Agenda

. Design document (SDD) is due on 11/06/15 as per SacCT.

. Proposed final presentations (formal one) on Nov 16-Dec 1.

. Proposed Final report including SRS + SDD + Testing + Final Remarks due on Dec 4.

. Final Date: To be announced.

. This week: Software Testing (Verification and Validation)

CSc 131 – Computer Software Engineering

Software Testing (Verification and Validation)

Acknowledgements

- Roger Pressman: "Software Engineering: A Practitioner's Approach", ISBN-10: 0073375977

Additional references:

- Ian Sommerville: "Software Engineering", ISBN-10: 0137035152

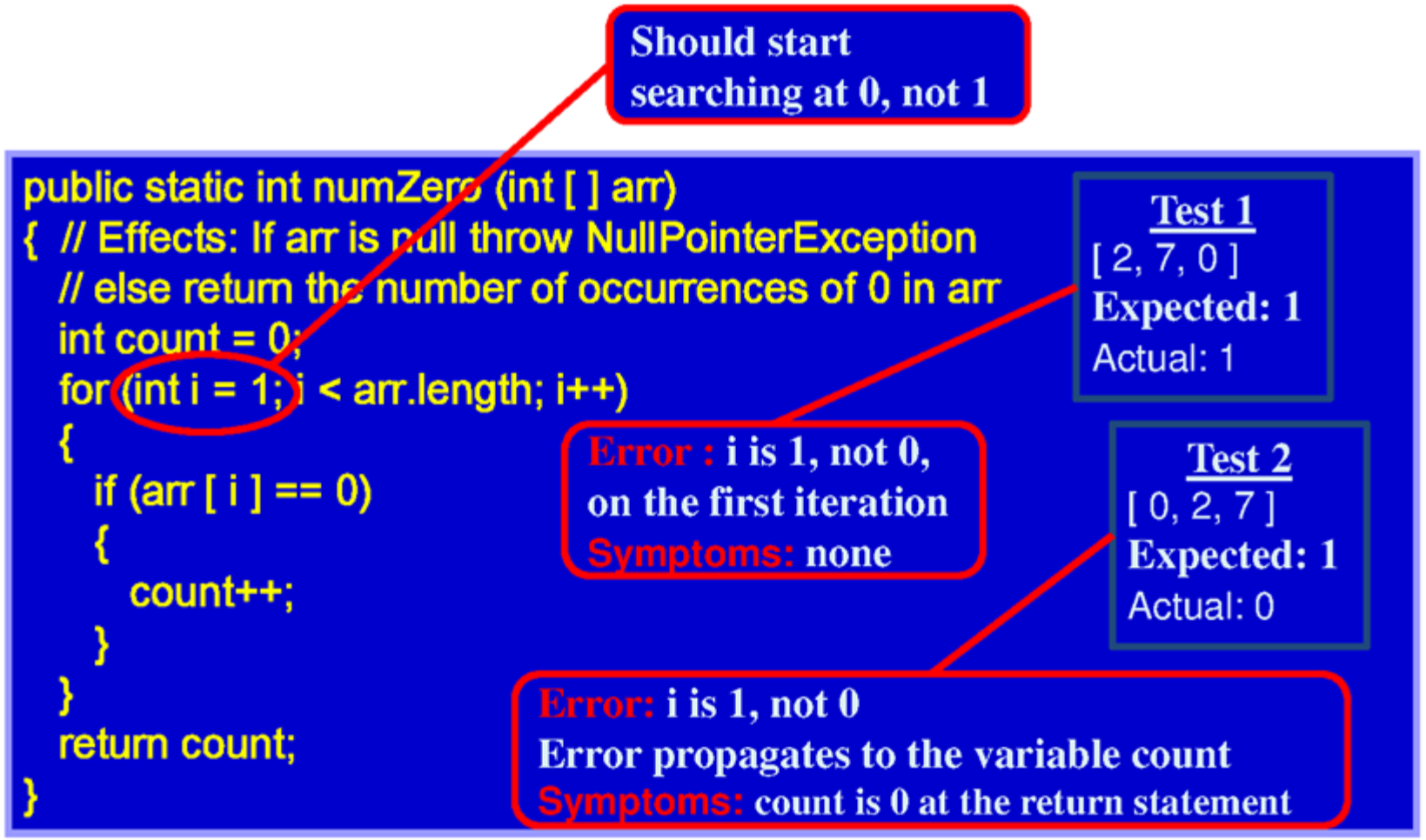
- Some materials were adapted from my previous CSC 230 course taught in Spring 2015

Software Testing (Verification and Validation)

Verification and Validation Outline:

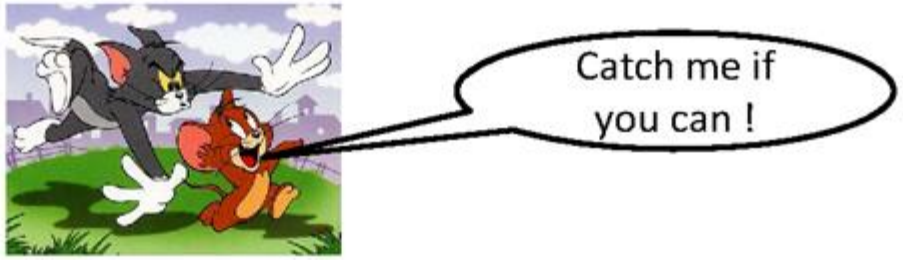
- Introduction & Motivations
- Part 1: Concepts
- Part 2: Testing Process
- Part 3: Deriving test cases

Example (similar CSC20 student's lab)



Techniques

Some “Bug detection” Techniques



- **Formal methods** :proving software correct

- **Testing**: **executing program** in a controlled environment (**input**) and **“validating” output** (IEEE definition).

Financial Impact of Software Errors

Recent research at Cambridge University (2013, link) showed that the global cost of software bugs is

around 312 billion of dollars annually

Goal: to increase software reliability

How to identify software bugs?

How do we know behavior is a bug?

Because we have some separate specification of what the program must do

- Separate from the code

Thus, knowing whether the code works requires us first to define what “works” means

- A specification

Not Testing ?

Cost of Not Testing

Poor Program Managers might say:
"Testing is too expensive."

- Testing is the most time consuming and expensive part of software development
- Not testing is even more expensive
- If we do not have enough testing effort early, the cost of testing increases

Testing Goals

- **The Major Objectives of Software Testing:**
  - Detect errors (or bugs) as much as possible in a given timeline.
  - Demonstrate a given software product matching its requirement specifications.
  - Validate the quality of a software testing using the minimum cost and efforts.
- **Testing can NOT prove product works 100%- -**
  - even though we use testing to demonstrate that parts of the software works

CSc 131 – Computer Software Engineering

Concepts

Part 1: Concepts

- Verification and Validation
- Static and dynamic verification
- Failure, Fault, Test case, Testing
- Black-box and white-box testing
- Test stubs and drivers

Verification and Validation

- **Verification**: - The software should conform to its specification (the functional and non-functional requirements).

Concerning: "Are we building the product right".

- **Validation**: - The software should do what the customer really requires.

Concerning: "Are we building the right product".

Verification Techniques

- Static verification

- Inspections, reviews

- Analyze static system representations to discover problems

- Applies to: specifications, design models, source code, test plans

- Dynamic verification

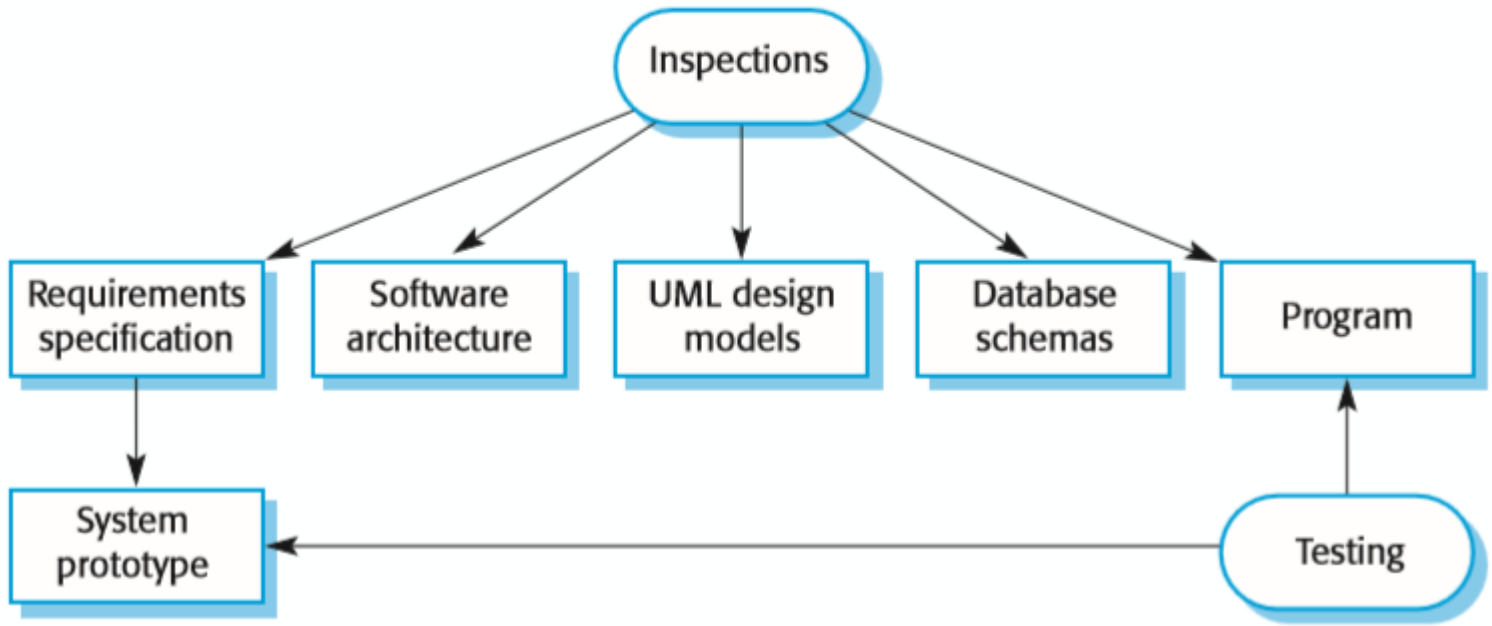
- Testing

- The system is executed with simulated test data

- Check results for errors, anomalies, and data regarding non-functional requirements.

Verification Techniques

Verification at different stages in the software process



Testing Concepts

- Failure - Deviation between the specification and the actual behavior of the system.

- Fault (aka “bug” or “defect”) - A design or coding mistake that may cause abnormal behavior (with respect to specifications)

- Test case - set of inputs and expected results that exercises a system (or part) with the purpose of detecting faults

- Testing - the systematic attempt to find faults in a planned way in the implemented software.

Test cases

Test cases should contain the following:

- Name - Explains what is being tested
- Input - Set of input data and/or commands and/or actions
- Expected results - Output or state or behavior that is correct for the given input.

Test cases: example only

TC#	Proj.Fun.-010	UC flow	2.2.2 main success scenario (Basic, alternative, exception flow name or function under test)
Objectives	**Try to use:** -Verify that (for TC with valid data) -Attempt to (for TC with invalid data)		
Preconditions	**Input**	**Expected Results**	
-The system displays… -User has successfully… -The system allows… -The user has been authenticated…	**(For different conditions where applicable)** -The user selects… -The user enters…	-Expected result may be copy-paste from Use Case but it depends on how the Use Case is written.	

Source: http://extremesoftwaretesting.com/Testing/TCtemplate.html

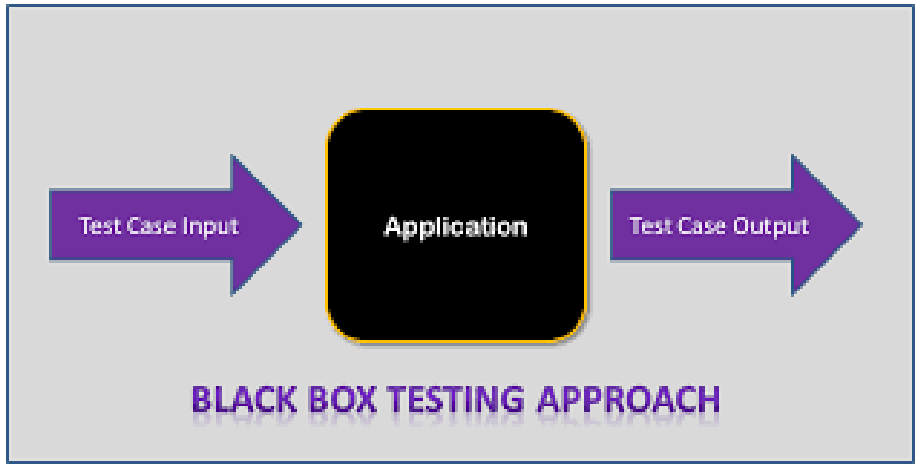
Black-box and White-box testing

Different kinds of test cases:

- Black-box tests

- focus on input/output behavior of the software

- are not based on how the software is structured or implemented.



Black-box and White-box testing

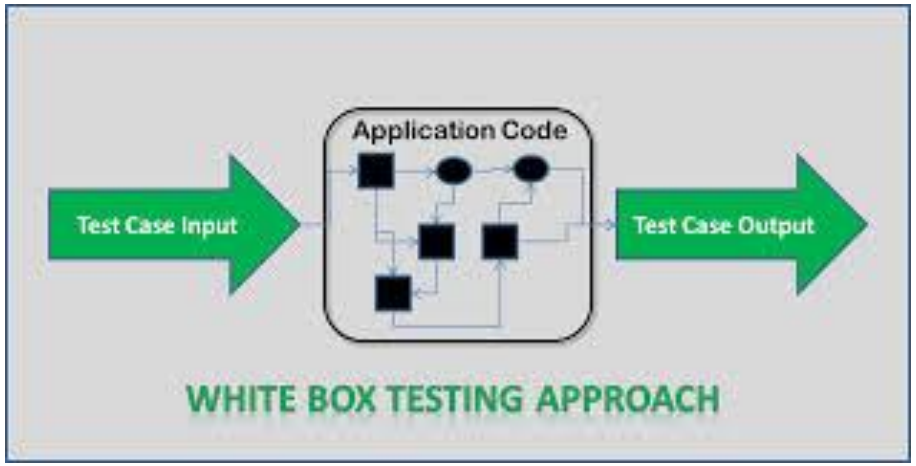
Different kinds of test cases:

White-box tests:

- focus on the internal structure of the software

- an internal perspective of the system is used to design test cases.

- goal: test all parts of code in the software



Test stubs and drivers

How to test units/components in **isolation**:

• test driver

- code that simulates the part of the system that calls the component under test.

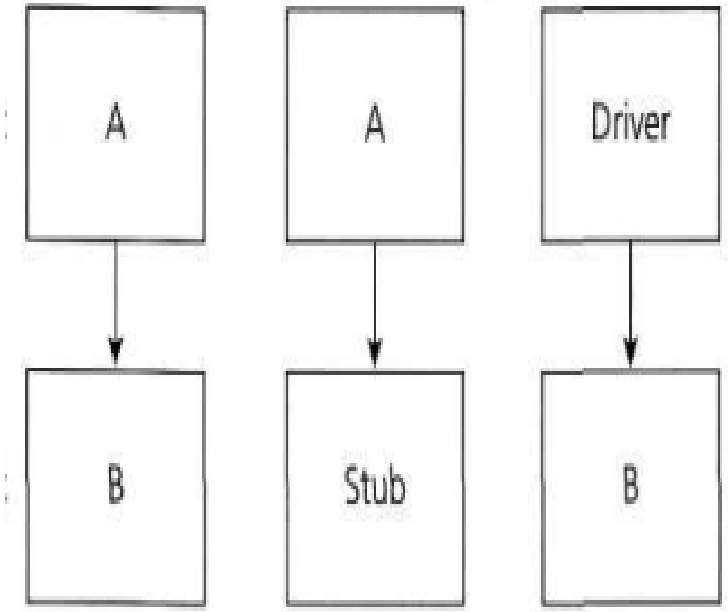
- often provides the input for a given test case

- this code is in a function that is executed during the test

• test stub

- code that simulates a component that is called by the tested component

- must support the called component's interface, and return a value of the appropriate type.



CSc 131 – Computer Software Engineering

Testing Process

Part 2: Testing Process

- Development Testing
  - Unit testing
  - Component testing
  - System testing

- Release Testing

- User Testing
  - Alpha testing
  - Beta testing
  - Acceptance testing

Testing Process

For each kind of testing activity we group them based on:

- Who performs the tests?

- Developers, independent testing team, users+customers

- What are the constraints of the tests?

- test a certain part of the system?

- test the system at a certain point in the process?

Software testing activities: Who performs the test?

- Development testing:

- developers test the system during development

- Release testing:

- a separate testing team tests a complete version of the system before it is released to users.

- User testing:

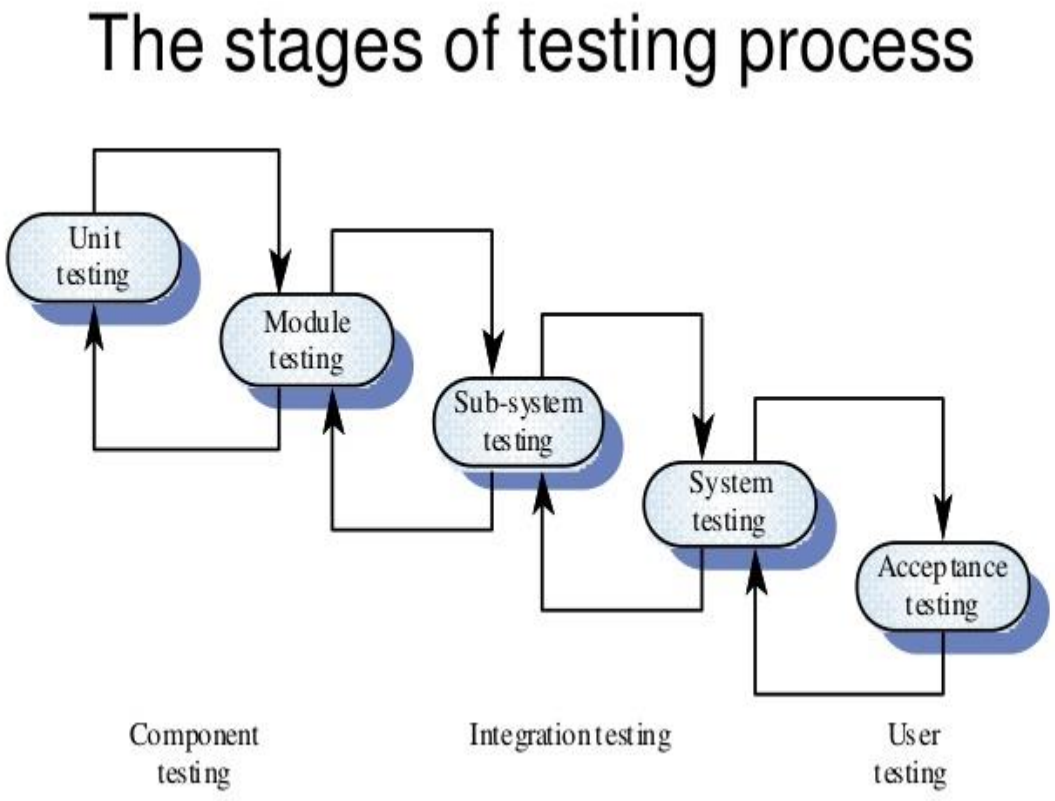
- Customers and users (or potential users) of a system test the system in their own environment.

Development testing
Which parts are tested?

- Unit testing - individual program units (i.e. classes) are tested

- Component testing - system components (composed of individual units) are tested to make sure the contained units interact correctly.

- System testing - the system components are integrated and the system is tested as a whole.



Unit Testing

Tests that are performed during the unit testing are below:

1) **Module Interface test**: In module interface test, it is checked whether the information is properly flowing into the program unit (or module) and properly happen out of it or not.

2) **Local data structures**: These are tested to inquiry if the local data within the module is stored properly or not.

3) **Boundary conditions:** It is observed that much software often fails at boundary related conditions. That's why boundary related conditions are always tested to make safe that the program is properly working at its boundary condition's.

4) **Independent paths**: All independent paths are tested to see that they are properly executing their task and terminating at the end of the program.

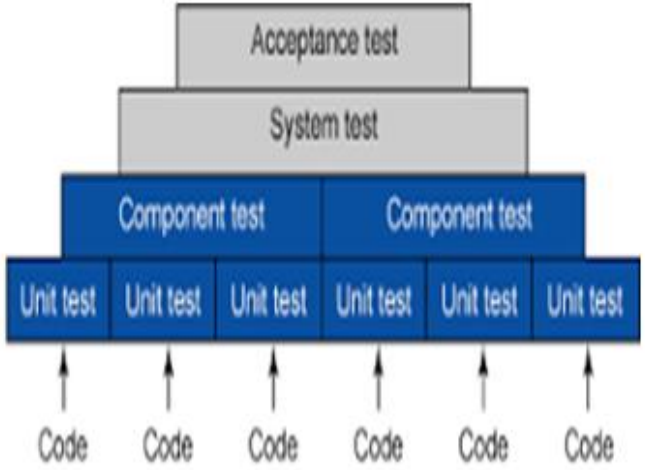
5) **Error handling paths:** These are tested to review if errors are handled properly by them or not.

Component testing

• Component testing

- System components (composed of individual units) are tested to make sure the units interact correctly.

- The functionality of these objects is accessed through the defined component interface.

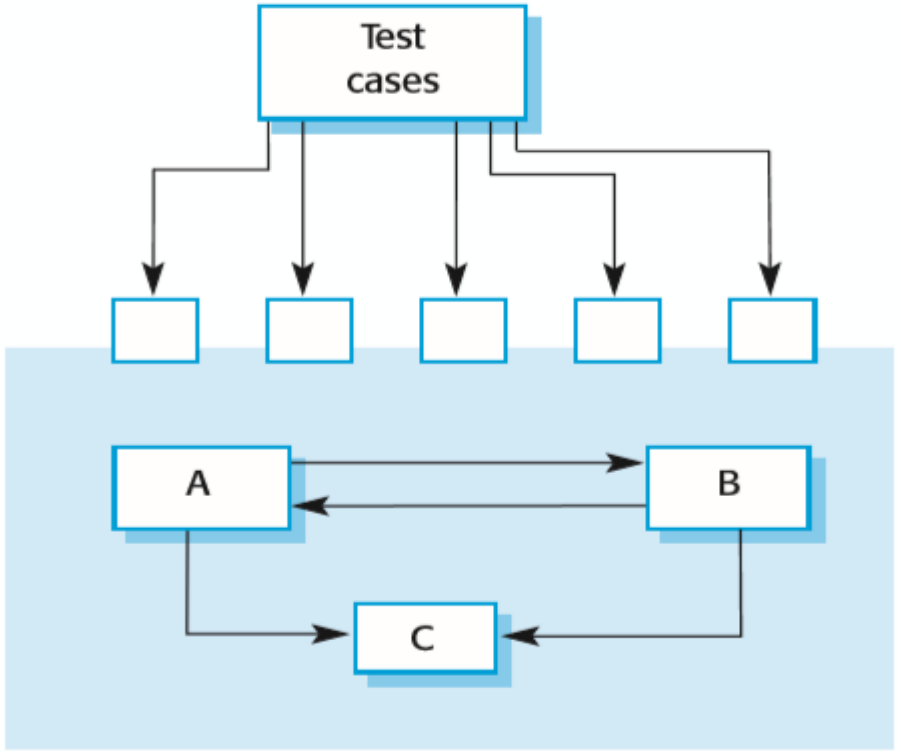


• Component testing is demonstrating that the component interface behaves according to its specification.

- Assuming the subcomponents (objects) have already been unit-tested

Component testing (cont)

Interface (component) testing



Small empty boxes represent the interface

System testing

- System testing

- the components in a system are integrated and the system is tested as a whole.

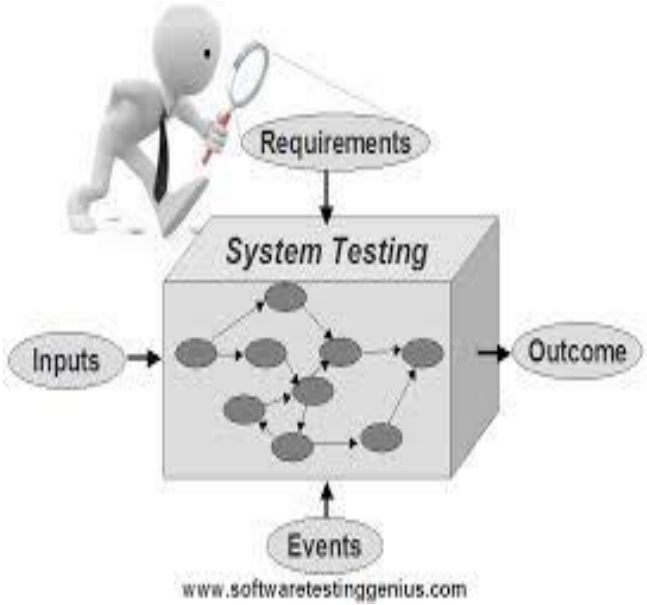
- Checks that:

- components are compatible,

- components interact correctly

- components transfer the right data at the right time across their interfaces.

- Tests the interactions between components.



Release testing

- Release Testing

- testing a particular release of a system that is intended for use outside of the development team.

- Similar to system testing by developers, but

- Tested by a team other than developers.

- Focus is on demonstrating system meets requirements.

- Primary goal: convince the supplier of the system that it is good enough for use.

- Black-box testing process where tests are derived from the system specification.

User testing

- User testing:

- Customers and users (or potential users) of a system test the system in their own environment.

- Essential even when comprehensive system and release testing have been carried out.

- Influences from the user’s working environment have a major effect on the reliability, performance, usability and robustness of a system.

- These cannot be replicated in a testing environment

Types of User testing

- Alpha testing

- Users of the software work with the development team to test the software at the developer's site.

- generic or custom software

- Beta testing

- A release of the software is made available to users to allow them to experiment and to report problems

- generic or custom software

- Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers.

- acceptance implies payment is due, may require negotiation.

- custom software.

CSc 131 – Computer Software Engineering

Deriving Test Cases

Deriving Test Cases

- Unit Testing
  - Partition testing (Equivalence Class Partitioning)
  - Boundary value analysis
  - Path testing (Path Analysis)
  - Guideline-based testing

- System + Release Testing
  - Use case-based testing
  - Scenario testing
  - Requirements-based testing

Unit testing: How are test cases developed?

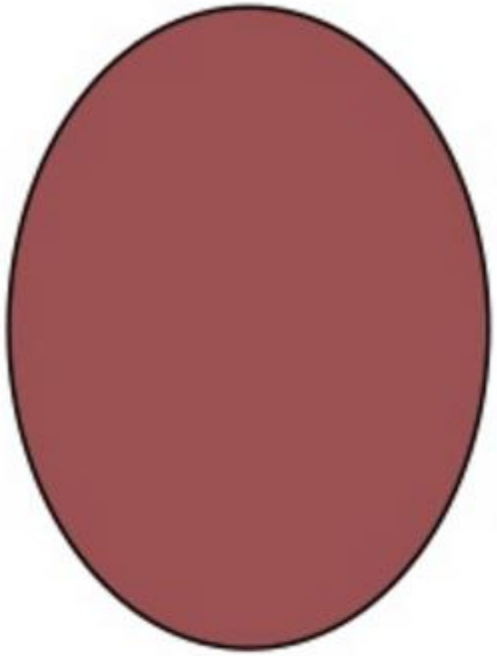
- Partition (or equivalence) testing: - identify groups of inputs that have common characteristics and should be processed the same way by the system, use one test case per group.
- Boundary value analysis - test the boundaries of the groups used in partition testing
- Path testing - exercise all possible paths through the code at least once
- State-based testing - define sequences of events to force all possible transitions.
- Guideline-based testing - use guidelines that reflect the kinds of errors programmers often make

Partition Testing

- Divide the set of all possible input data of a software unit into partitions
  - What's a partition?

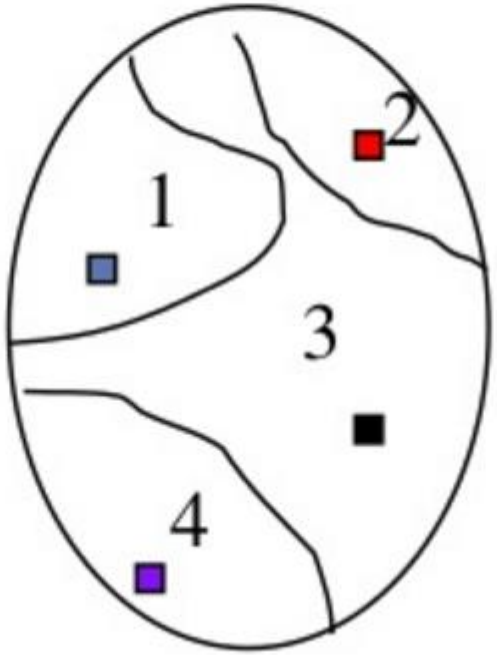
    A partition of a set X is a set of nonempty subsets of X such that every element x in X is in exactly one of these subsets
  - program should behave similarly for all data in a given partition
  - Determine partitions from specifications
- Design **one** test case for each partition, using sample input data from the given partition.
- Enables good test coverage with fewer test cases.

Partition Testing (cont)



Input domain

Too many test inputs.



Input domain partitioned into four sub-domains.

Four test inputs, one selected from each sub-domain.

Partition Testing (cont)

Partition testing: example

Function returning the number of days in a month:

```
int getNumDaysInMonth(int month, int year);
```

Partition	month value	year value	expected result
Month with 31 days, non-leap years	7 (July)	1901	
Months with 31 days, leap years	7 (July)	1904	
Months with 30 days, non-leap years	6 (June)	1901	
Month with 30 days, leap year	6 (June)	1904	
Month with 28 or 29 days, non-leap year	2 (February)	1901	
Month with 28 or 29 days, leap year	2 (February)	1904	

Boundary Value Analysis

Complements equivalence partitioning (typically combined)

In practice, more errors found at *boundaries* of equivalence classes than within the classes

Divide input domain into equivalence classes

Also divide output domain into equivalence classes

Need to determine inputs to cover each output equivalence class

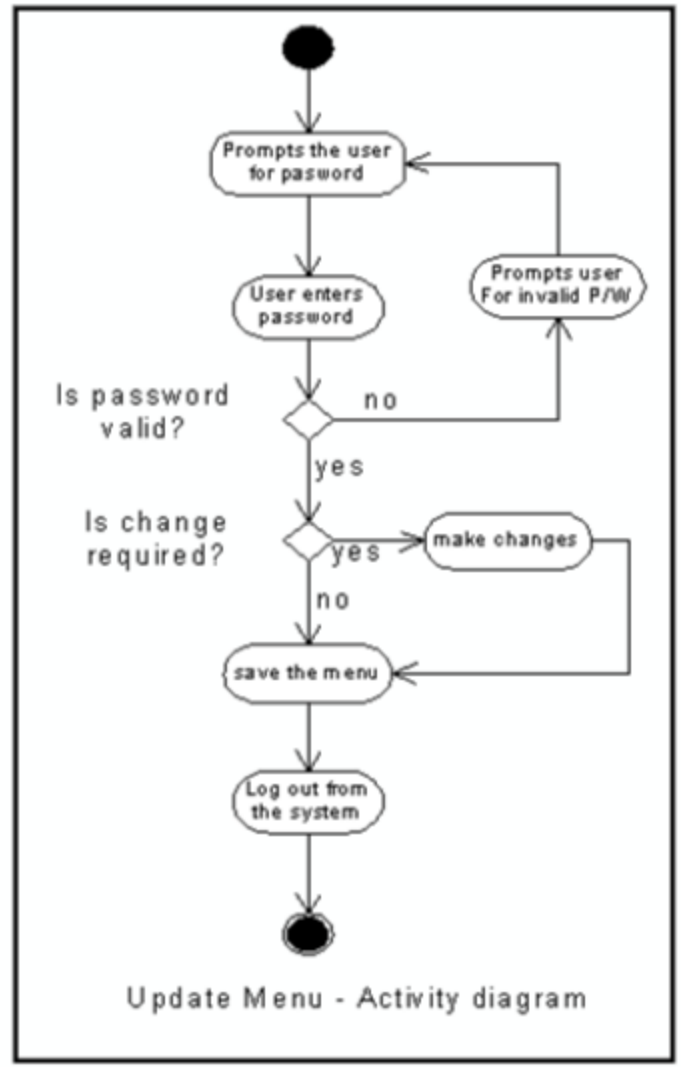
Again one test case per equivalence class

Path testing

- Exercise all possible paths through the code at least once
  - a white-box testing technique
  - convert code to control-flow diagram
  - choose input data so that each path through diagram is executed

Example: Make sure that at least one test case forces each oval to execute:

- one with valid password that requires no change
- one with invalid password, then valid password that requires a change



Guideline-based testing

- Choose test cases based on previous experience of common programming errors
- For example:

- Choose inputs that force the system to generate all error messages

- Repeat the same input or series of inputs numerous times

- Try to force invalid outputs to be generated

- Force computation results to be too large or too small.

System and Release testing: How are test cases developed?

- Use case-based testing: - use the use-cases developed during requirements engineering to develop test cases.
- Scenario testing - use scenarios (user stories) developed during requirements engineering to develop test cases.
- Requirements-based testing - examine each requirement in the SRS and develop one or more tests for it.

System and Release testing: How are test cases developed?

- Use case-based testing: - use the use-cases developed during requirements engineering to develop test cases.
- Scenario testing - use scenarios (user stories) developed during requirements engineering to develop test cases.
- Requirements-based testing - examine each requirement in the SRS and develop one or more tests for it.

System and Release testing: How are test cases developed?

- Use case-based testing: - use the use-cases developed during requirements engineering to develop test cases.
- Scenario testing - use scenarios (user stories) developed during requirements engineering to develop test cases.
- Requirements-based testing - examine each requirement in the SRS and develop one or more tests for it.

Use case-based testing: example

Use case name	PurchaseTicket
Entry condition	The Passenger is standing in front of ticket Distributor. The Passenger has sufficient money to purchase ticket.
Flow of events	1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last button pressed is considered by the Distributor. 2. The Distributor displays the amount due. 3. The Passenger inserts money. 4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger. 5. If the passenger inserted more money than the amount due, the Distributor returns excess change. 6. The Distributor issues tickets. 7. The Passenger picks up the change and the ticket.
Exit condition	The Passenger has the selected ticket.

Future of Software Testing

1. Increased specialization in testing teams will lead to more efficient and effective testing
2. Testing and QA teams will have more technical expertise
3. Developers will have more knowledge about testing and motivation to test better
4. Agile processes put testing first—putting pressure on both testers and developers to test better
5. Testing and security are starting to merge
6. We will develop new ways to test connections within software-based systems

Final Remarks

1. Focus on normal test cases but do NOT ignore boundary condition.
   - Sometime, boundary condition failure could turn out to be very fatal.

2. Play defensive role (like driving) when designing/coding.

3. If possible, get your users involve in early testing and provide feedbacks to test cases construction.

4. Smart testing