

# CS 4320 / 7320

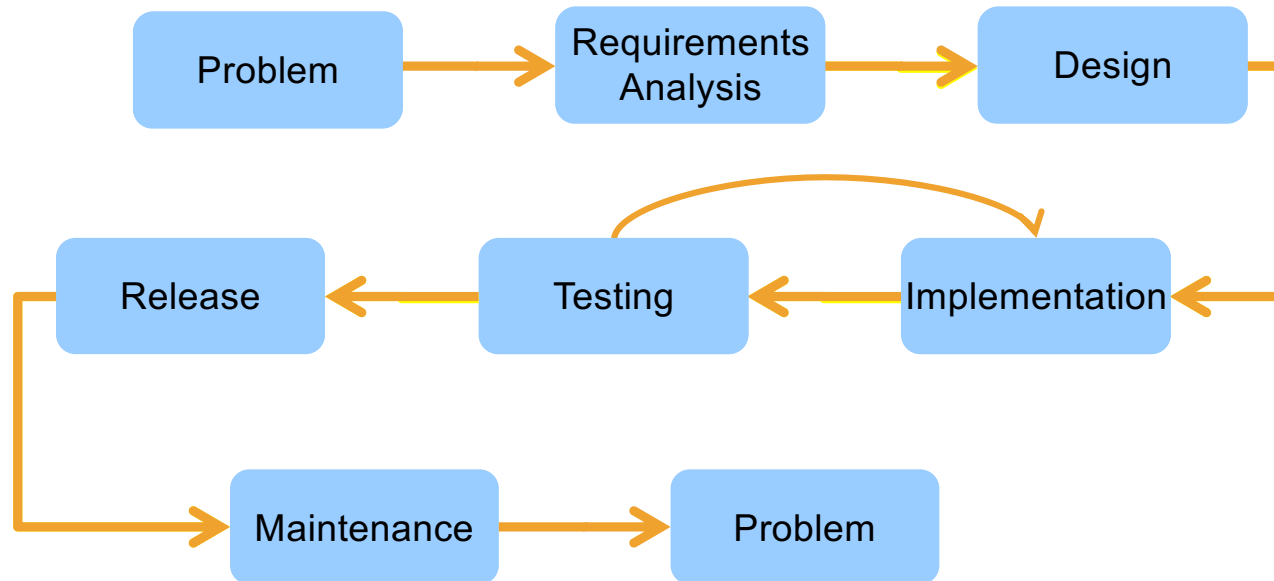
## Software Engineering

Module 7 –

*Construction and Testing*

# What is the SDLC?

## Where does Testing fit?



# Testing is....

... **dynamic** verification (or validation) that a program provides **expected** behaviors on a **finite** set of test cases, suitably **selected** from the (usually) infinite execution domain.

Software is designed to produce desired output from a set of inputs, based on the instructions *you think you have implemented*



The Test Target can vary in granularity:  
a unit, a combinations of components,  
a sub-system, the whole system

# Testing Goals

- Demonstrate the software meets the requirements
- Discover failures in the software
- *Faults cause failures*
- *Testing **reveals** failures*
- *The cause, the fault, may be hard to find*

# Test Levels

- Unit Testing
- Integration Testing
- System Testing

# Unit Testing: What is it?

Testing program components at the **smallest functional** unit

Unit tests are organized into **test suites** of related tests, for example...

For OOP, a class should have its own test suite(s) that exhaustively cover ***every method***, with **multiple** tests!

In MVC design, models and controllers can be unit tested.

DB stored procedures can be unit tested.

# Unit Testing: How to do it?

Units must be de-coupled to do unit testing.

A **driver** is a program that feeds test data to the test target and prints results.

A **stub** replaces modules that are invoked by the component to be tested, substituting dummy results for the test target to use

Good **interfaces** are essential

Units with **High Cohesion** are easier to test. (*Why?*)



# Test Driven Development

1. Identify new functionality needed in software / component
2. Write tests for new functionality
  - Functionality tests, unit tests
3. Run tests, which should ALL FAIL
4. Implement functionality in proper components and re-factor as necessary
5. Repeat step 3 and 4 until all tests pass

# Integration Testing

Testing if program components integrate & interact as expected

- Stubs are replaced with actual components

- Can include UI events driving behavior, database calls

These tests **begin** to represent test-cases of the **requirements**

As software/system complexity increases, more opportunities for components to interact

Each interaction between components may affect the state of the system or components

# System Testing

All components of the system are functioning together

For testing:

- Non-functional requirements – security, speed, accuracy, reliability.
- System configurations
- External interfaces

# Oops, we failed a test....

What do you do?

If coding is “done” - create an issue/bug

*Is it a design or code issue?*

*Fix the design and/or fix the code*

Then what?

**Regression testing**

*If tests pass.. close the issue/bug*

# When are we done with testing?

Does it do what it is supposed to do?

Who is the ultimate authority on this?

*Client product: client representative(s)*

*Consumer product: selected target user(s)*

## User Acceptance Testing

Not automated, typically scenario or story-board driven

# Validation vs Verification

Verification: Does it meet the detailed specifications?

Validation: Does it do what it is supposed to do?

*If it meets specifications, how could it not do what it is supposed to do?*





# Validation vs Verification

Which test(s) are validation?

Which test(s) are verification?

Test levels and objectives we've covered:

*Unit tests, integration tests, system tests,  
regression tests, user acceptance tests*



# Other testing objectives

Performance testing

Security testing\*\*

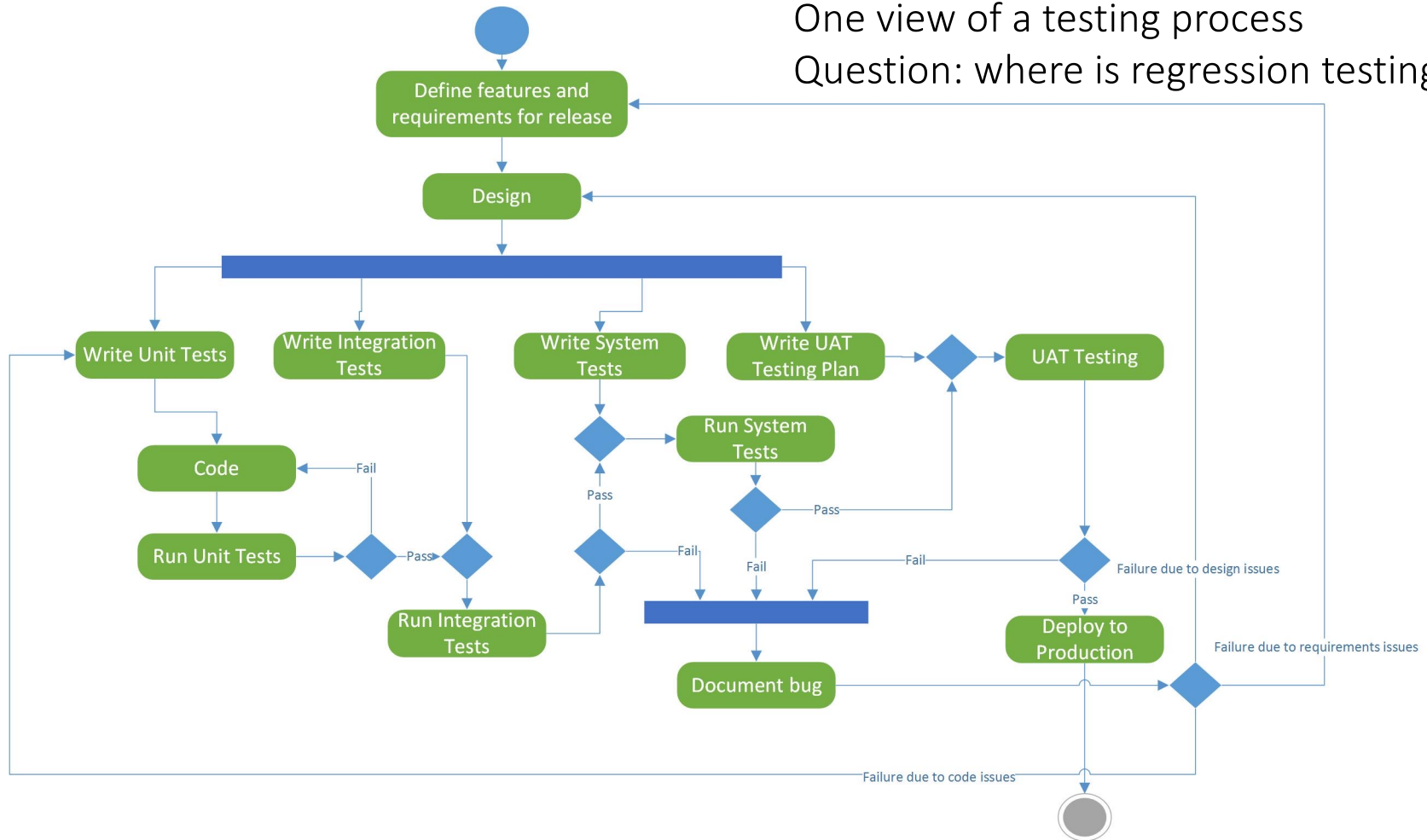
Stress testing

Interface testing

Configuration testing

Usability and Human Computer Interaction Testing

One view of a testing process  
Question: where is regression testing?



# Limitations of tests

Program testing can be used to show the *presence* of bugs, but never to show their *absence*.

Edsger Dijkstra

Testing is dynamic verification that a program provides expected behaviors on a *finite* set of test cases, suitably *selected* from the (usually) *infinite* execution domain.

SWEBOK Chapter 4

# Designing Tests: Black Box vs. White Box

Black Box Testing:

Tests are designed based on input / output only

White Box (or Clear Box) Testing:

Tests are designed based on the software design and coding

# Designing Tests: Input-based techniques

## **Equivalence Partitioning:**

Divide the input into logical subsets and take a sample from each subset

## **Pairwise Testing:**

Combine interesting values of input variables (rather than all combinations)

## **Boundary-Value Analysis:**

Choose input values on or near boundaries of the variable domains. Also choose values outside the boundaries.

## **Random Testing:**

Random input values from the domain



**Bill Sempf**  
@sempf

Follow

QA Engineer walks into a bar. Orders a beer.  
Orders 0 beers. Orders 999999999 beers.  
Orders a lizard. Orders -1 beers. Orders a  
sfdeljknesv.

RETWEETS  
29,009

LIKES  
19,494



12:56 PM - 23 Sep 2014



**meanporridge** @meanporridge · 15 Aug 2015

@sempf @SirStendec And nobody thought to order NULL beers, and that's what finally crashes the bartender in production.

2 12 28



**Bill Sempf** @sempf · 15 Aug 2015

@meanporridge @SirStendec I only had 140 characters...

2 6

# Designing Tests: Code-based techniques

## Control flow-based criteria:

Cover all statements, or statement blocks, or some combinations of statements.

Path testing: test all entry-to-exit control flow paths

*All* is usually not feasible – limit loops, aim for a % coverage

# Designing Tests: Code-based techniques

## Data flow-based criteria:

Select paths to explore the sequence of events related to variable status.  
When do variables receive values? When are these values used?

Helps find:

- *Variables that are declared but never used*
- *Variables that are used but never declared*
- *Variables that are defined multiple times before being used*
- *Variables that are deallocated before being used*



## Designing Tests: Test the unhappy path

Don't just think about what the code is supposed to do, but the error and exception handling that should happen if something goes wrong.

# Automated Testing

## **Advantages of automated testing**

Once tests have been developed, can be run quickly and repeatedly

Allows easy regression testing

Easy -> testing more likely to get done -> defects more likely to be found

## **Kinds of automated testing**

Graphical User Interface testing – a testing framework records or generates UI events

API driven testing –tests using a programming interface to the application (whole application, classes, modules, or libraries) , might use a framework

# What is “Done”?

*Done:*

With coding... but what if a test fails?

*Done, Done:*

With coding and testing.. But is it ready to be used?

*Done, Done, **Done**:*

With code, testing, and deployment... In production! **DONE!**