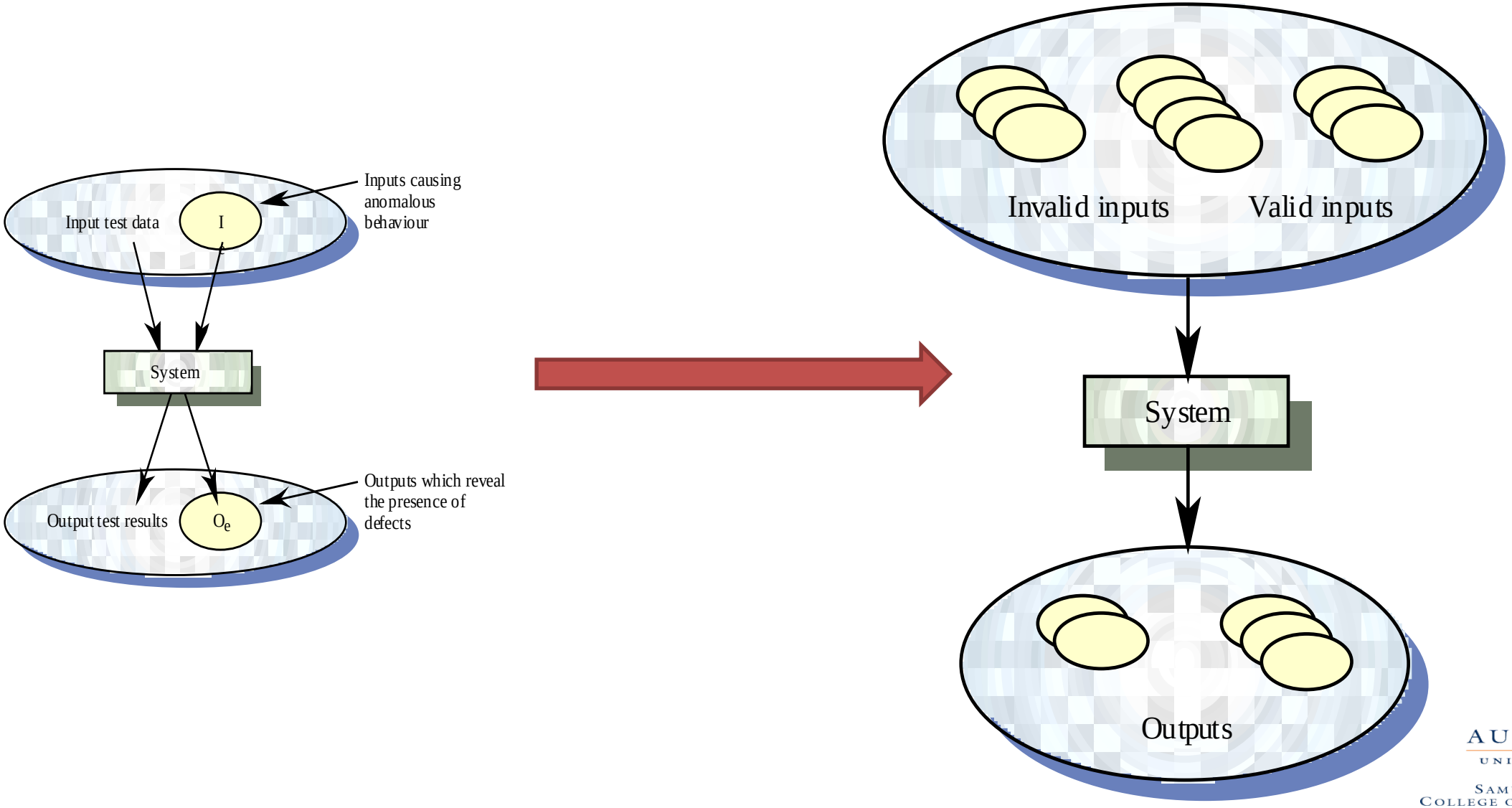
Steps to perform TDD



Black Box Testing

- focuses on input/output of each component or call
- without knowledge of how the class under test is implemented
  - Program is treated as a black box.
  - Implementation details do not matter.
  - Requires an end-user perspective.
  - Criteria are not precise.
  - Test planning can begin early.

Equivalence Partitioning



Equivalence Partitioning

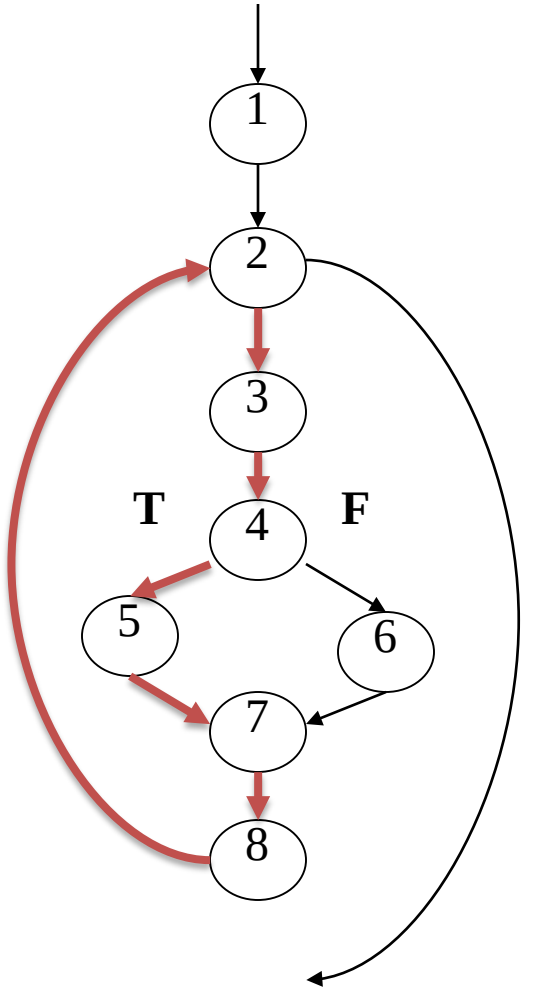
- Basic idea: consider input/output domains and partition them into equiv. classes
  - For different values from the same class, the software should behave equivalently
- Use test values from each class
  - Example: if the range for input x is 2..5, there are three classes: “<2”, “between 2..5”, “5<”
  - Testing with values from different classes is more likely to uncover errors than testing with values from the same class

White Box Testing

- focuses on internal states of objects and code, trying to cover all code paths/statements
- requires internal knowledge of the component (very hard!!)
  - ➢Control-flow-based testing
  - ➢Data-flow-based testing

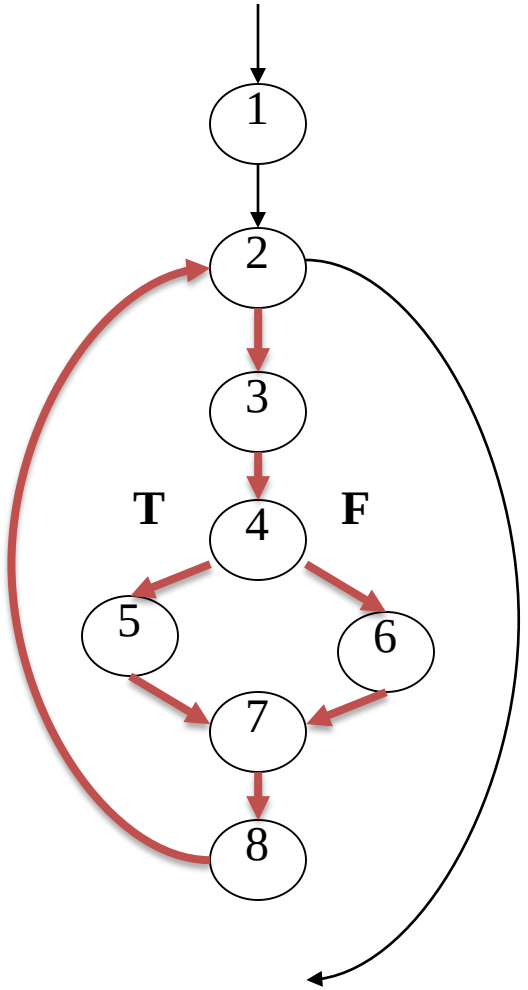
Statement Coverage

- Traditional target: statement coverage
  - Need to write test cases that cover all nodes in the control flow graph
- Intuition: code that has never been executed during testing may contain errors
  - Often this is the "low-probability" code



Branch Coverage

- Target: write test cases that cover all branches of predicate nodes
  - True and false branches of each IF
  - The two branches corresponding to the condition of a loop
  - All alternatives in a SWITCH statement



Question

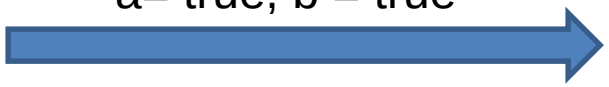
Does 100% statement coverage imply 100% branch coverage?

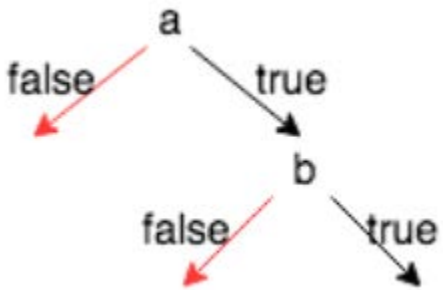
Answer

No!

```
if(a){
   if(b){
     bool statement1 = true;
   }
}
```

a= true, b = true



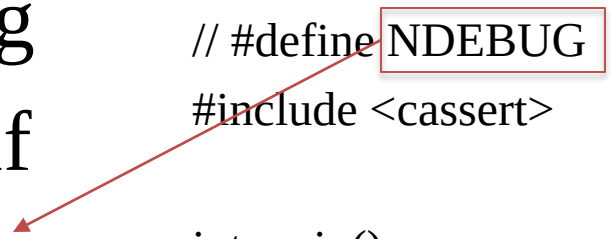


Back to our class/project

- **Assertion**
  - ➢Used to test a class or function by making an assertion about its
  - ➢behavior
  - ➢Fatal if assertion not met
- Syntax
  - ➢ *void assert (int expression);*
  - ➢*Expression to be evaluated. If this expression evaluates to 0, this causes an assertion failure that terminates the program*

Example

- If NDEBUG
- defined, assert does nothing
- not defined, assert checks if its argument (which must have scalar type) compares equal to zero



```
#include <iostream>
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>

int main()
{
    assert(2+2==4);
    std::cout << "Execution continues past the first assert\n";
    assert(2+2==5);
    std::cout << "Execution continues past the second assert\n";
}
```