Integration and System Testing

Software Testing and Quality Assurance

Integration Testing – Drivers and Stubs

- Drivers and Stubs are temporary software components
- A test *driver* calls the software under test, passing the test data as inputs.
- In manual testing, where the system interface has not been completed, a test driver is used in its place to provide the interface between the test user and the software under test.

Definitions – Integration Tests

- Integration test data is selected to ensure that the components or sub-systems of a system are working correctly together.
- Test cases will explore different interactions between the components, and make sure the correct results are produced.

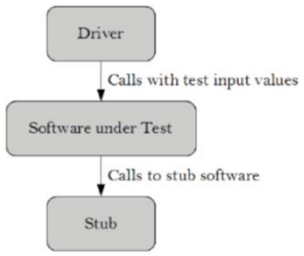
Drivers

- Drivers can have varying levels of sophistication.
- It could be hard-coded to run through a fixed series of input values, read data from a prepared file, contain a suitable random number generator etc..

Definitions – System Tests

- System test data is selected to ensure that the system as a whole is working.
- Test cases will therefore explore the different inputs and combinations of inputs to the system to ensure that the system satisfies its specification

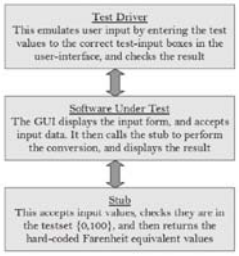
Drivers and Stubs - Diagram



Stubs

- A *stub* is a temporary or dummy software that is required by the software under test to operate properly.
- This is a throw-away version to allow testing to take place.
- It will provide a fixed or limited set of values to be passed to the software under test.

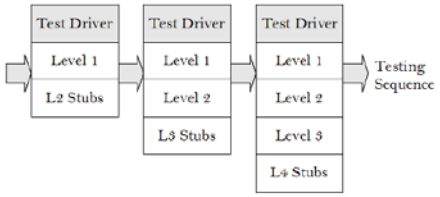
GUI – Celsius to Fahrenheit Example



GUI – Celsius to Fahrenheit Example

- This program converts temperature in Celsius to Fahrenheit.
- The driver sends input data, the temperature to be converted, via a windowing system to the user interface under test.
- The core functionality performs the conversion. However, it has not been coded so a stub is written to emulate this.

Top-Down Integration testing



GUI – Celsius to Fahrenheit Example

- The user interface calls the stub to do the conversion and display the result.
- The result is picked up by the driver and checked for correctness.
- Only two test case data values are defined (0 and 100 Celsius). The stub can only convert these values.

Top-Down Integration testing

- Testing moves from the top layer of the program to the bottom layers.
- Stubs are used to simulate modules being called by the modules under test.

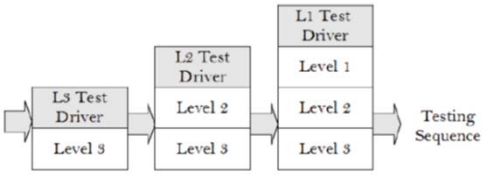
Top-Down Integration testing

- Strengths
  - An early outline of the overall program is available so design errors can be found early, giving confidence to the team and customer.
- Weaknesses
  - Proper Stubs are difficult to design.
  - If lower levels are still being created while upper level is complete, sensible changes that could be applied to the upper levels could be ignored.
  - On adding the lower layers, the upper layers will need to be tested again.

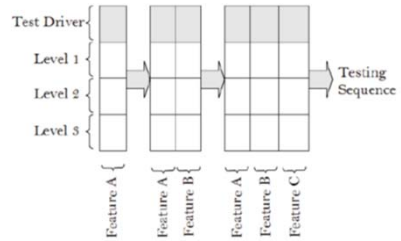
Sandwich Integration

- This is a hybrid of Top-down and Bottom-up.
- A target layer is defined somewhere in the middle of the program and testing converges here.
- The top and bottom layer can be tested in parallel and reduces the need for stubs and drivers.
- It is complex to plan and difficult to select the target layer.

Bottom-up Integration testing



End-to-end User functionality



Bottom-up Integration testing

- Strengths
  - Overcomes the disadvantages of top-down testing.
  - Drivers easier to produce then stubs.
  - Developers will have a better understanding of the lower layer modules by the time they reach the top, making them easier to test.
- Weaknesses
  - Harder to imagine the final, working system
  - Important user interaction modules only tested at the end.
  - Drivers need to be created

End-to-end User functionality

- The software is integrated from the bottom up but only small increments of user functionality are added across all the software layers.
- Overcomes problems with adding large amounts of functionality as in the Bottom-up approach.
- The end user can view and test the software as it is being developed, supporting intermediate changes more easily.
- This approach is associated with *Agile Test-driven* development.

Integration testing - Considerations

- Number of stubs and drivers needed.
- Location of the critical modules in the system.
- Which layers will be available for testing first.

System Testing - Categories

- **Ergonomic Testing** Used to verify the ease-of-use of the system. This can be either automated (verifying font sizes, information placement on the screen, use of colours, or the speed of progress by users of different experience levels), or manual (based on feedback forms completed by users).
- **Functional Testing** As for Unit Testing, this is used to verify that the system behaves as specified. The interface used for testing is the System Interface which may consistof one or more of: the user interface, the network interface, dedicated hardware interfaces, etc.

System Testing

- This is testing the system as a whole and is invariably done using black box testing
- System testing can be broken into a number of categories. Not all of these may be carried out on the product or be applicable to it.

System Testing - Categories

- **Interoperability Testing** Used to verify that the software can exchange and share information with other required software products. Typically, tests are run on all pairs of products to see that all the information that needs to be shared is transferred correctly. This is particularly important for communications software.
- Note that Conformance Testing is used to make sure that a system conforms to a standard; Interoperability Testing is used to make sure that it actually works with other conformant products. In theory, both are not needed: in practice, it has been found that conformance testing alone is not sufficient.

System Testing - Categories

- **Conformance Testing** Used to verify that the system conforms to a set of published standards. Many standards will have a published suite of conformance tests, or have a selected authority to run these tests. This is particularly important for communications software, as software system must correctly inter-operate with other implementations.
- **Documentation Testing** Used to verify that the documentation (in printed form, online, help, or prompts) is sufficient for the software to be installed and operated efficiently. Typically a full installation is performed, and then the different system functions are executed, exactly as documented. Responses are checked against the documented responses.

System Testing - Categories

- **Performance/Load Testing** Used to verify that the performance targets for the software have been met. Some of these tests can be static, but most are dynamic. These tests verify that metrics such as the configuration limits (static), response latency or maximum number of simultaneous users (dynamic) are measured and compared to the specified requirements. Note: just measuring performance is not testing. To test performance, there must be a specification of what is required.

System Testing - Categories

- **Portability Testing** Used to verify that the software is portable to another operating environment. A selection of tests from the original platform are run in each new environment. The new platforms may be software platforms (such as 32-bit and 64-bit versions of the same operating system, or implementations by different vendors), or hardware platforms (such as mobile phones, tablets, laptops, workstations, and servers), or different environments (databases, networks, etc.).
- **Regression Testing** Used to verify that any changes to the software (bug fixes, upgrades, new features, or other modifications to the original application) have not introduced new faults to previously working software. This is performed by re-executing some or all of the system tests again. See Section 9.6 for a discussion on using Repair Tests for Regression Testing. Minimising the number of tests to be run is an active research area

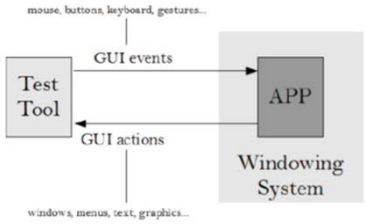
System Testing – Test Environment

- A generic test environment model for System Testing is shown in the Figure. The Test Tool provides input (I/P) to the SUT, and receives output (O/P) from the SUT, over the system interface (I/F).
- In some cases the interface will be synchronous, where every input generates an output.
- In other cases the interface will be asynchronous, where an input may create a sequence of outputs over time, or the software may spontaneously generate outputs based on timers or other internal events

System Testing - Categories

- **Release Testing** Used to verify, prior to distribution, that a product operates properly without interfering with applications or hardware. Typically System Tests are used to exercise the system being released, while the behaviour of other applications and the hardware is monitored for correct operation.
- **Stress Testing** Used to verify failure behaviour and expose defects when the load exceeds the specified limits. Used to verify that the system degrades gracefully or not. This is one form of testing where the requirements may be somewhat vague, and verifying that the software has passed a test may require interpretation. Typically an excessive load is placed on the software, usually in terms of "input events", and the test is at least to ensure that the system does not crash. Ideally, the testing will verify that valid responses continue to be given.

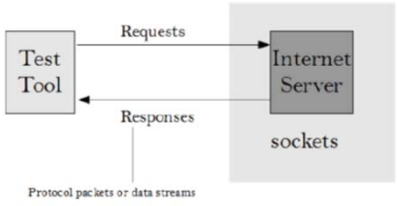
System Testing Models - GUI



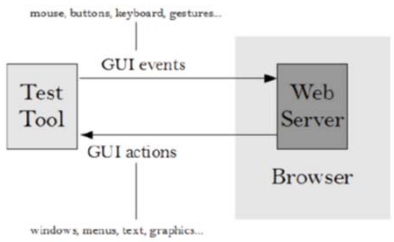
System Testing - Categories

- **Security Testing** Used to verify that the system security features are not vulnerable to attack. The tests will deliberately attempt to break security measures. Examples would include impersonating another user, accessing protected files, accessing network ports, breaking encrypted storage or communications, or inserting unauthorised software.
- **Safety Testing** Used to verify system operation under safety-critical conditions. The tests deliberately cause problems under controlled conditions and verify the system's response. Particularly necessary for life-critical, safety-critical, and mission-critical systems, such as medical device software or avionics.

System Testing Models - Network



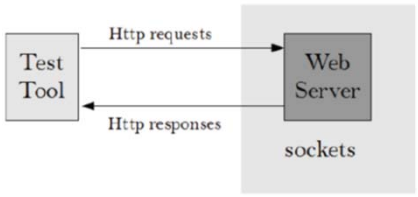
System Testing Models – Web based/Browser



System Testing - GUI Example

- System Testing of a GUI can take the following forms:
  - Ensure the interface can be navigated correctly.
  - Ensure the correct elements are on each screen.
  - Verify the correct output actions occur for each input event.

System Testing Models – Web based/Direct



Navigating the interface

- Any of the described strategies for state-diagram coverage can be used.
- Normally *all transitions* should be aimed for: every user action that causes a change in the user interface causes that change correctly

System Testing - GUI Example

- From a testing viewpoint, a GUI consists of multiple windows, each containing various elements. The elements in turn can contain further elements.
- User input takes the form of input events, to which the system responds with actions which cause the results to be displayed to the user.
- A GUI invariably has a defined sequence of windows that can be displayed based on the user input events: for example, displaying a menu, displaying a sub-menu, displaying a new window on top of the existing window (perhaps to set preferences, or save work to a file).

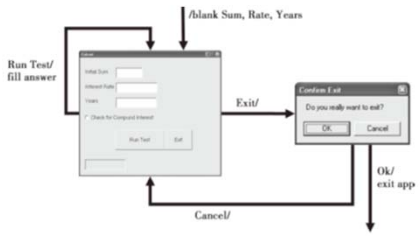
Screen Elements

- If screens are specified in detail they can be checked to ensure that they appear correctly. Example attributes to test for
  - Element type: textbox, button
  - Element placement: abolute position, relative position
  - Element values: initial, modified
  - Element appearance: size, font , colour

Correct behaviour

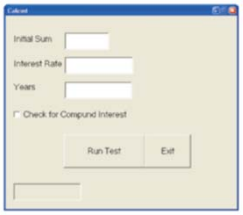
- Checked using the methods of black-box testing
- Also consider inputs may be on different windows or menus
- Expected outputs may appear on different windows

Simple Interest Calculator- GUI State Diagram



Simple Interest Calculator GUI

- Considering an example system of a simple interest calculator that has two windows, a Main window



Simple Interest Calculator- GUI State Diagram

- On entry, the *Calcint* window is displayed and the fields *initial sum*, *interest rate* and *years* are blank.
- If the *Run Test* button is clicked, the *answer* is filled.
- If the *Exit* button is clicked, the *Confirm Exit* window is activated.
- If the *Cancel* button is clicked, the *Confirm Exit* window is deactivated, and focus returns to the *Calcint* window.
- If the *OK* button is pressed, the action *exit app* is taken to terminate the application.

Simple Interest Calculator GUI

- and an Exit window

Simple Interest Calculator

• To the nearest Euro the interest calculation is given by

$$interest = \begin{cases} initial * (1 + rate) * years & \text{if not compound} \\ initial * (1 + rate)^{years} & \text{if compound} \end{cases}$$

System Tests – Navigating the interface

- Use an *every transition* test strategy, where each transition is a test case.
- **Test Cases**
  1. Entry transition
  2. Run Test Transition from Calcint window
  3. Exit transition from Calcint window
  4. Cancel transition from Confirm Exit window
  5. OK transition from Confirm Exit window
- These can be tested within a single test

System Tests – Interface appearance

Test No.	Test Cases Covered	Inputs	Expected Output
2	1, 2, 3, 4, 5	Start application Sum.text() Rate.test() Years.test()	Calcint window active Empty string Empty string Empty string

System Tests – Navigating the interface

Test No.	Test Cases Covered	Inputs	Expected Output
1	1, 2, 3, 4, 5	Start application Click Run Test Click Exit Click Cancel Click Exit Click OK	Calcint window active Calcint window active Confirm Exit window active Calcint window active Confirm Exit window active Application exits

System Tests – Functional Tests

- If Equivalence Partitioning is applied then the following test cases are valid

1. Initial sum=100
2. Interest rate=5
3. Years=10
4. Compound interest=yes
5. Compound interest=no

System Tests – Interface appearance

- The specification states that the text output boxes should be initially blank

System Tests – Functional Tests

Test No.	Test Cases Covered	Inputs	Expected Output
3	1, 2, 3, 4	Initial sum=100 Interest rate=5 Years=10 Compound interest=unchecked Run test	Output=150
4	1, 2, 3, 5	Initial sum=100 Interest rate=5 Years=10 Compound interest=checked Run test	Output=163

Acceptance Testing

- Acceptance Testing (commonly referred to as Alpha and Beta Testing) is used to verify that a system works correctly in a real operational environment.
- Alpha Testing is usually restricted in scope and carried out at the software developer's location.
- In Beta testing, the software is usually distributed for many volunteer users to run in their own environment.
- Beta testing is particularly good for identifying faults that occur during normal use - factors which might not have been considered by the development team may be identified and fixed prior to release of the product.

Beta Testing Key Factors

- Making the test group as large and diverse as possible.
- Providing incentives for users to return reports.
- Keeping the user evaluations confidential.

Customer Acceptance Testing

- Customer Acceptance Tests are used by a customer to ensure that the system that they ordered works properly. Typically they would be used prior to payment for the system, or they might be used in a comparative analysis of multiple tendered solutions.
- In some cases the tests, or only a subset of the tests, are released to the software developer.
- In other cases, the tests are kept secret, and details only released of tests that fail.