Integration testing

Satish Mishra

mishra@informatik.hu-berlin.de

This session

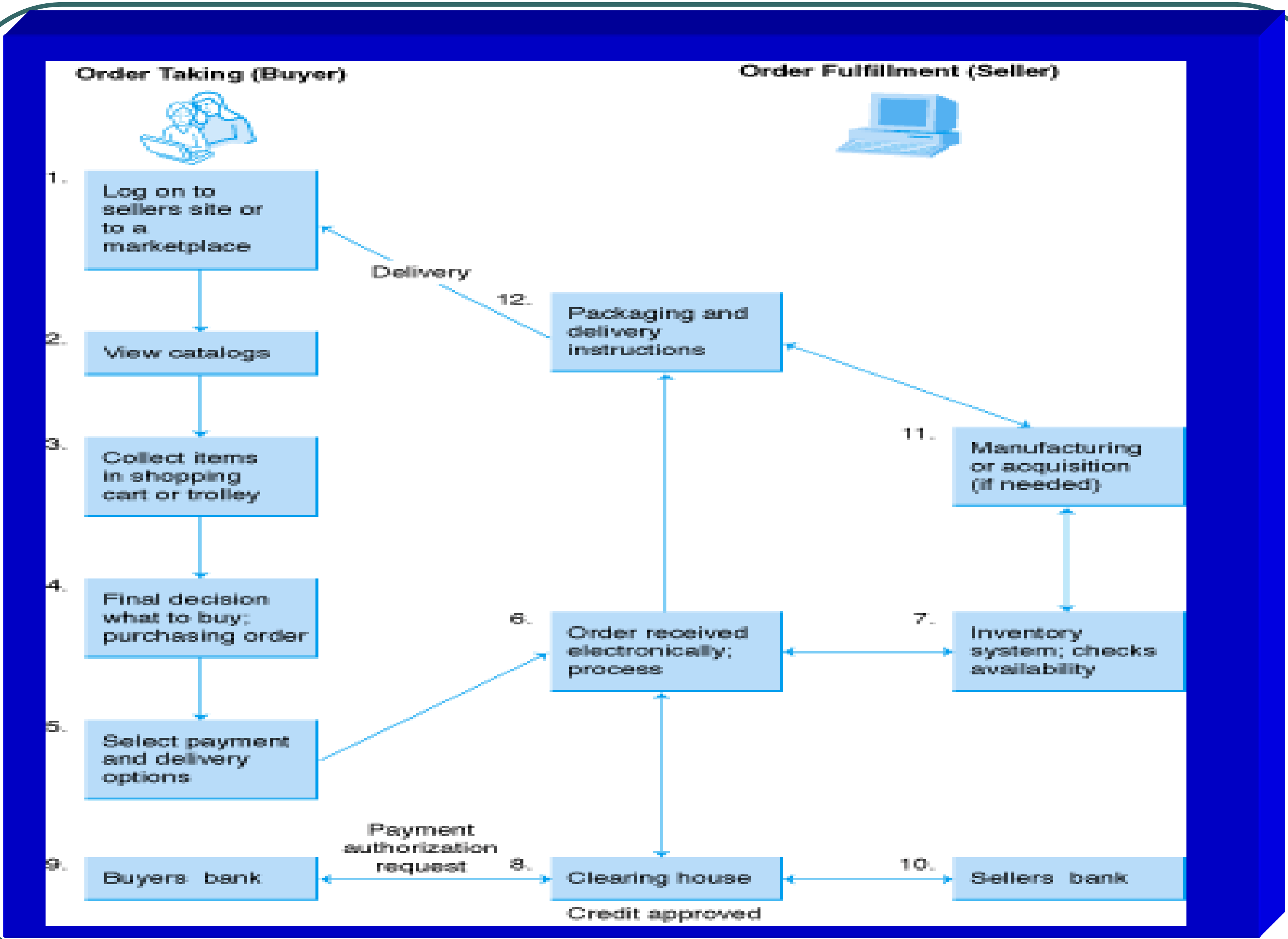
- Integration testing
- Component / Module testing
  - Stub and driver
  - Examples
- Different approaches to integration testing
  - Bottom-up
  - Top-down
  - Big-bang
  - Sandwich
- Exercise
  - Discussions

What is integration testing

- Testing in which software components, hardware components, or both together are combined and tested to evaluate interactions between them

- Integration testing usually go through several realword business scenarios to see whether the system can successfully complete workflow tasks

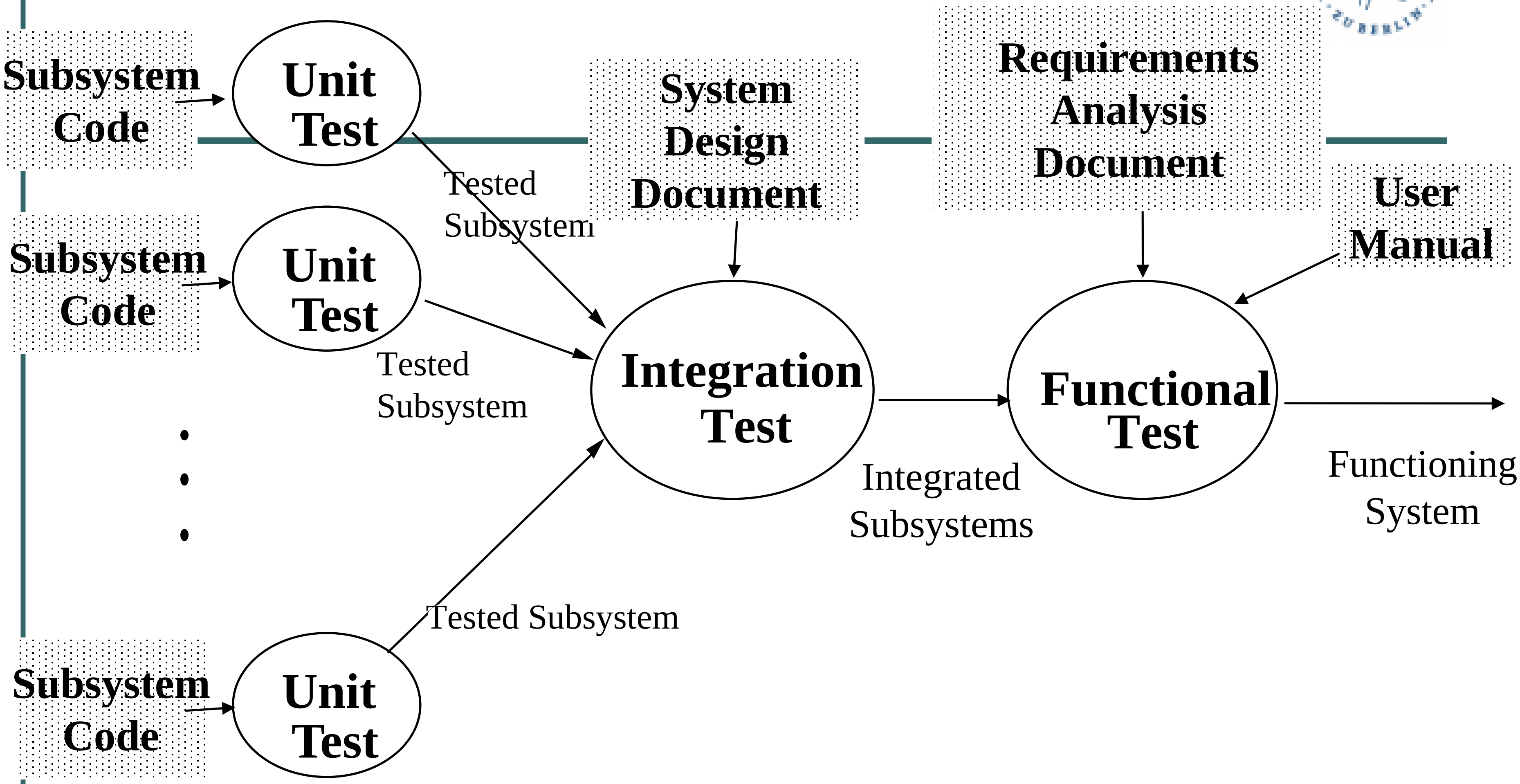
- Integration plan specifies the order of combining the modules into partial systems


Order Taking (Buyer)
Order Fulfillment (Seller)
1.
Log on to sellers site or to a marketplace
2.
View catalogs
3.
Collect items in shopping cart or trolley
4.
Final decision what to buy; purchasing order
5.
Select payment and delivery options
Delivery
12.
Packaging and delivery instructions
11.
Manufacturing or acquisition (if needed)
6.
Order received electronically; process
7.
Inventory system; checks availability
9.
Buyers bank
Payment authorization request
8.
Clearing house
Credit approved
10.
Sellers bank

Component / Module testing

- A unit is the smallest testable piece of software. A unit is defined as a database trigger, stored procedure, function, report, form, batch load program or PL/SQL program.
- Unit testing is the testing that is performed to validate that the unit does satisfy its functional specification and/or that its implementation structure does match the intended design structure.
- Module /Component consists of units integrated into a module that performs a specific business function

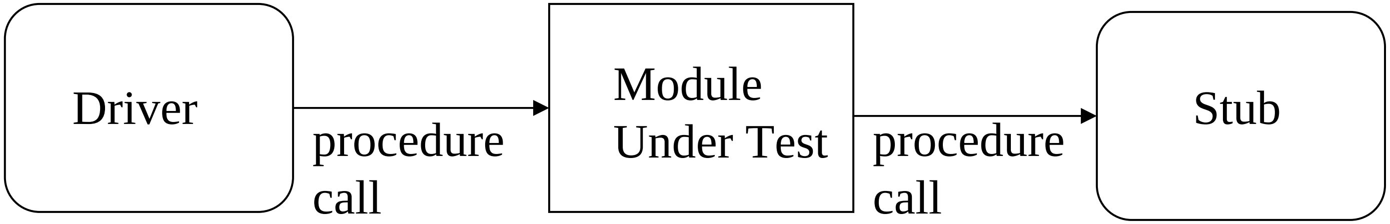
Testing Activities



Drivers and Stubs

- **Driver:** A program that calls the interface procedures of the module being tested and reports the results
  - A driver simulates a module that calls the module currently being tested
- **Stub:** A program that has the same interface procedures as a module that is being called by the module being tested but is simpler.
  - A stub simulates a module called by the module currently being tested

Drivers and Stubs



Example 1:

- We now consider an example in which we need to develop a "calculator" that converts between different units (e.g., feet to inches).
  - We will implement in java !!
  - Class UnitConverter
    - Method convert
    - Method getMultiplier
  - Class Driver
    - Method main

Soln..

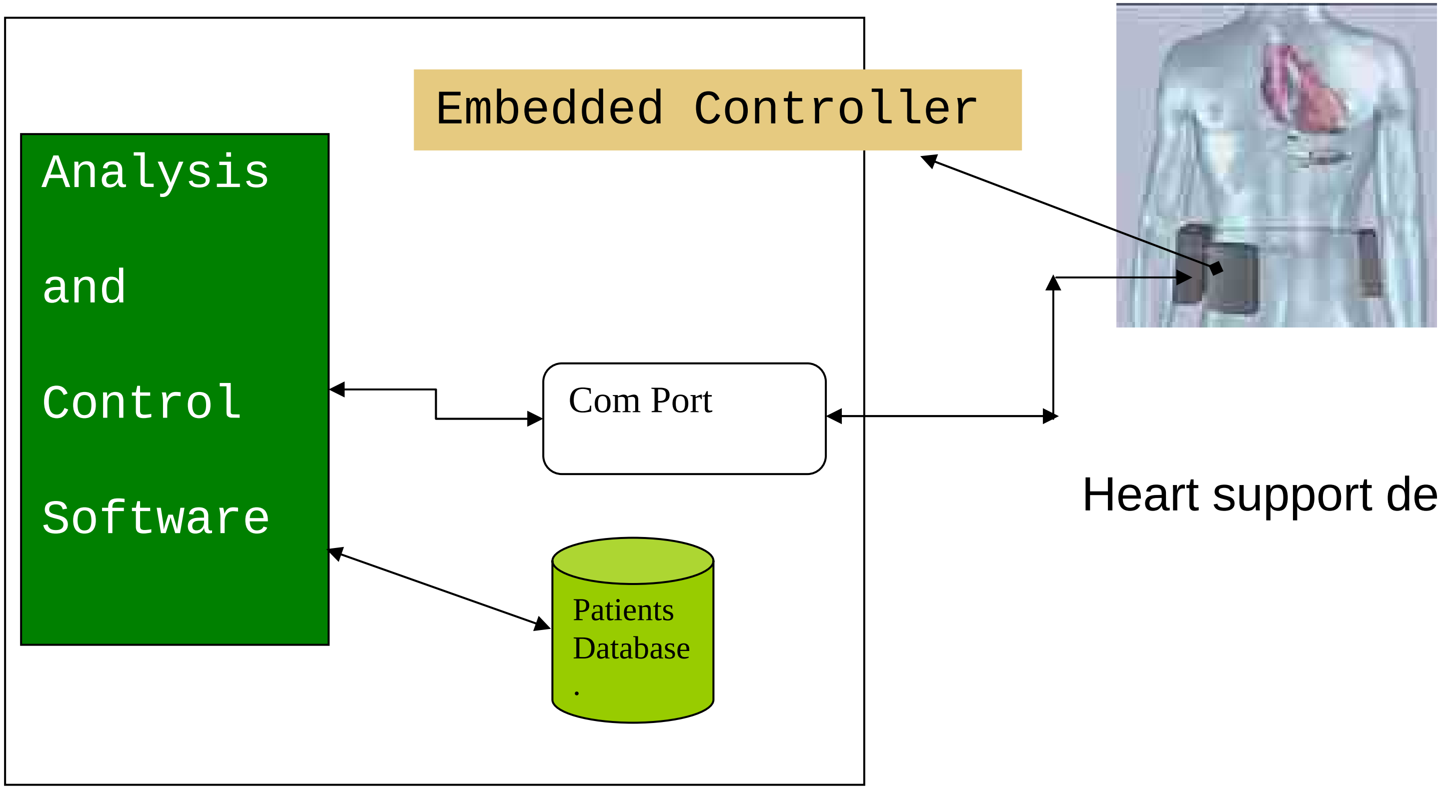
```
public class UnitConverter
{
public UnitConverter()   // Not writing comments since we don't have place..
    {
    }
public double convert(double value, String from, String to)
    {
        double     result;
        result = value * getMultiplier(from, to);
        return result;
    }
public double getMultiplier(String from, String to)
    {
        double       multiplier;
        multiplier = 1.0;
        if  (from.equals("inches")) {
            if      (to.equals("feet"))   multiplier = 1.0/12.0;
            else if (to.equals("yards"))  multiplier = 1.0/12.0/3.0;
            else if (to.equals("miles"))  multiplier = 1.0/12.0/3.0/1760.0;
        } else if (from.equals("feet")) {
            if      (to.equals("inches")) multiplier = 12.0;
            else if (to.equals("yards"))  multiplier = 1.0/3.0;
            else if (to.equals("miles"))  multiplier = 1.0/3.0/1760.0;
        } else if (from.equals("yards")) {
            if      (to.equals("inches")) multiplier = 3.0*12.0;
            else if (to.equals("feet"))   multiplier = 3.0;
            else if (to.equals("miles"))  multiplier = 1.0/1760.0;
        } else if (from.equals("miles")) {
            if      (to.equals("inches")) multiplier = 12.0*3.0*1760.0;
            else if (to.equals("yards"))  multiplier = 1760.0;
            else if (to.equals("feet"))   multiplier = 3.0*1760.0;
        }
        return multiplier;
    }
}
```

Soln..

```
public class Driver
{
public static void main(String[] args)
    {
        double                converted, original;
        int                   i, j;
        String                from, to;
        String[]              units = {"inches","feet","yards","miles"};
        UnitConverter         calculator;
        calculator = new UnitConverter();
        original  = 10.0;

        for (i=0; i < units.length; i++) {
           for (j=0; j < units.length; j++) {
              from      = units[i]; // This could be outside the inner loop
              to        = units[j];
              converted = calculator.convert(original, from, to);
              System.out.println(original+" "+from+" = "+converted+" "+to);
           }
        }
    }
}
```

How to test this system ??

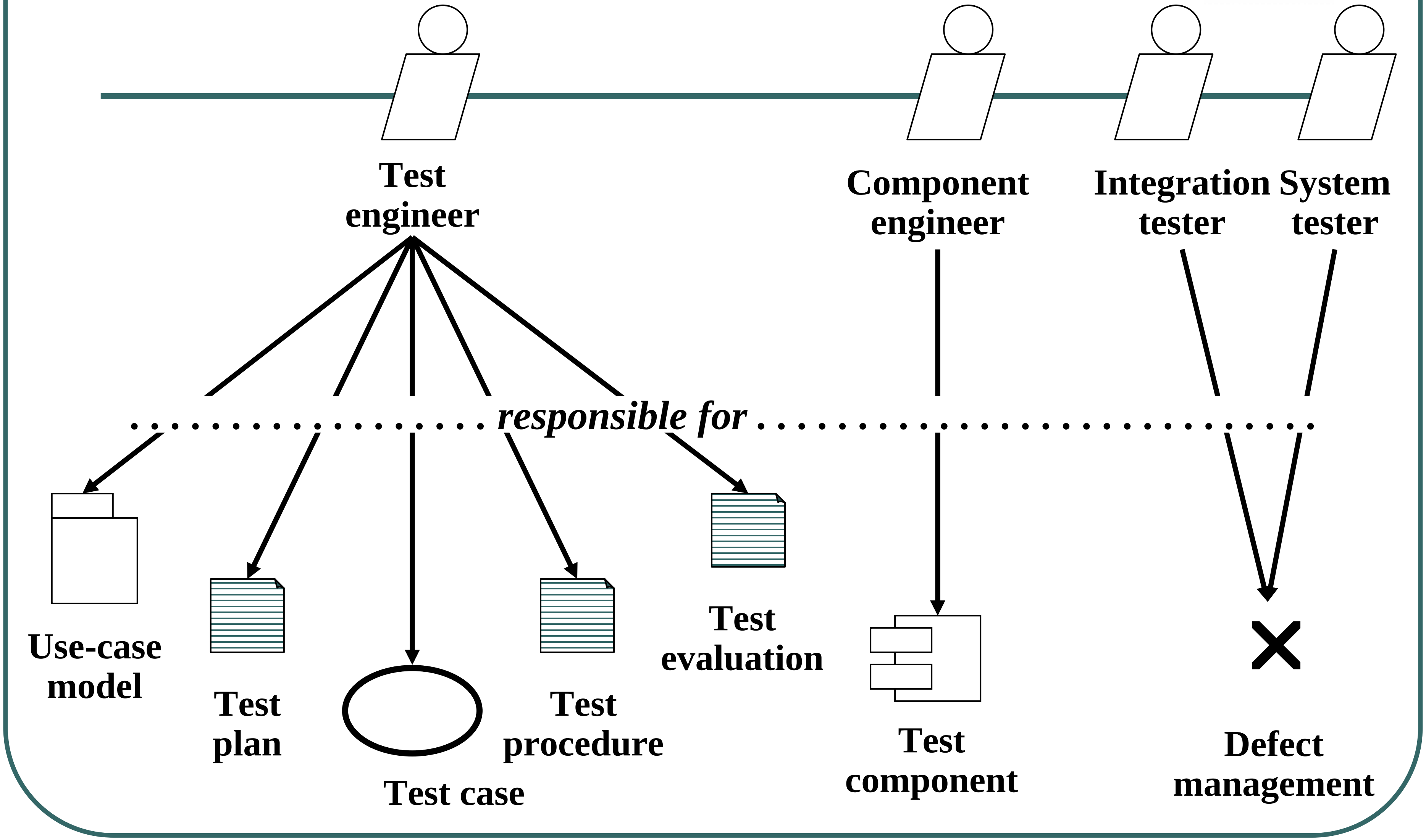


Heart support device

Integration testing approaches

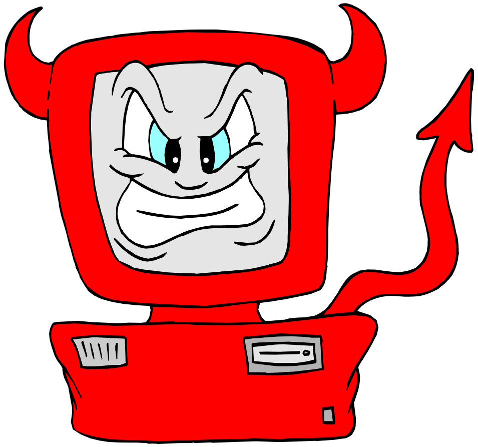
- Different approaches to integration testing
  - Bottom-up
  - Top-down
  - Big-bang
  - Sandwich

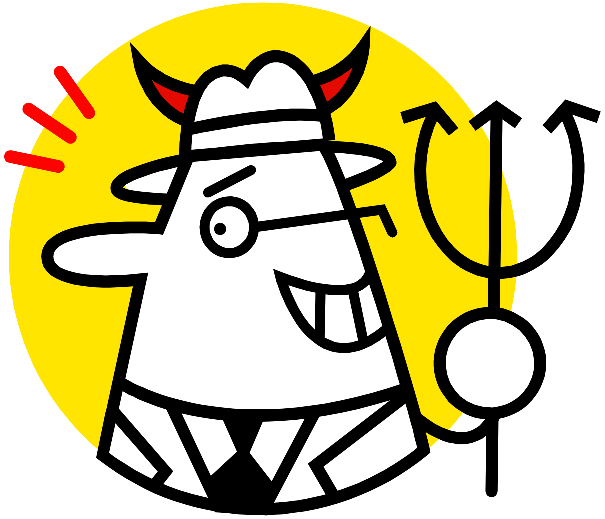
Artifacts and Roles for Integration Testing



What we have here…

- Developers do everything they can to make the software work.
- Testers do everything they can to make the software fail.

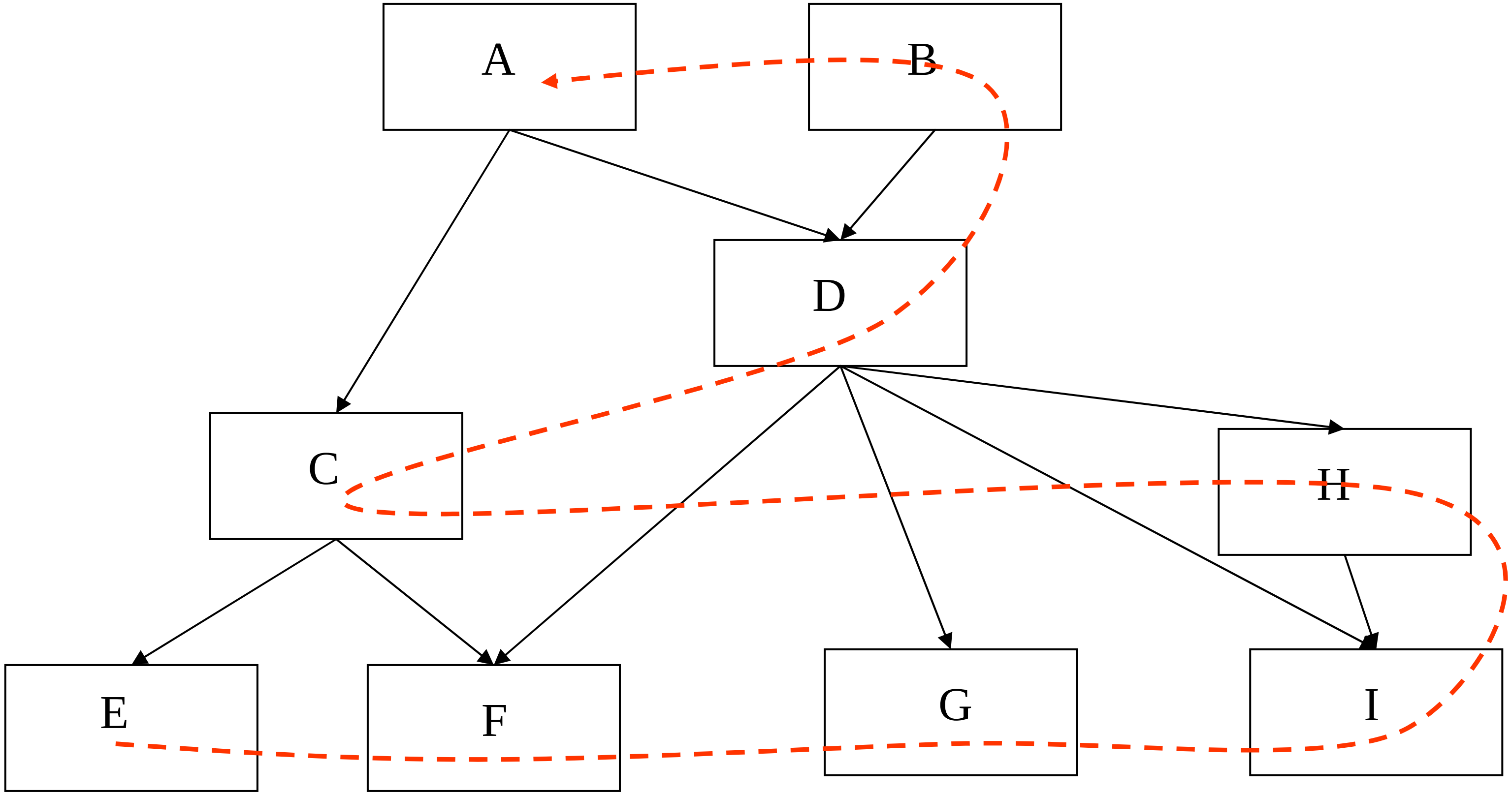




Bottom-Up Integration

- Only terminal modules are tested in isolation
- Modules at lower levels are tested using the previously tested higher level modules
- This is done repeatedly until all subsystems are included in the testing
- Requires a module driver for each module to feed the test case input to the interface of the module being tested
  - However, stubs are not needed since we are starting with the terminal modules and use already tested modules when testing modules in the lower levels
- Disadvantage: Tests the most important subsystem last

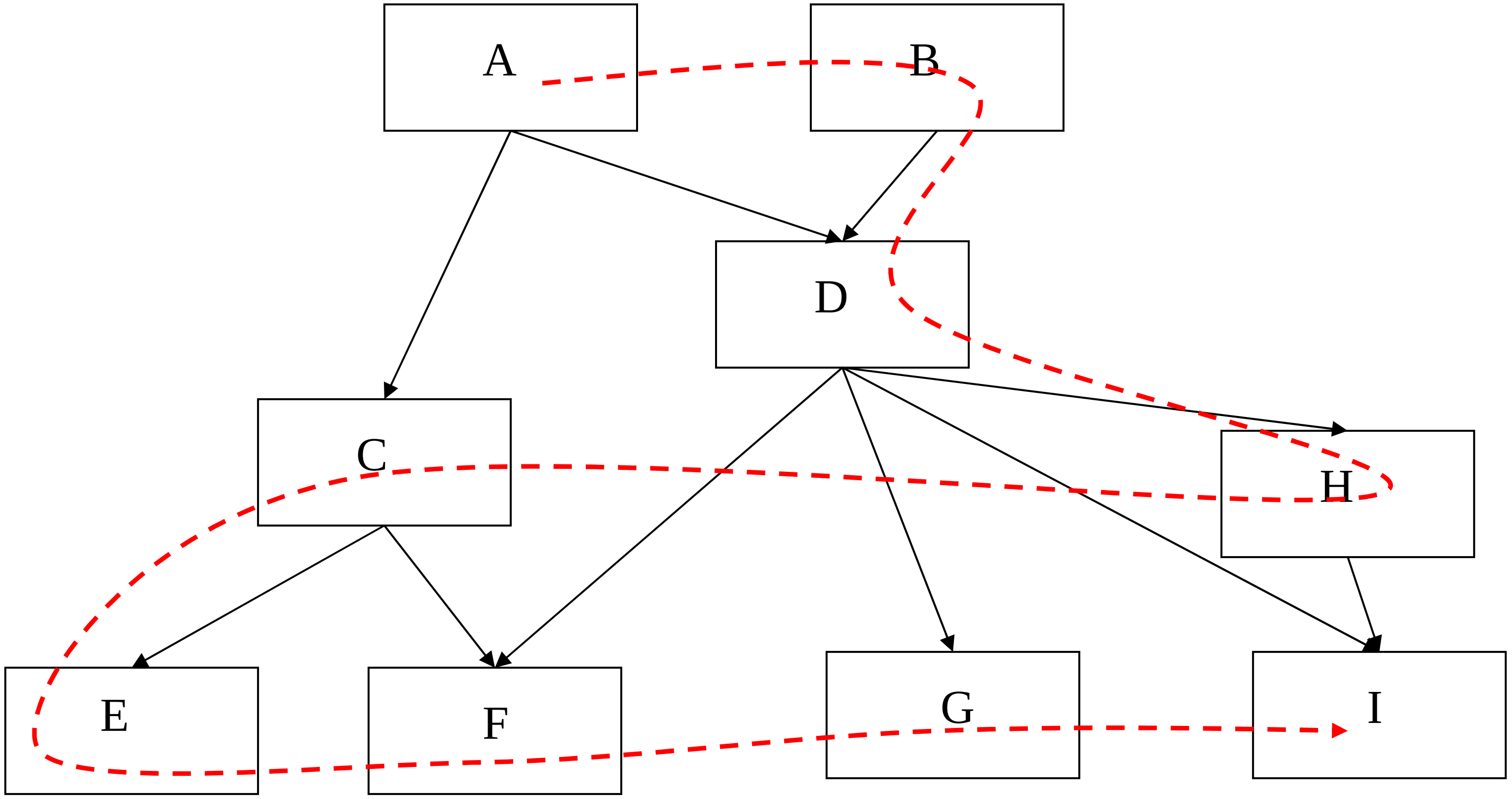
Bottom-up Integration



Top-down Integration

- Only modules tested in isolation are the modules which are at the highest level
- After a module is tested, the modules directly called by that module are merged with the already tested module and the combination is tested
- Do this until all subsystems are incorporated into the test
- Requires stub modules to simulate the functions of the missing modules that may be called
  - However, drivers are not needed since we are starting with the modules which is not used by any other module and use already tested modules when testing modules in the higher levels

Top-down Integration



Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is view as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers
- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration

Other Approaches to Integration

- Big Bang Integration
  - Every module is unit tested in isolation
  - After all of the modules are tested they are all integrated together at once and tested

Implementation Approaches Advantages and Disadvantages

- Big bang advantages
  - None

- Big bang disadvantages
  - Difficult to debug
  - Much throwaway code
  - Critical and peripheral modules not distinguished
  - User does not see product until very late in the development cycle

Implementation Approaches Advantages and Disadvantages

- Top-down advantages
  - Separately debugged modules
  - System test by integrating previously debugged modules
  - Stubs are easier to code than drivers
  - User interfaces are top-level modules

- Top-down disadvantages
  - Stubs must be written
  - Low-level, critical modules built last
  - Testing upper-level modules is difficult

Implementation Approaches Advantages and Disadvantages

- Bottom-up advantages
  - Separately debugged modules
  - System test by integrating previously debugged modules
  - Testing upper-level modules is easier

- Bottom-up disadvantages
  - Drivers must be written
  - Upper-level, critical modules are built last
  - Drivers are more difficult to write than stubs
  - User interfaces are top-level modules

Summary of testing

- Verify operation at normal parameter values
  (a black box test based on the unit's requirements)
- Verify operation at limit parameter values
  (black box)
- Verify operation outside parameter values
  (black box)
- Ensure that all instructions execute
  (statement coverage)
- Check all paths, including both sides of all branches
  (decision coverage)

Summary of testing

- Check the use of all called objects
- Verify the handling of all data structures
- Verify the handling of all files
- Check normal termination of all loops

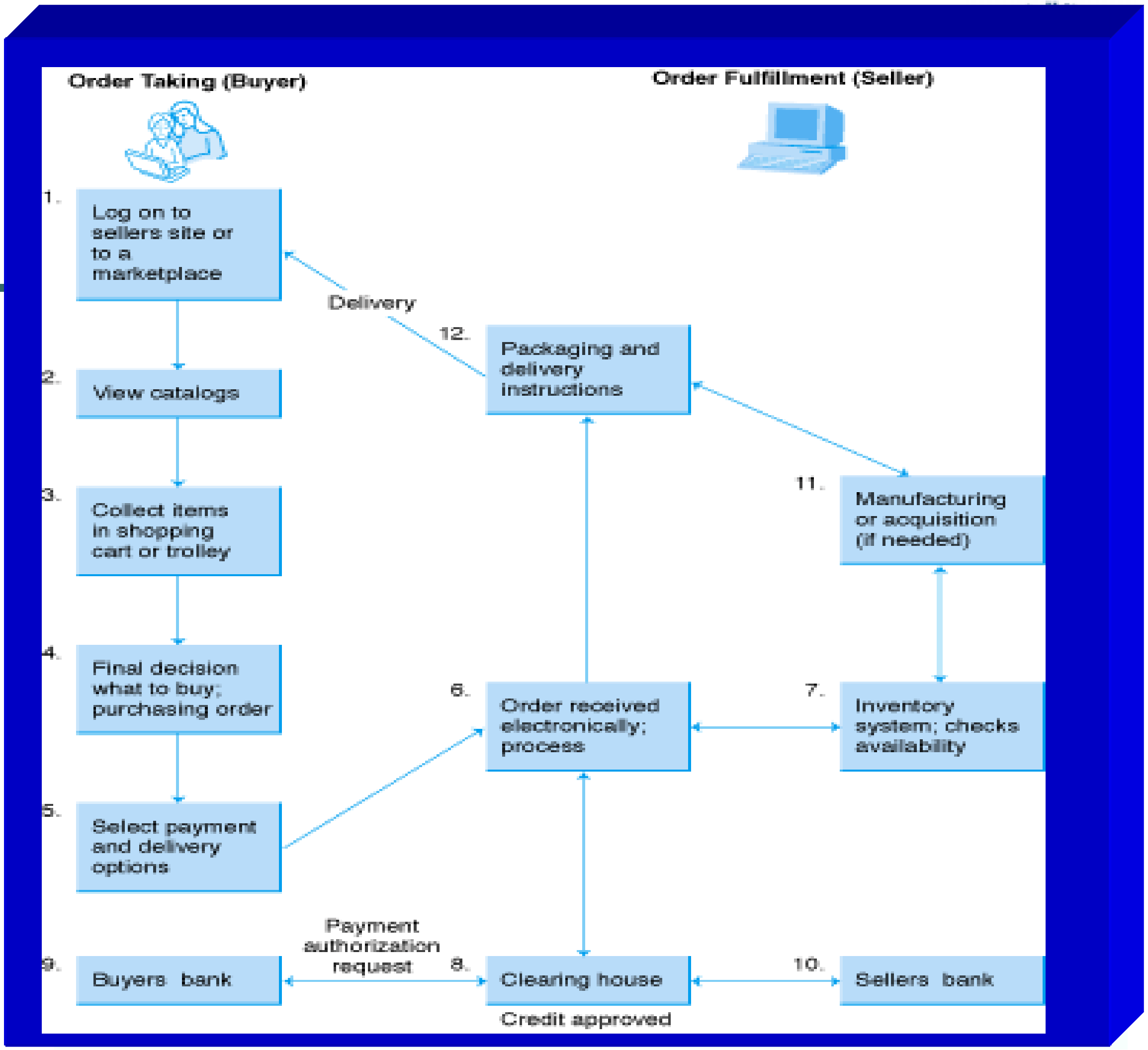
  (part of a correctness proof)
- Check abnormal termination of all loops

Summary of testing

- Check normal termination of all recursions
- Check abnormal termination of all recursions
- Verify the handling of all error conditions
- Check timing and synchronization
- Verify all hardware dependencies

Statements of **(Watts Humphrey)**

Exercise 2.0


Order Taking (Buyer)
Order Fulfillment (Seller)
1. Log on to sellers site or to a marketplace
2. View catalogs
3. Collect items in shopping cart or trolley
4. Final decision what to buy; purchasing order
5. Select payment and delivery options
6. Order received electronically; process
7. Inventory system; checks availability
8. Clearing house
9. Buyers bank
10. Sellers bank
11. Manufacturing or acquisition (if needed)
12. Packaging and delivery instructions
Delivery
Payment authorization request
Credit approved

- Write the integration test cases for the above system with the consideration of integration approach .

Description

- As per the diagram requirement is clear and we should be able to design the system with our knowledge of software engineering.
- Consider object / Module oriented design and write the integration test cases for the system.

Thank You