CUI Abbottabad

Department of Computer Science

SOFTWARE TESTING

Lecture 10

Integration Testing, Interface Errors and Integration Techniques

CONTENT…

- Integration Testing
- Different Types of Interface Errors
- Integration Techniques
  - Incremental
  - Top Down
  - Bottom Up
  - Big Bang
  - Sandwich

INTEGRATION TESTING

- A software module is a self-contained element of a system.
- Modules are individually tested commonly known as *unit testing.* Next major task is to put the modules, i.e., pieces together to construct the complete system.
  - *Construction of a working system from the pieces is not a straightforward task because of numerous interface errors.*
- Objective of *system integration testing* (SIT) is to build a "working" version of the system.
  - *Putting modules together in an incremental manner.*
  - *Ensure that additional modules work as expected without disturbing the functionalities of the modules already put together.*
- ***Integration testing*** is said to be complete when:
  - The system is fully integrated together
  - All the test cases have been executed
  - All the severe and moderated defects found have been fixed

Module 1 Alone

Module 1 is tested and working fine

Module 2 Alone

Module 2 is Tested and Working fine

Module 1

Module 2

After Integration second module is not working fine

INTEGRATION TESTING

- Advantages of conducting *system integration testing* (SIT) are as follows:
  - Defects are detected early
  - It is easier to fix defects detected earlier
  - We get earlier feedback on the health and acceptability of the individual modules and on the overall system
  - Scheduling of defect fixes is flexible, and it can overlap with development

INTERFACE ERRORS

- Three common paradigms for interfacing modules:
  - Procedure call interface
  - Shared memory interface
  - Message passing interface
- The problem arises when we “put modules together” because of interface errors.
- **Interface errors:** Interface errors are those that are associated with structures existing outside the local environment of a module, but which the module uses.

DIFFERENT TYPES OF INTERFACE ERRORS

- Construction
- Inadequate functionality
- Location of functionality
- Changes in functionality
- Added functionality
- Misuse of interface
- Misunderstanding of interface
- Data structure alteration
- Inadequate error processing
- Additions to error processing
- Inadequate post-processing
- Inadequate interface support
- Initialization/value errors
- Validation od data constraints
- Timing/performance problems
- Coordination changes
- Hardware/software interfaces

CONSTRUCTION

- Some programming languages, such as C, generally separate the interface specification from the implementation code.
- In a C program, programmers can write a statement #include header.h, where header.h contains an interface specification.
- Since the interface specification lies somewhere away from the actual code, programmers overlook the interface specification while writing code.
- Therefore, inappropriate use of #include statements cause construction errors.

INADEQUATE FUNCTIONALITY

- These are errors caused by implicit assumptions in one part of a system that another part of the system would perform a function.
- However, in reality, the “other part” does not provide the expected functionality intentionally or unintentionally by the programmer who coded the other part.

LOCATION OF FUNCTIONALITY

- Disagreement on or misunderstanding about the location of a functional capability within the software leads to this sort of error.
- The problem arises due to the design methodology, since these disputes should not occur at the code level.
- It is also possible that inexperienced personnel contribute to the problem.

CHANGES IN FUNCTIONALITY

- Changing one module without correctly adjusting for that change in other related modules affects the functionality of the program.

ADDED FUNCTIONALITY

- A completely new functional module, or capability, was added as a system modification.
- Any added functionality after the module is checked in to the version control system without a cross reference is considered to be an error.

MISUSE OF INTERFACE

- One module makes an error in using the interface of a called module. This is likely to occur in a procedure–call interface.
- Interface misuse can take the form of wrong parameter type, wrong parameter order, or wrong number of parameters passed.

MISUNDERSTANDING OF INTERFACE

- A calling module may misunderstand the interface specification of a called module. The called module may assume that some parameters passed to it satisfy a certain condition, whereas the caller does not ensure that the condition holds.
- For example, assume that a called module is expected to return the index of an element in an array of integers. The called module may choose to implement binary search with an assumption that the calling module gives it a sorted array. If the caller fails to sort the array before invoking the second module, we will have an instance of interface misunderstanding.

DATA STRUCTURE ALTERATION

- The problem arises when the size of a data structure is inadequate, or it fails to contain a sufficient number of information fields.
- The problem has its genesis in the failure of the high-level design to fully specify the capability requirements of the data structure.

INADEQUATE ERROR PROCESSING

- A called module may return an error code to the calling module. However, the calling module may fail to handle the error properly.

INADEQUATE POST-PROCESSING

- These errors are caused by a general failure to release resources no longer required, for example, failure to deallocate memory.

INADEQUATE INTERFACE SUPPORT

- The actual functionality supplied was inadequate to support the specified capabilities of the interface.
- For example, a module passes a temperature value in Celsius to a module which interprets the value in Fahrenheit.

INITIALIZATION/VALUE ERRORS

- A failure to initialize, or assign, the appropriate value to a variable data structure leads to this kind of error. Problems of this kind are usually caused by simple oversight.
- For example, the value of a pointer can change; it might point to the first character in a string, then to the second character, after that to the third character, and so on. If the programmer forgets to reinitialize the pointer before using that function once again.

VALIDATION OF DATA CONSTRAINTS

- A specified relationship among data items was not supported by the implementation. This can happen due to incomplete detailed design specifications.

TIMING/PERFORMANCE PROBLEMS

- These errors were caused by inadequate synchronization among communicating processes.
- A race condition is an example of these kinds of error. In the classical race, there are two possible events event a and event b happening in communicating processes process A and process B, respectively. There is logical ground for expecting event a to precede event b. However, under an abnormal condition event b may occur before event a. The program will fail if the software developer did not anticipate the possibility of event b preceding event a and did not write any code to deal with the situation.

COORDINATION CHANGES

- These errors are caused by a failure to communicate changes to one software module to those responsible for other interrelated modules.

HARDWARE/SOFTWARE INTERFACES

- These errors arise from inadequate software handling of hardware devices.
- For example, a program can send data at a high rate until the input buffer of the connected device is full. Then the program has to pause until the device frees up its input buffer. The program may not recognize the signal from the device that it is no longer ready to receive more data. Loss of data will occur due to a lack of synchronization between the program and the device.

SYSTEM INTEGRATION TECHNIQUES

- Common approaches to perform system integration testing:
  - Incremental
  - Top-down
  - Bottom-up
  - Sandwich
  - Big-bang
- **Pre-requisite**
  - A module must be available to be integrated
  - A module is said to *available* for combining with other modules when the module's *check-in request form* is ready

INCREMENTAL

- Integration testing is conducted in an incremental manner as a series of test cycles
- In each test cycle, a few more modules are integrated with an existing and tested build to generated larger builds
- The complete system is built, cycle by cycle until the whole system is operational for system-level testing.

TOP-DOWN

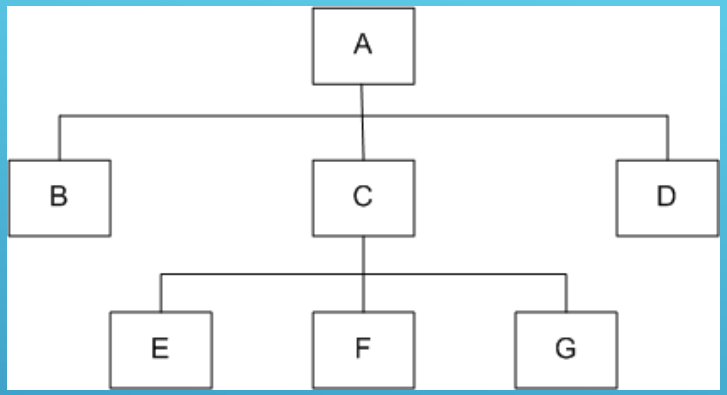


Figure 1: A module hierarchy with three levels and seven modules

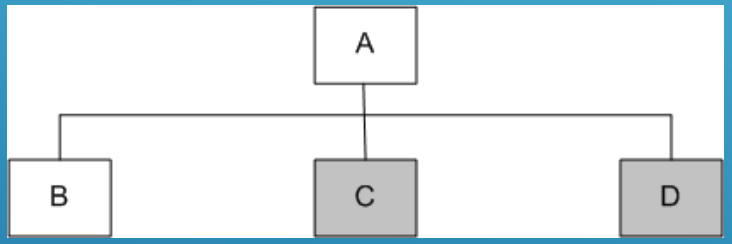


Figure 2: Top-down integration of modules A and B

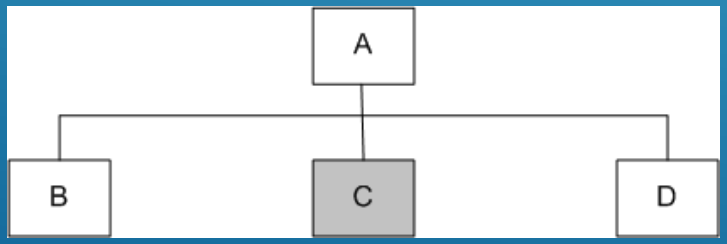


Figure 3: Top-down integration of modules A, B and D

- Module A has been decomposed into modules B, C, and D
- Modules B, D, E, F, and G are terminal modules
- First integrate modules A and B using stubs C` and D` (represented by grey boxes)
- Next stub D` has been replaced with its actual instance D
- Two kinds of tests are performed:
  - Test the interface between A and D
  - Regression tests to look for interface defects between A and B in the presence of module D

TOP-DOWN

- Stub C` has been replaced with the actual module C, and new stubs E`, F`, and G`
- Perform tests as follows:
  - first, test the interface between A and C;
  - second, test the combined modules A, B, and D in the presence of C
- The rest of the process depicted in the right hand side figures.

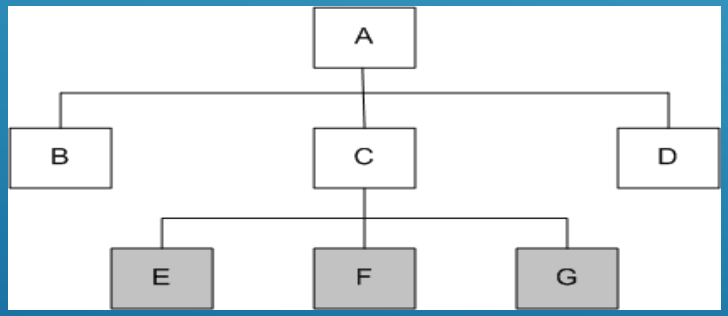


Figure 4: Top-down integration of modules A, B, D and C

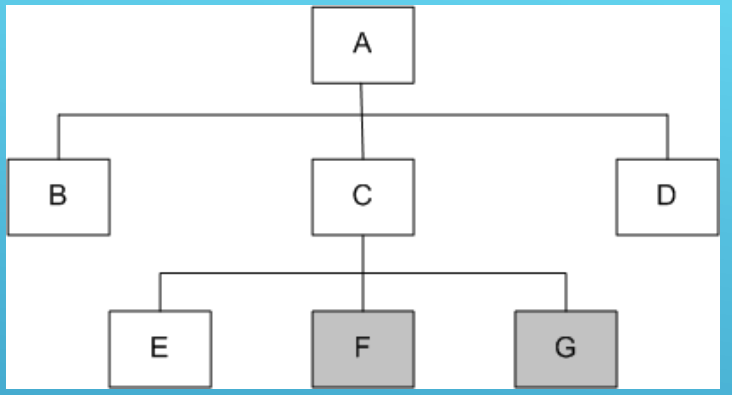


Figure 5: Top-down integration of modules A, B, C, D and E

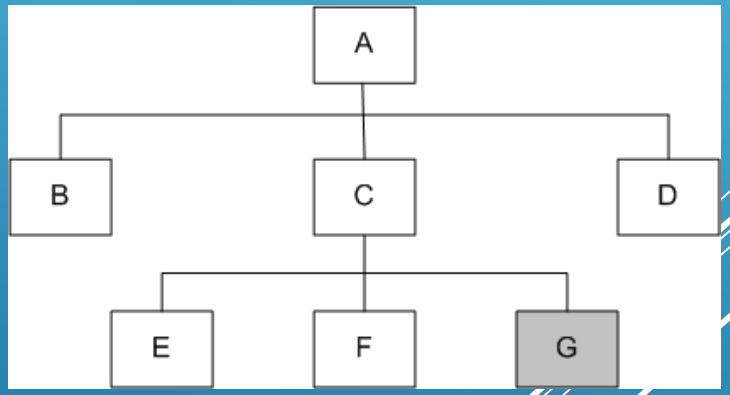


Figure 6: Top-down integration of modules A, B, C, D, E and F

BOTTOM-UP

- We design a test driver to integrate lowest-level modules E, F, and G
- Return values generated by one module is likely to be used in another module
- The test driver mimics module C to integrate E, F, and G in a limited way.
- The test driver is replaced with actual module , i.e., C.
- A new test driver is used
- At this moment, more modules such as B and D are integrated
- The new test driver mimics the behavior of module A. Finally, the test driver is replaced with module A and further test are performed

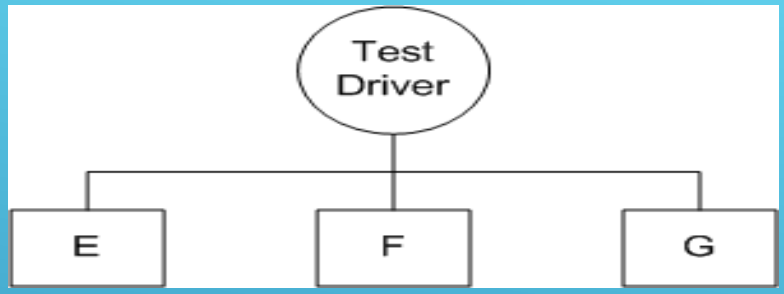


Figure 1: Bottom-up integration of module E, F, and G

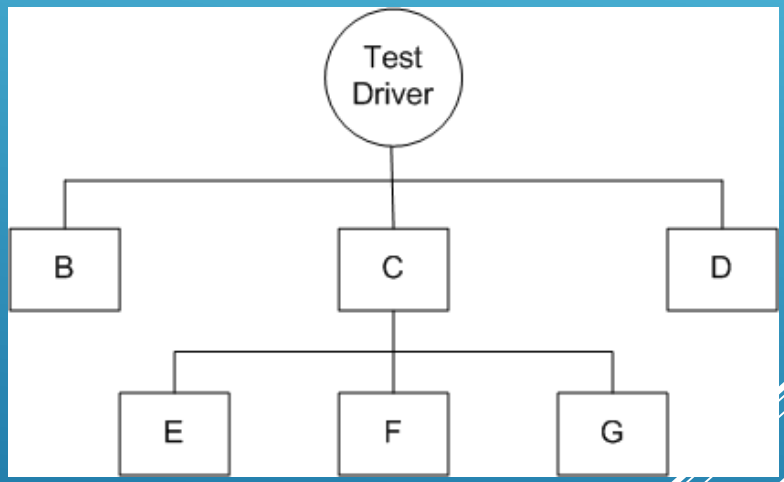


Figure 2: Bottom-up integration of module B, C, and D with F, F, and G

STUBS AND DRIVERS

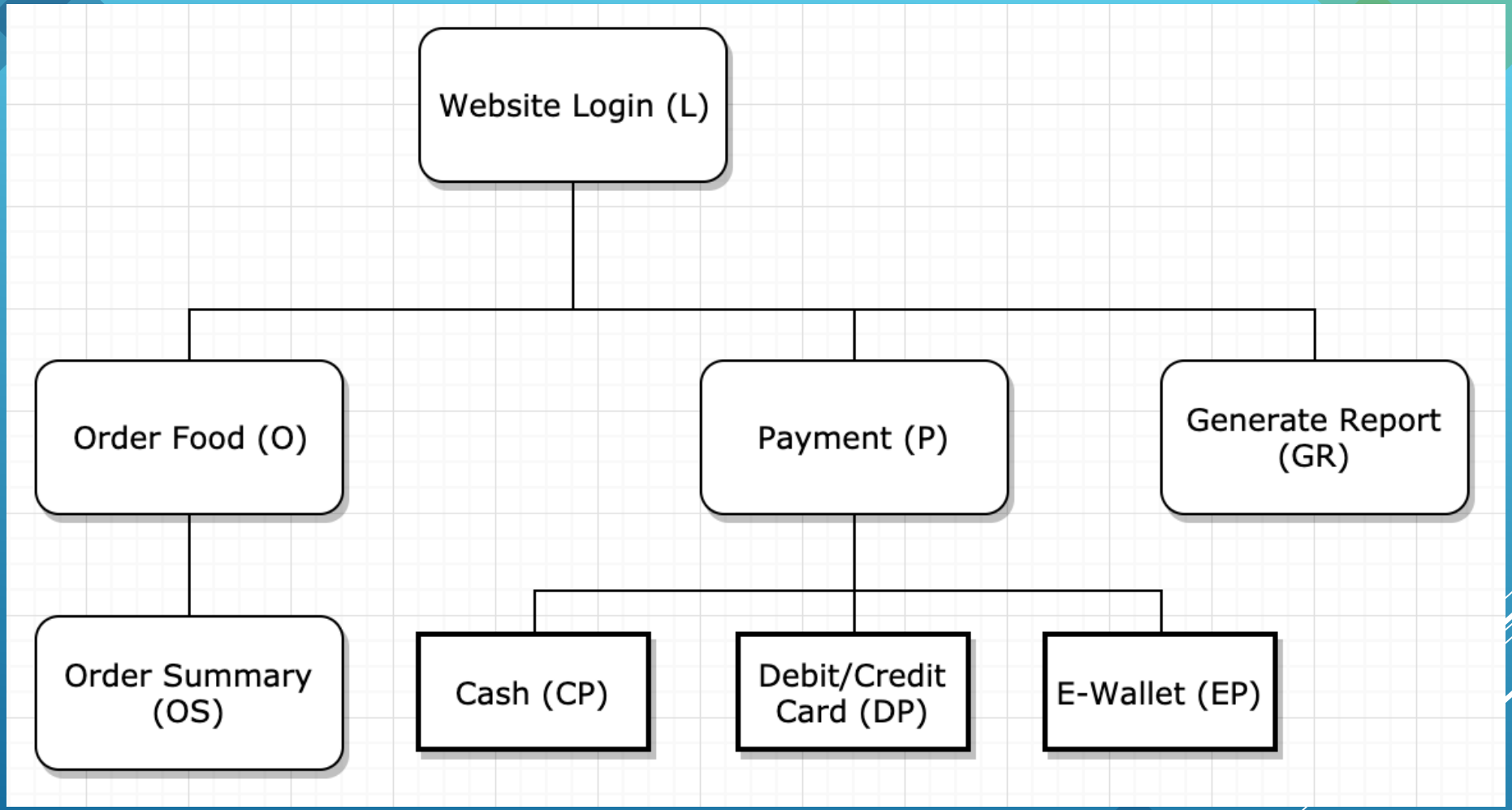
- **Stubs and drivers** are two such elements used in software testing process, which act as a temporary replacement for a module
- Or the **replica** of the modules, which acts as a substitute to the undeveloped or missing module

STUBS

- Stubs are used to test modules and are created by the team of testers during the process of Top Down approach of Integration Testing. With the assistance of these test stubs testers can stimulate the behavior of the lower level modules that are not yet integrated with the software.

DRIVERS

- Drivers, like stubs, are used by software testers to fulfil the requirements of missing or incomplete components and modules. These are usually complex than stubs and are developed during Bottom Up approach of integration testing.
- Drivers can be utilized to test the lower levels of the code, when the upper level of codes or modules are not developed.


Website Login (L)
Order Food (O)
Payment (P)
Generate Report (GR)
Order Summary (OS)
Cash (CP)
Debit/Credit Card (DP)
E-Wallet (EP)

BIG-BANG AND SANDWICH

- **Big-bang Approach**
  - First all the modules are individually tested
  - Next all those modules are put together to construct the entire system which is tested as a whole
- **Sandwich Approach**
  - In this approach a system is integrated using a mix of top-down, bottom-up, and big-bang approaches
  - A hierarchical system is viewed as consisting of three layers
  - The bottom-up approach is applied to integrate the modules in the bottom-layer
  - The top layer modules are integrated by using top-down approach
  - The middle layer is integrated by using the big-bang approach after the top and the bottom layers have been integrated