COMP642

Object Oriented Programming

Lectorial 10 - Testing and Debugging

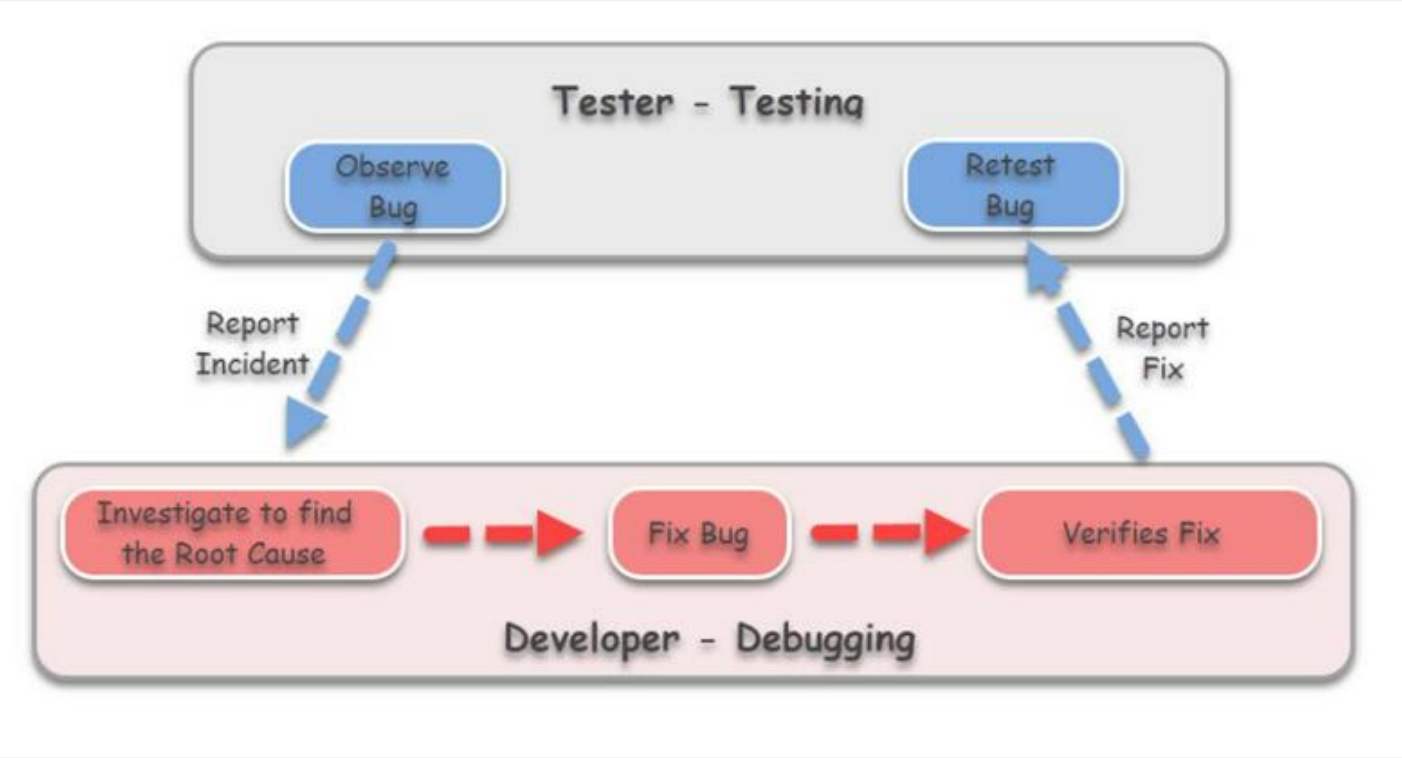
Testing

- A process to verify whether the system is working in the same way expected to ensure that the software is defect free.
- Identifies errors or bugs in the system
- Aims at finding potential problems

Debugging

- The process of finding and resolving defects that prevent correct operation of computer software.
- A manual process to find and eliminate specific system errors
- Aims at solving the problem

Debugging vs. Testing



- Testing is the process to find bugs and errors.
- Debugging is the process to correct the bugs found during testing.

Debugging (1)

- Multistep process that involves:
  - identifying a problem
  - isolating the source of the problem
  - correcting the problem or determining a work around
- Use stand-alone debugger tool or debug mode of an integrated development environment (IDE)

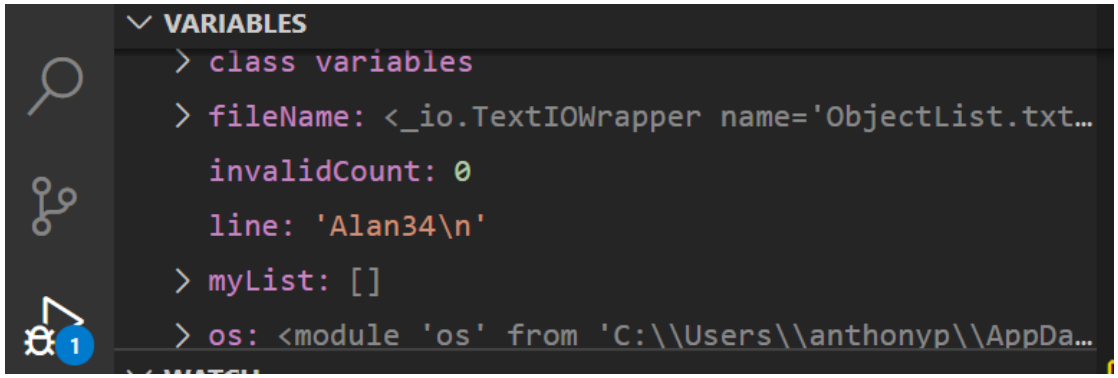
Debugging (2)

- Standard practice is to set up “breakpoint”

```
37  fileName = open("ObjectList.txt", "r")
38  for line in fileName:
39      data = line.split(',')
40      name = data[0]
41      try:
42          age = int(data[1])
43          aStudent = Student(name, age)
44          validCount += 1
```

Run the program to the next breakpoint

Can view the memory and see variables



Common Coding Errors

- Syntax errors
  - Occur when the code does not follow the syntax rules of the programming language
  - Examples: misspellings, missing brackets, incorrect indentation
- Runtime errors
  - Happen during the execution of the program and cause it to crash
  - Examples: division by zero, invalid type conversions, file not found
- Logic errors
  - Program runs without crashing but produces incorrect results
  - Hardest to identify and debug
  - Examples: off-by-one errors, incorrect conditions in loops, wrong calculations

Debugging Techniques

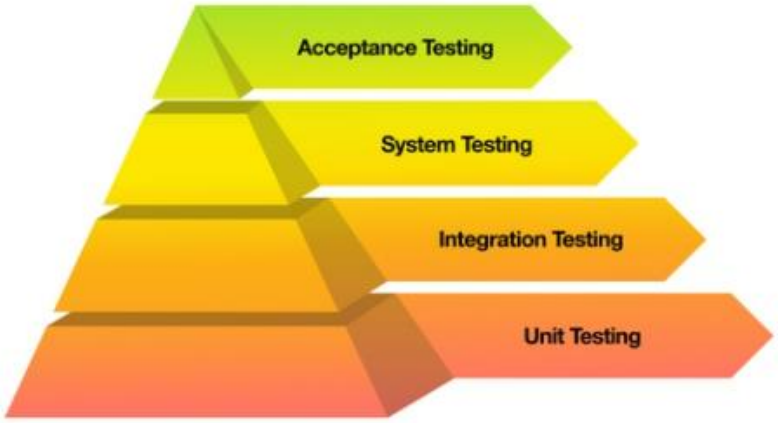
- Easiest and most effective way to start debugging is just to print things out
- Use flags: Flags are variables that indicate whether a specific section of code was executed or if a particular event has occurred.
- Comment out sections of code
- Desk checking: Manually walk through the code while tracking variable values
- Peer checking: Have someone else review your code
- Take a break and revisit your code later

Good Programming Practice

- Readability
  - Use meaningful names for variables, constants, functions, and classes
  - Follow naming conventions
  - Apply proper formatting and consistent indentation
- Maintainability
  - Keep high-level functions clean and uncluttered
  - Order functions in a logical sequence
  - Avoid deep nesting
  - Use comments to explain the code
- Extendability
  - Avoid using global variables whenever possible
  - Follow the DRY principle (Don’t Repeat Yourself)
  - Write reusable functions
  - Return values from functions instead of using print statements

Unit Testing

- Individual units or components of a software are tested.
- Purpose is to validate that each unit of the software code performs as expected.
- Done during the development of an application by the developers
- Isolate a section of code (method or object) and verify its correctness



Why Unit Testing?

- Reduces bugs in new features and existing features
  - Catches small bugs early, preventing larger issues later
- Provides good documentation
  - Clarifies functionality and usage for each unit
- Reduces cost of change
  - Errors fixed early are cheaper than those found later
- Faster debugging
  - Isolates code sections for quick issue identification
- Contributes to faster development
  - Fewer bugs allow for a focus on new features
- Promotes better design
  - Encourages modular, maintainable code for robust architecture

How to do Unit Testing?

- Manual
  - List all application features and document input types and expected results
  - Testers manually interact with each feature, input data, and check actual outcomes against expected results.
- Automated
  - Execution of the test plan using scripts instead of manual intervention
  - Developers write specific code in the application to test different functionalities, automatically comparing actual results to expected outcomes for each unit.
  - Enables more rigorous testing

Python Testing Frameworks

- unittest
  - Built into the Python standard library, it includes both a testing framework and a test runner.
- Nose2
  - An extension to unittest, Nose2 simplifies the creation and execution of test cases but is not part of the Python standard library.
- pytest
  - Supports execution of unittest test cases. pytest cases are defined as functions in a Python file, with names starting with **test_**.

pytest (1)

- Python-based testing framework for writing and executing test codes
- Supports the use of the built-in assert statement
- Enables filtering to select specific test cases
- Allows skipping certain test cases as needed
- Provides options to run a subset of the entire test suite

pytest (2)

```python
def add(x, y=2):
    return x + y

def product(x, y=2):
    return x* y
```

The methods that need to be tested

```python
def test_add():
    assert math_func.add(7,3) == 10
    assert math_func.add(7) == 9
    assert math_func.add(5) == 7

def test_product():
    assert math_func.product(5,5) == 25
    assert math_func.product(5) == 10
    assert math_func.product(7) == 14
```

Test cases

```
PS S:\COMP\COMP642\Sem 2 2021\Python Code\Testing> pytest -v test_math_func.py
=================================== test session starts ====================
platform win32 -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-1.0.0 -- C:\Use
a\Local\Programs\Python\Python39\python.exe
cachedir: .pytest_cache
rootdir: S:\COMP\COMP642\Sem 2 2021\Python Code\Testing
collected 2 items

test_math_func.py::test_add PASSED
test_math_func.py::test_product PASSED
```

The output

Integration Testing

- Involves testing multiple components of the application to check that they work together
- Conducted after unit testing has been completed
- Helps uncover errors that may lie in the interfaces between modules
- Ensures that newly added components do not disrupt the functionality of existing components
- Can be performed by developers or testers

Why Integration Testing?

- Defects still exists:
  - Different programming logic used by different programmers developed different modules
  - Changes in user requirements may not have been addressed during unit testing.
  - Database issues
  - Interface issues
  - Inadequate exception handling

Approaches to Integration Testing

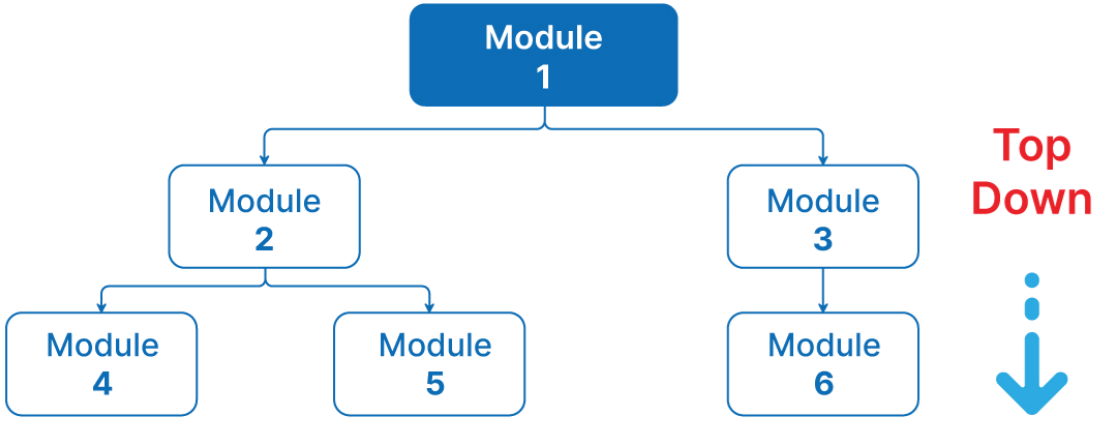
- Top-Down Approach: Testing starts from the top level and progresses downwards following the control flow.
- Bottom-Up Approach: Testing begins from the lower levels and moves upwards through the control flow.
- Big Bang Approach: All components or modules are integrated simultaneously, and then the entire system is tested as a whole.

Stubs and Drivers

- Dummy programs in integration testing used to facilitate the software testing activity
- Act as substitutes for the missing modules in the testing
- Do not implement the entire programming logic or the software module but they simulate data communication with the calling module while testing.
- Stub: called by module under test
- Driver: calls the module to be tested

Top-Down Approach

- Testing takes place from top to bottom following the control flow of software system
- Higher level modules are tested first, and then lower-level modules are tested and integrated.
- Stubs are used for testing if some modules are not ready



Advantages:

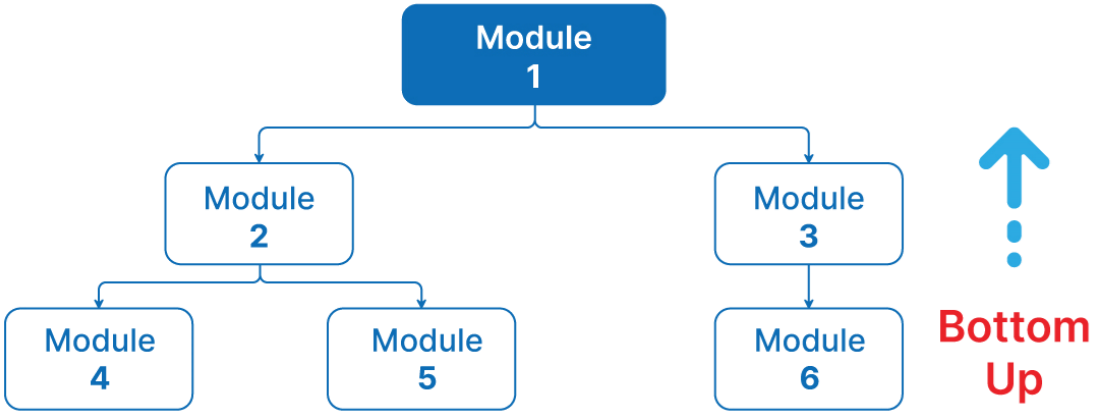
- Fault localisation is easier
- Less time required
- Detect major flaws

Disadvantages:

- Need many stubs
- Poor support for early release
- Basic functionality tested late

Bottom-Up Approach

- Lower modules are tested first.
- These tested modules are then further used to facilitate the testing of higher modules.
- Process continues until all the modules at top level are tested.
- Once lower modules are tested an integrated, the next level of modules are formed.
- Drivers are used for testing of some modules are not ready



Advantages:

- Test conditions are easier to create
- Less time required

Disadvantages:

- Requires several drivers
- Data flow is tested late
- Poor support for early release
- Key interface defects are detected late

Big Bang Approach

- All components or modules are integrated together once and then tested as a unit.
- If all the components in the unit are not completed, the integration process will not execute.

Advantages:

- All components are tested at once
- Convenient for small systems
- Saves testing time

Disadvantages:

- Lots of delays before testing
- Difficult to trace cause of failures
- Possibility for missing interface links
- Critical modules are not prioritised

Sandwich Approach

- Combination of top-down and bottom-up approaches (Hybrid Integration Testing)
- Top level modules are tested with lower-level modules at the same time
- Lower-level modules are integrated with top modules and tested as a system
- Makes use of both stubs and drivers

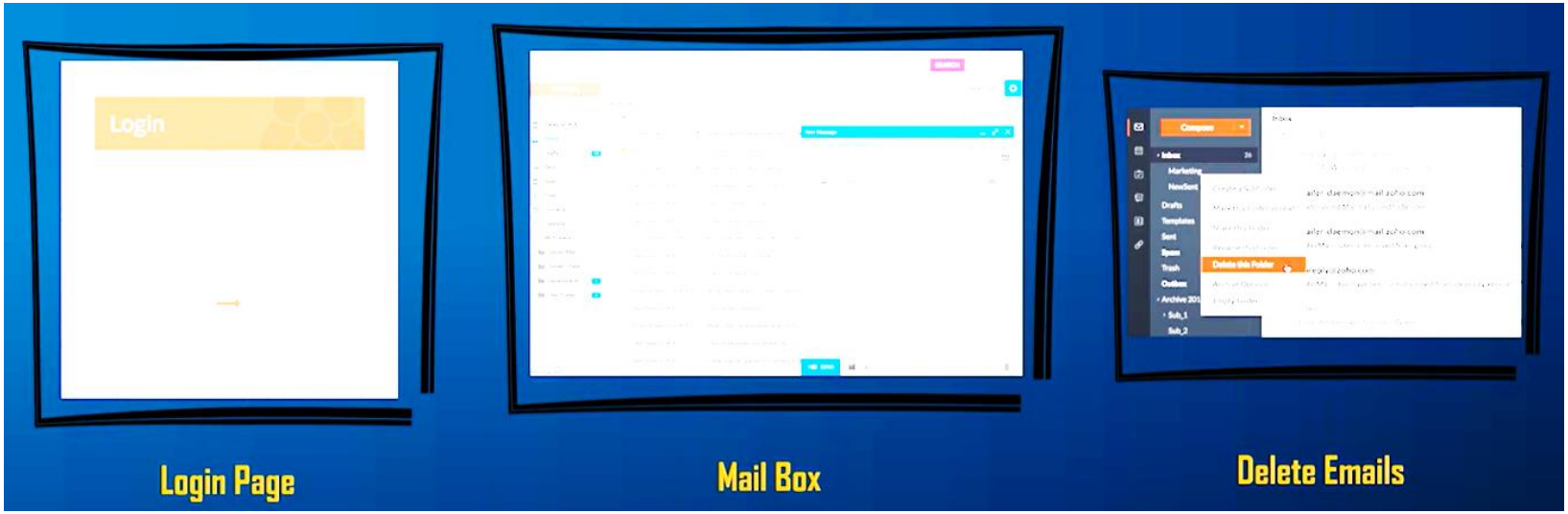
Advantages:

- Both layers can be tested in parallel

Disadvantages:

- High cost
- Big skill set needed
- Extensive testing is not done

Example of Integration Testing (1)



- Focuses mainly on the interfaces and flow of the data/information between the modules
- Priority is on integrating links rather than the unit functions.

Example of Integration Testing (2)

Test Case ID	Test Case Objective	Test Case Description	Expected Outcome
A	Test the interface link between Login Page and the Mail Box Page	Enter the login details & click on login button to login	You should be directed to the Mail Box Page
B	Check the interface link between Mail Box & the Delete Email Module	From mail box select the email you want to delete & click on delete	Selected email should be deleted and should appear in the Trash Folder