SCHULICH School of Engineering

UNIVERSITY OF CALGARY

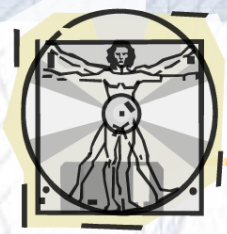
SENG 637
Dependability and Reliability of Software Systems

Chapter 11: Integration, System and Acceptance Testing

Department of Electrical & Software Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

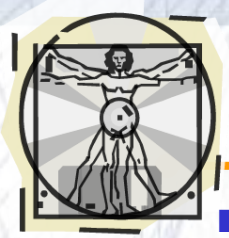
http://people.ucalgary.ca/~far



Contents

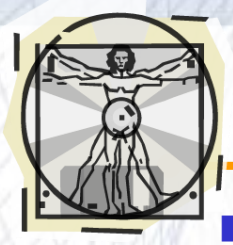
- Test process
  - Unit testing ← already covered
  - Integration testing
    - Big bang strategy
    - Bottom up strategy
    - Top down strategy
    - Sandwich strategy
- System testing
- Acceptance testing

Focus of this chapter



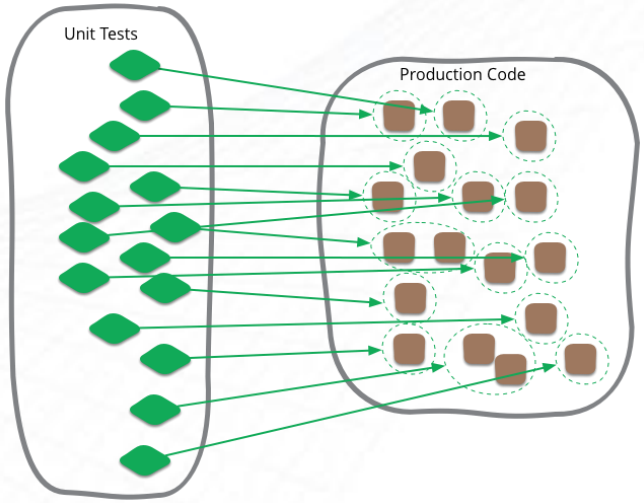
Recap

- We are learning techniques and tools to become test experts
- These techniques can be used by developers and testers
  - Why we need to test?
  - Exploratory testing
  - Unit testing and Junit
  - Black box, white box testing
  - Code coverage
  - Mutation testing
  - GUI testing
  - Test automation
  - System reliability
  - Reliability assessment

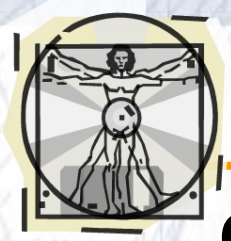


Unit Testing - Recap

- Unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect
- Usually done by developers



A unit is the smallest testable part of an application like functions, methods, classes, procedures, interfaces



Story from NASA …

Let's watch a video

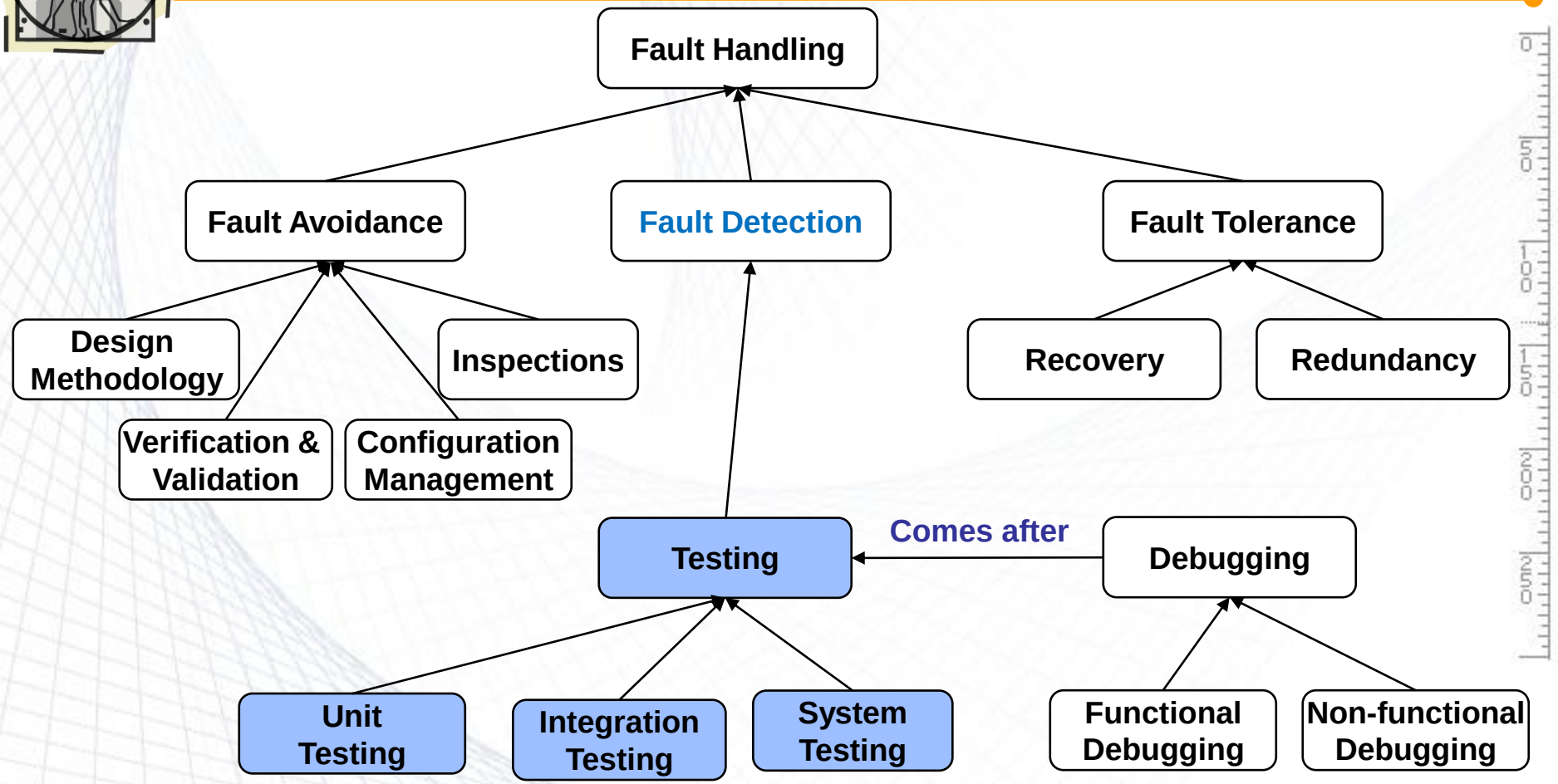
Case: In September 1999, the Mars Climate Orbiter mission failed after successfully traveling 416 million miles in 41 weeks. It disappeared just as it was to begin orbiting Mars. NASA announced a $50,000 project to discover how this could have happened (Fordahl, 1999)

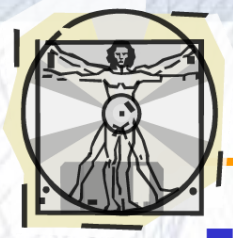
Root cause: two different measuring units were used, English units (pounds) and metric units (newtons)

This fault should have been revealed by integration testing!

https://www.youtube.com/watch?v=urcQAKKAAl0

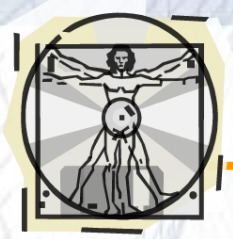
Dealing with SW Failures/Faults



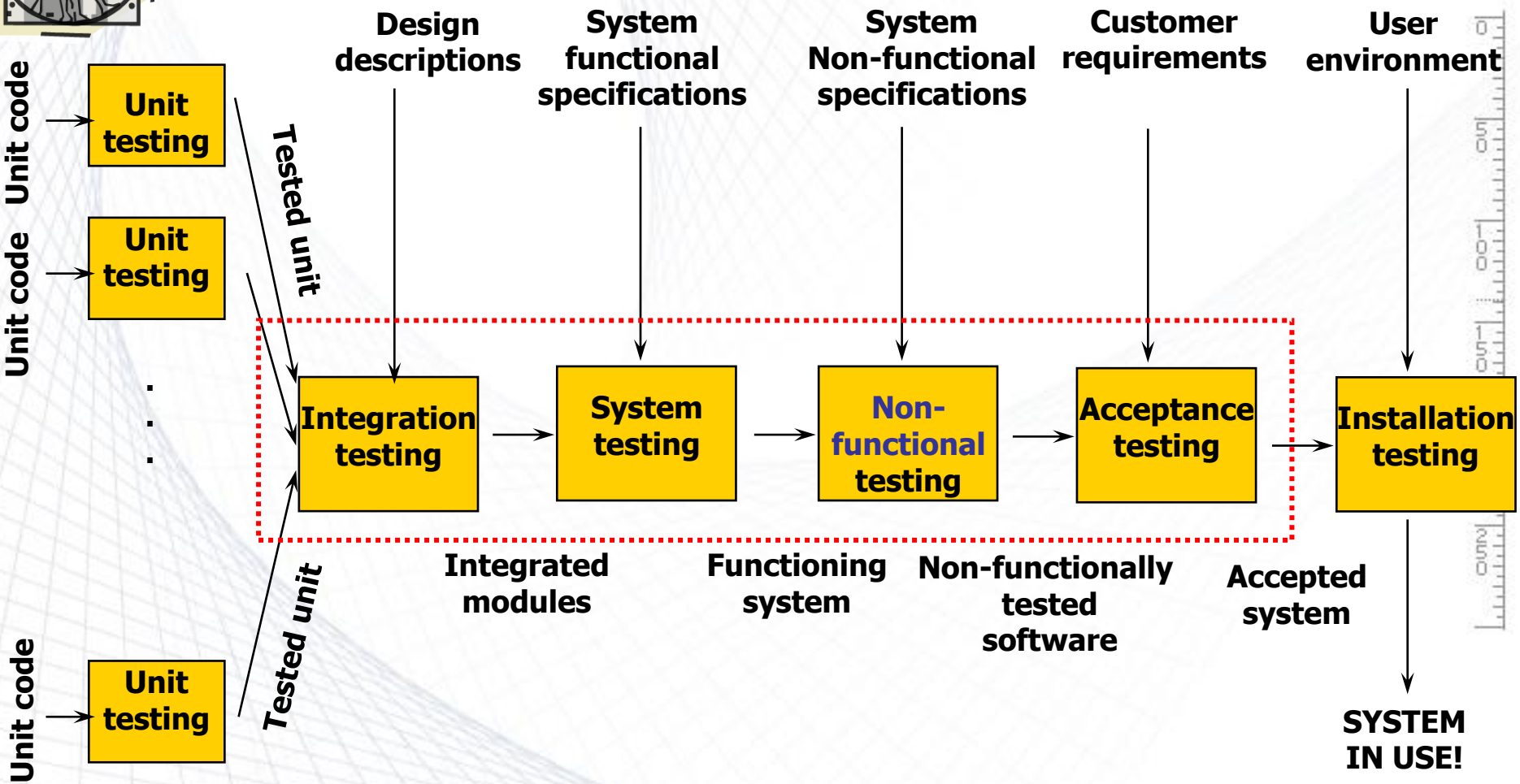


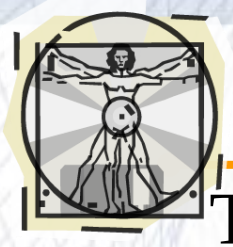
Test Levels/ Test Organization

- Since we can have failures in different level and stages of the software process, we need testing at different level and stages
  - **Unit testing**
    - Testing of individual components
  - **Integration testing**
    - Testing to expose problems due to combination of components
  - **System testing**
    - Testing the complete system prior to delivery
  - **Acceptance testing**
    - System level testing from users' perspective



Test Process – Typical Example





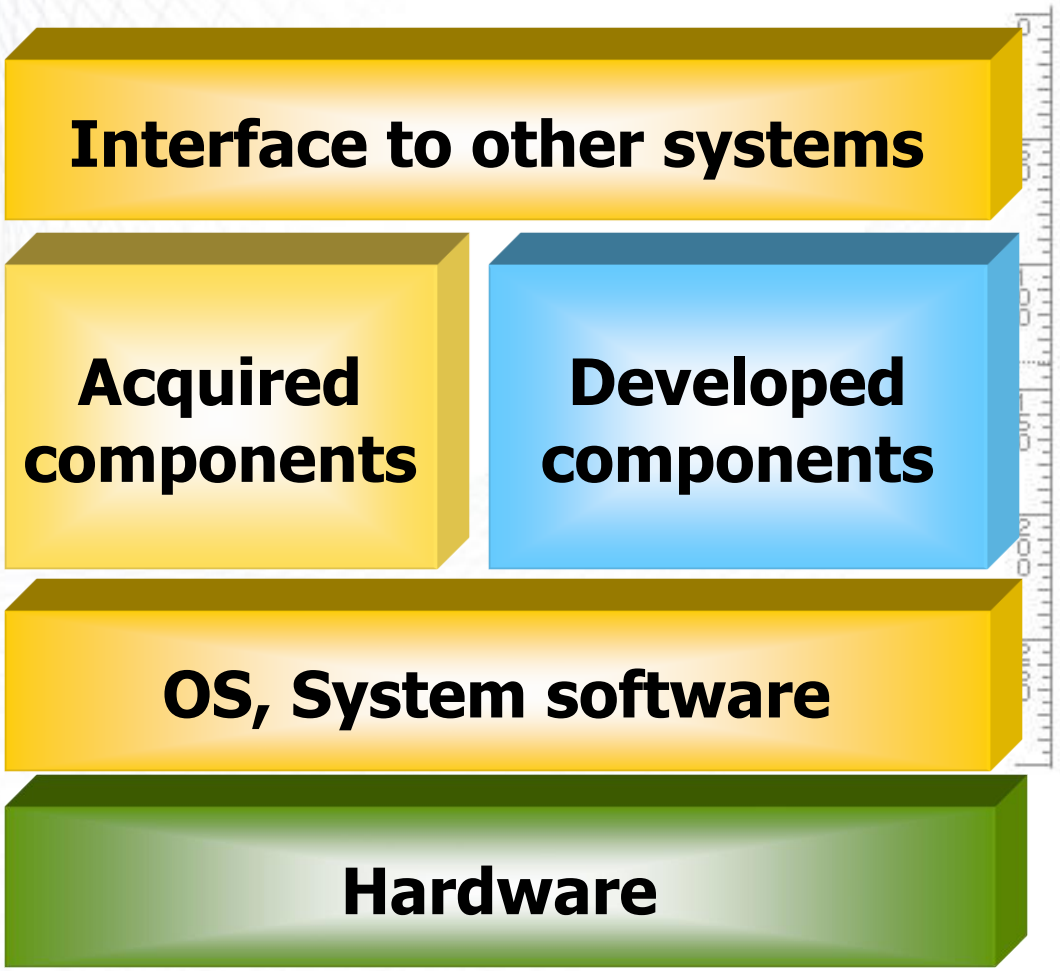
Test Management

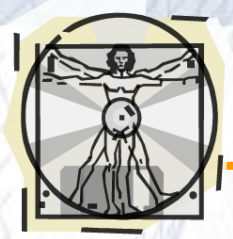
The procedure for preparing integration and system testing involves:

1. Estimate the number of new test cases needed for the current release
2. Allocate the number of new test cases among the subsystems to be tested (system level)
3. Allocate the number of new test cases for each subsystem among its new components (component level)
4. Specify and document the new test cases
5. Adding the new test cases to the ones already available (from unit testing and/or from a previous release)

Test Case Allocation

- Allocate the bulk of the test cases to the developed product itself
- Don't forget to test the rest, too
- Rule of thumb:
  - %80 dev system
  - %20 the rest

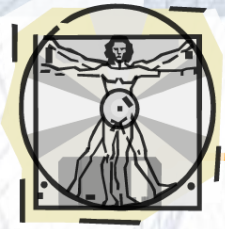




Test Procedures

- Test procedure is a controller program that sets up environmental conditions and invokes randomly selected test cases at random times

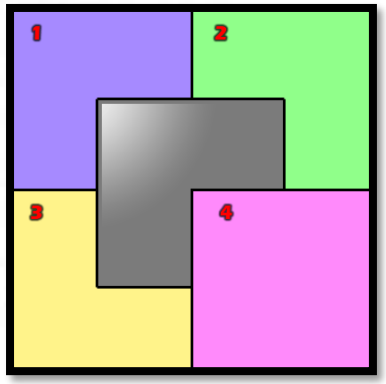
- Prepare one test procedure for the system or one test procedure for each operational mode (e.g. based on user type, time of day, etc.)



Test Time Allocation

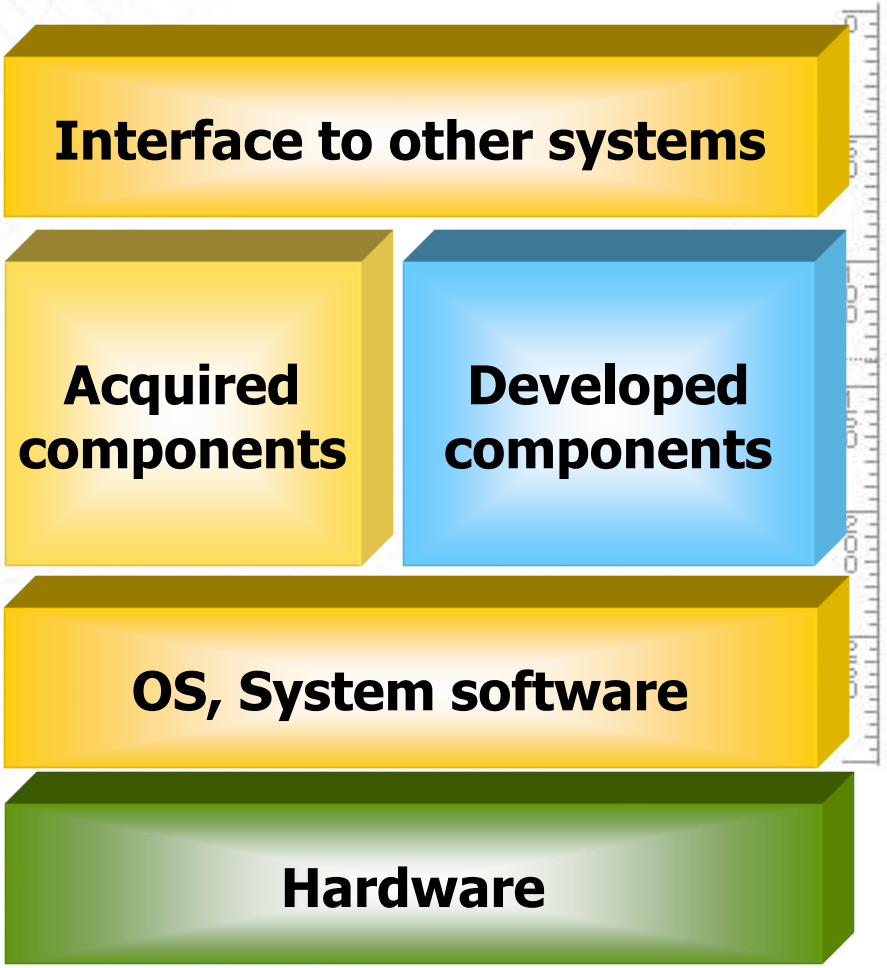
Allocate test time:

1) Among *subsystems* to be tested
2) Among *unit*, *regression* and *integration test* for each subsystem
3) Among *operational modes* if necessary



Test Time Allocation

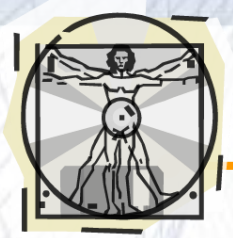
- Various systems involved get their share in test time
- Allocate time to interface to other systems
- Allocate time to the rest in the same proportion that test cases were allocated
- Rule of thumb:
  - %80 developed system
  - %20 the rest



UNIVERSITY OF CALGARY

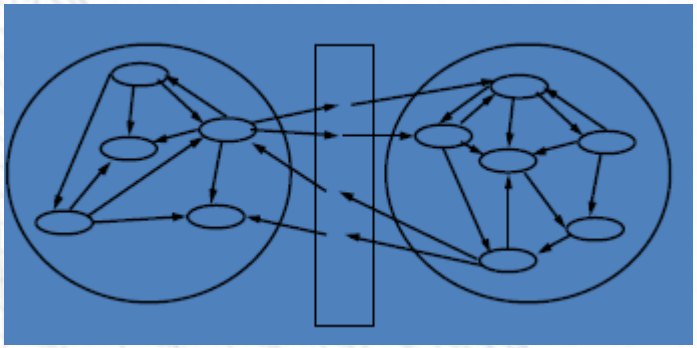
Section 2

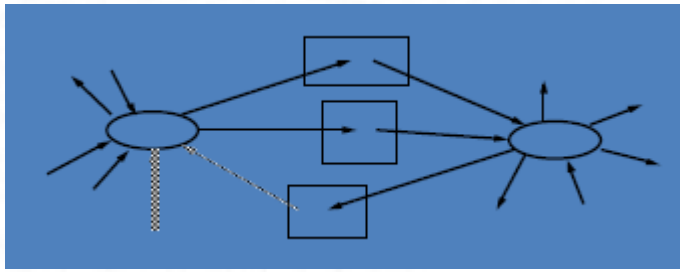
Integration Testing

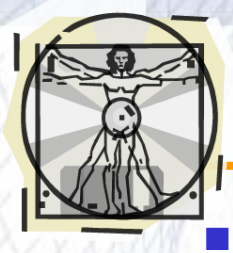


Integration Testing

- The entire system is viewed as a collection of subsystems (e.g. sets of classes) determined during the system and object design
- **Goal:** Test all interfaces between subsystems and the interaction of subsystems
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration

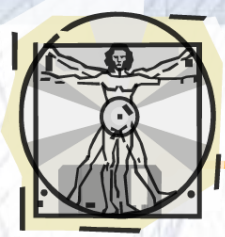




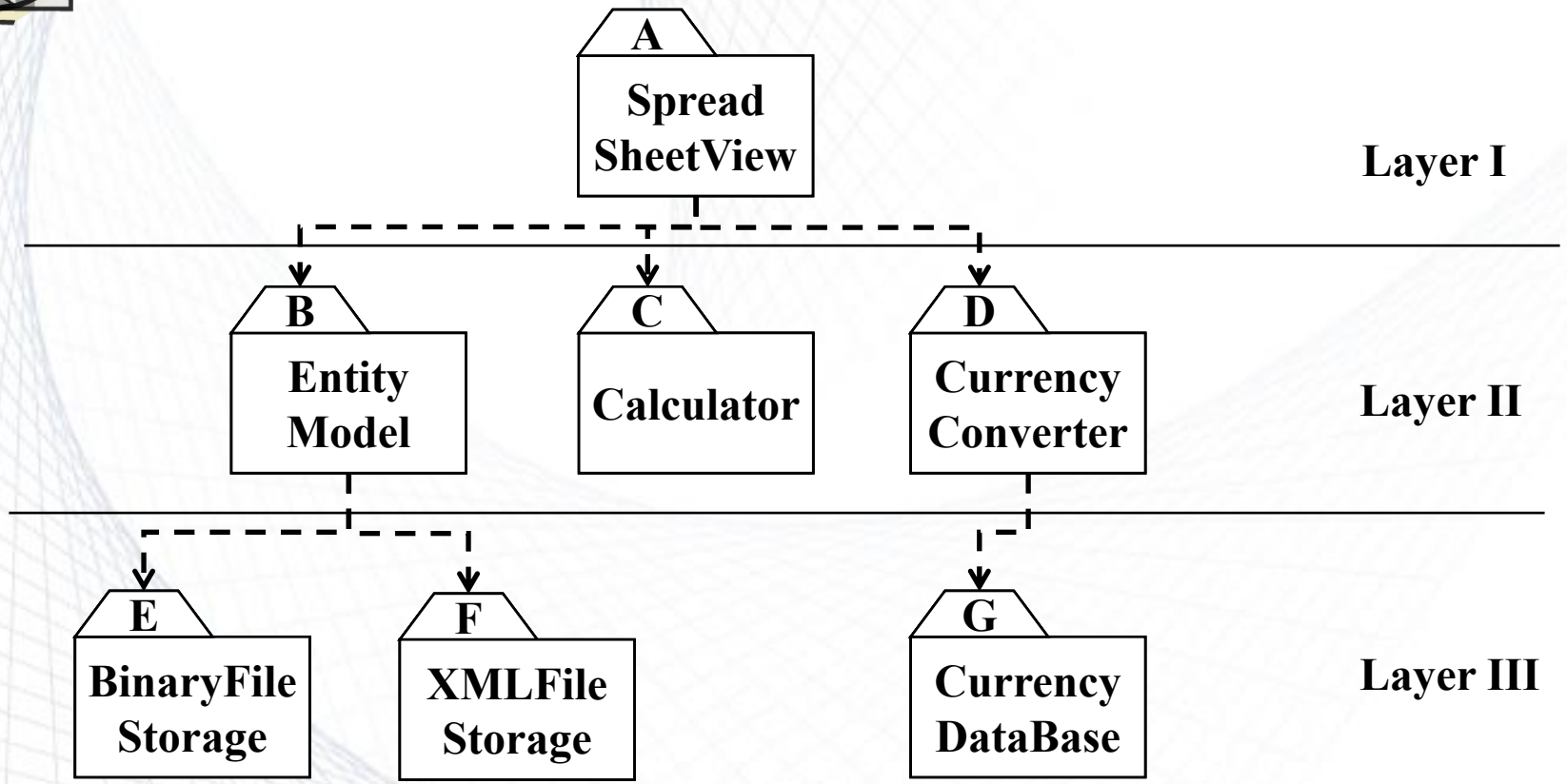


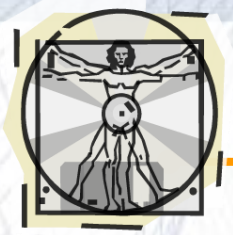
Why Integration Testing?

- Testing more than one (unit-tested) component
- Many failures result from faults in the interaction of components
  - What the set can perform that is not possible individually
- Testing communication between components
- Often many off-the-shelf components and/or libraries are used that cannot be unit tested
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive



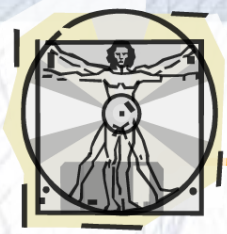
Example: A 3-Layer-Design





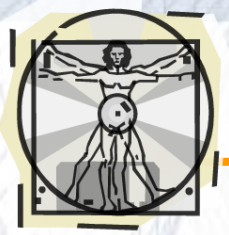
Integration Testing Strategy

- Integration testing strategy determines the order in which the subsystems are selected for testing and integration
- Decomposition-based integration
  - top–down
  - bottom–up
  - sandwich
  - and the vividly named "big bang"
- Call Graph–based integration (outside of course scope)

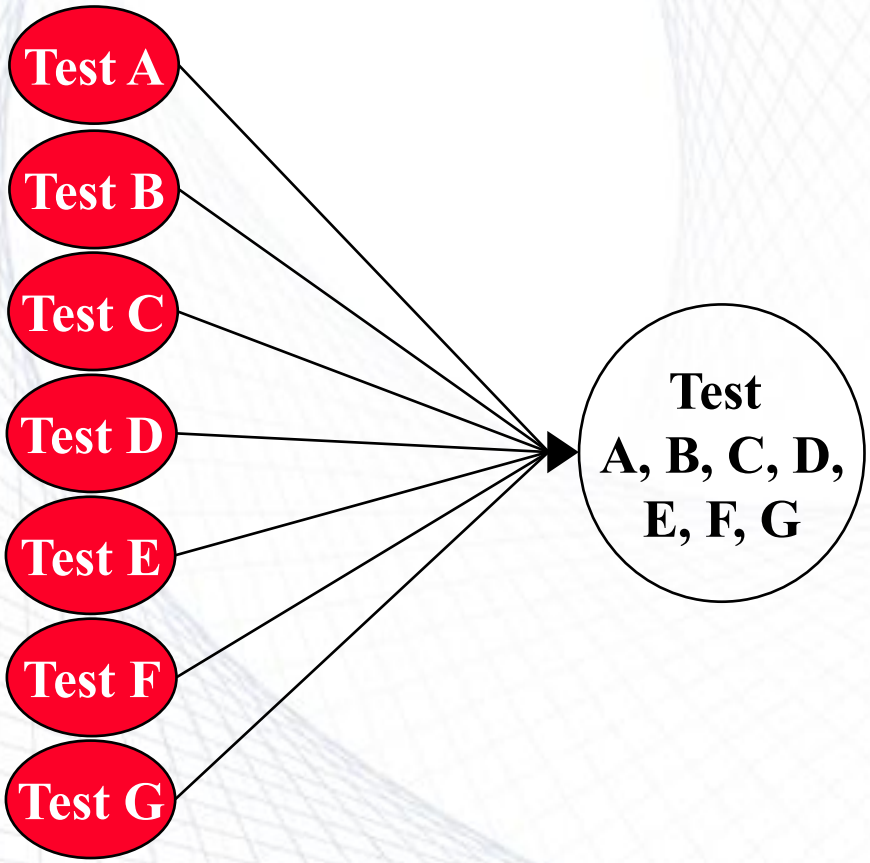


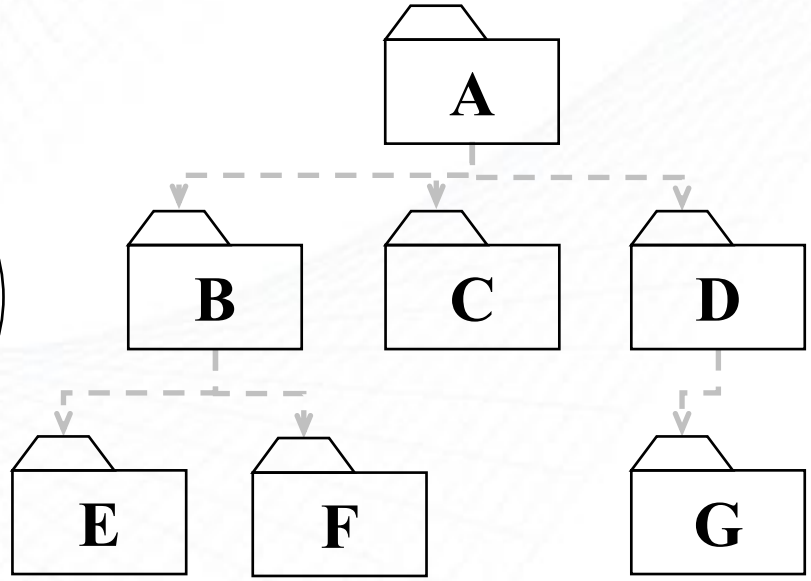
Big-Bang Integration

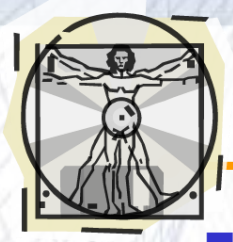
- Units completed and tested independently
- Integrated all at once
  - Quick and cheap (no stubs, mocks, drivers)
  - Faults
    - Later are discovered
    - More are found
    - More expensive to repair
- Most commonly used approach



Big-Bang Integration







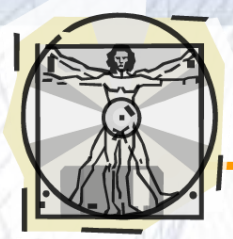
Big-Bang Integration

- In theory
  - if we have already tested units why not just combine them all at once? Wouldn't this save time?
  - based on false assumption of no interaction faults
- In practice
  - takes longer to **locate and fix** faults
  - re-testing after fixes (i.e. regression) is more extensive

Incremental Integration

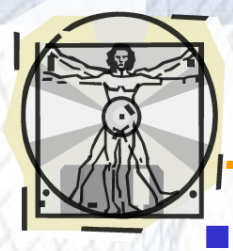
- Baseline 0: unit tested component
- Baseline 1: two components
- Baseline 2: three components, etc.

- Advantages
  - easier fault location and fix
  - easier recovery from disaster / problems



Bottom-up Integration

- Modules integrated in clusters as desired
- Shows feasibility of modules early on
- Emphasis on functionality and performance
  - Usually, test stubs are not needed
  - Errors in critical modules are found early
  - Many modules must be integrated before a working program is available
  - Interface errors are discovered late

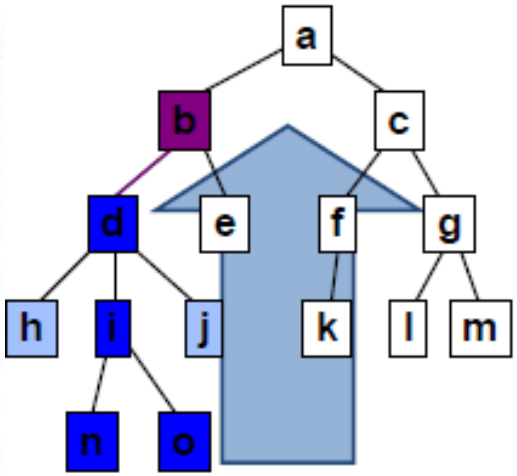


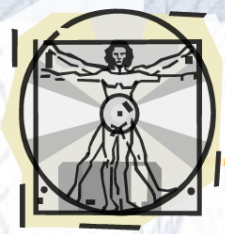
Bottom-up Integration

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems that call the previously tested subsystems are tested
- This is repeated until all subsystems are included
- Drivers are needed

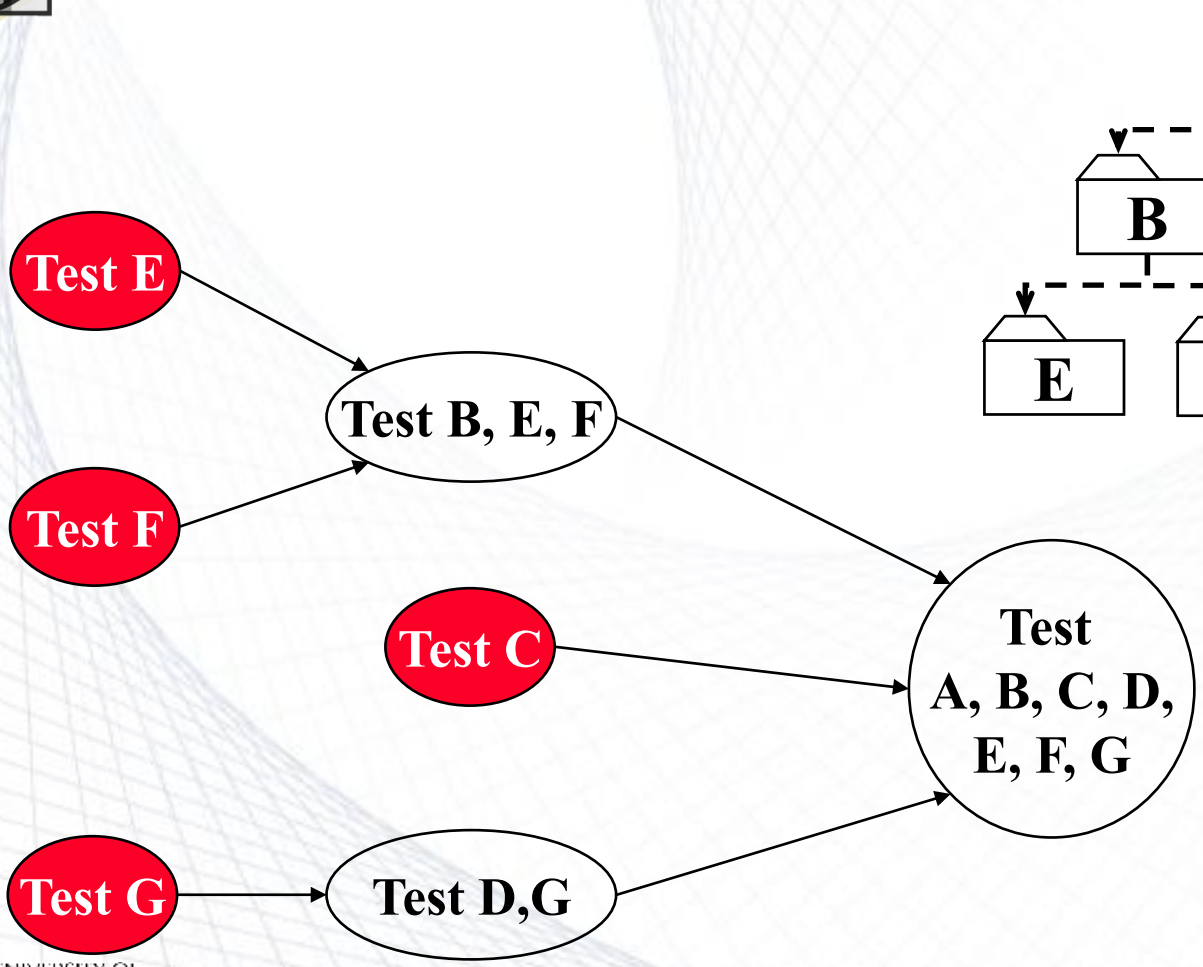
Baselines:

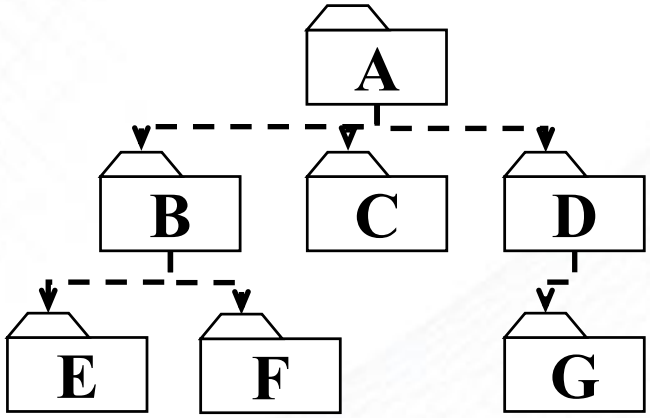
baseline 0: component n
baseline 1: n + i
baseline 2: n + i + o
baseline 3: n + i + o + d
etc.

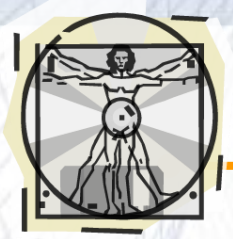




Bottom-up Integration

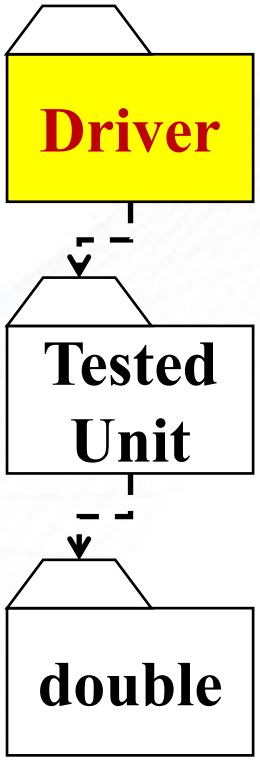


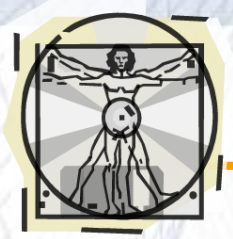




Drivers

- **Driver:**
  - A component, that **calls** the TestedUnit
  - Controls the test cases
  - invoke baseline
  - send any data baseline expects
  - receive any data baseline produces



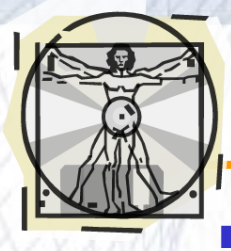


Bottom-Up Integration

- Pros
  - Visibility of detail
  - Fault localization
- Cons
  - Tests the most important subsystem (user interface) last
  - Drivers are needed

Top-down Integration

- The control program is tested first
- Modules are integrated one at a time
- Major emphasis is on interface testing
  - Interface errors are discovered early
  - Forms a basic early prototype
  - Test stubs are needed
  - Errors in low levels are found late

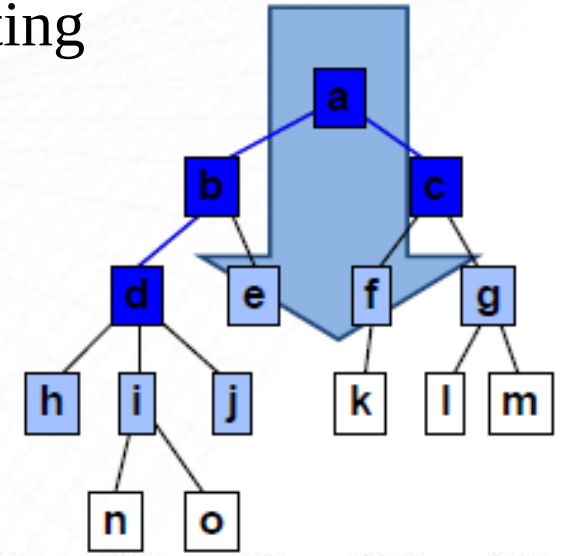


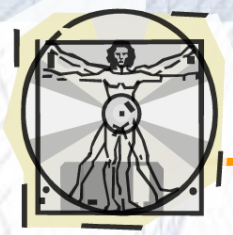
Top-down Integration

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Test doubles are needed to do the testing

Baselines:

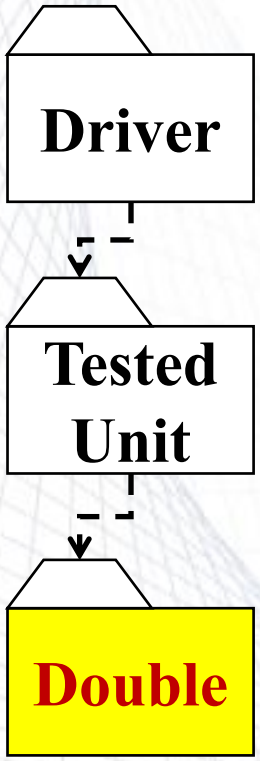
baseline 0: component a
baseline 1: a + b
baseline 2: a + b + c
baseline 3: a + b + c + d
etc.





Test Doubles (Mocks)

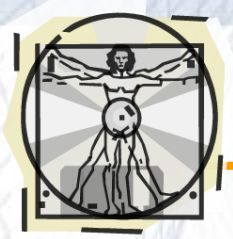
- A test double is an object that can stand in for a real object in a test
- Test doubles are sometimes all commonly referred to as "mocks"



We have already learnt this!

Similar to how a stunt double stands in for an actor in a movie

The term "test double" was coined by Gerard Meszaros in the book xUnit Test Patterns. You can find more information about test doubles in the book, or on the book's website.



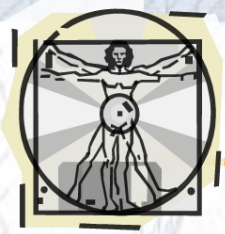
Top-down Integration

Pros

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed
- Critical control structure tested first and most often
- Can demonstrate system early (show working menus)

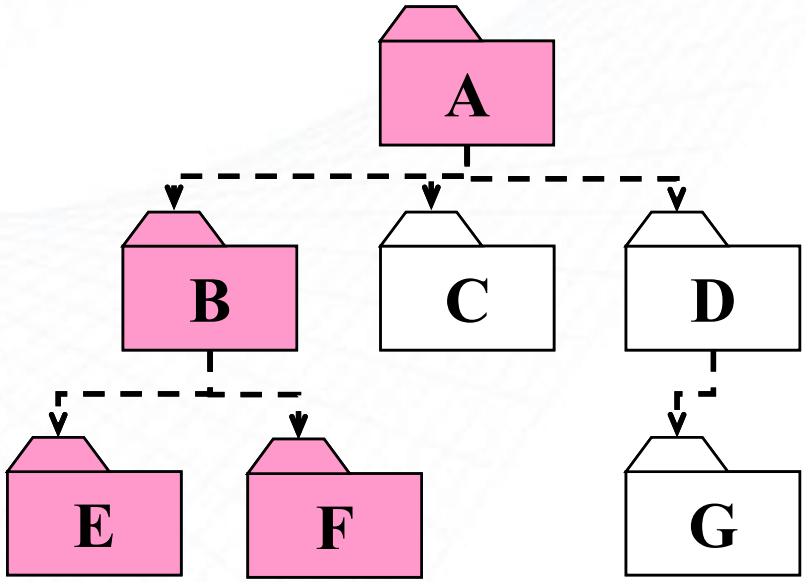
Cons

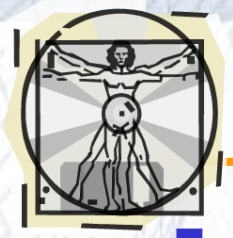
- Large number of doubles/mocks may be required
- Writing doubles/mocks is difficult, especially if the lowest level of the system contains many classes/methods
- Details left until last



Sandwich Testing Strategy

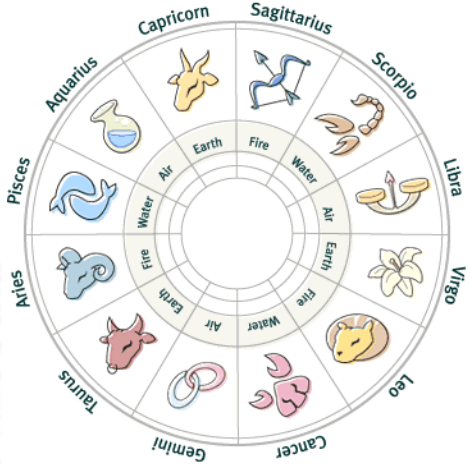
- Combines top-down with bottom-up strategy
- If we think about it in terms of the decomposition tree, we are really only doing big bang integration on a subtree
- Advantages:
  - Better fault localization
  - Less stubs
  - Can be done in parallel

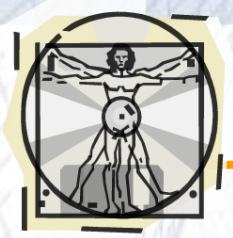




Example: Integration Strategies

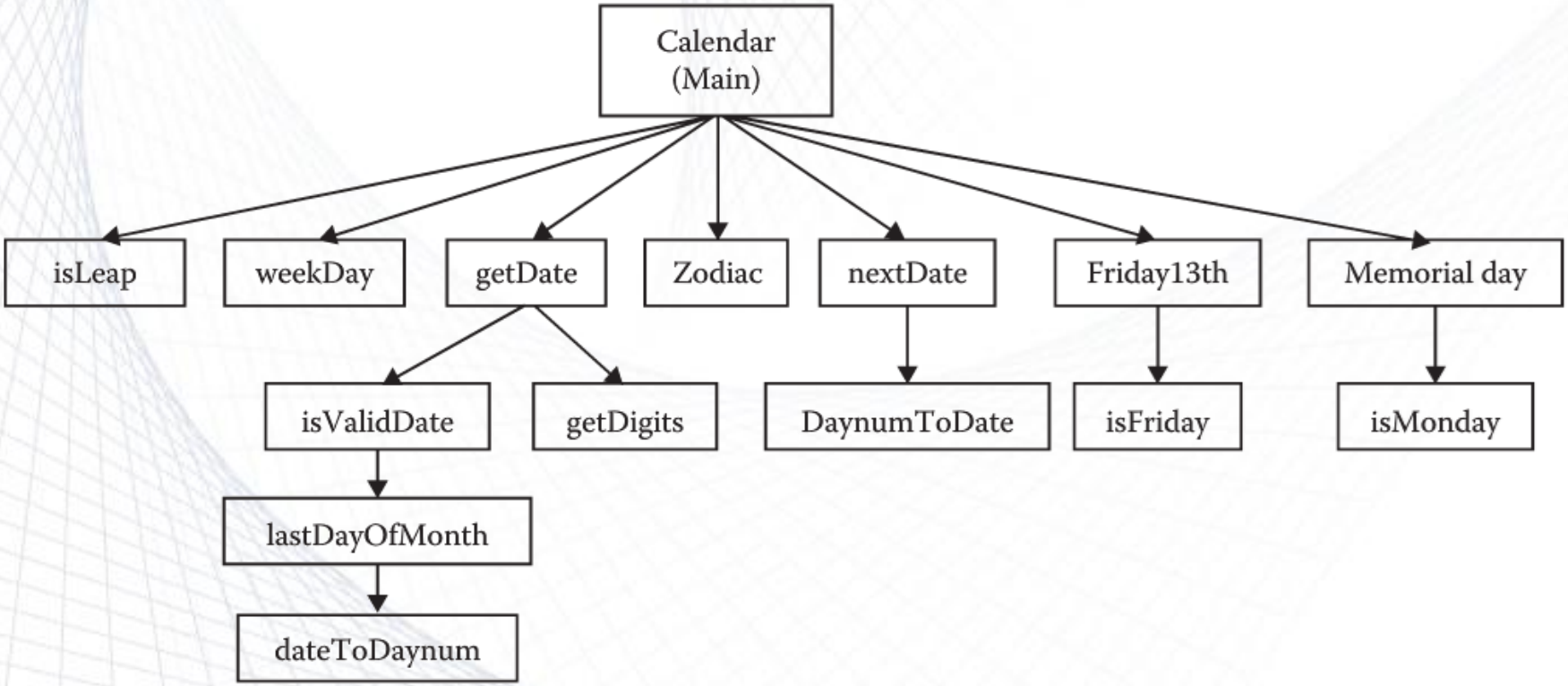
- The Calendar program acquires a date in the form mm, dd, yyyy, and provides the following functional capabilities:
  - The date of the next day
  - The day of the week corresponding to the date (i.e. Monday, Tuesday, etc.)
  - The zodiac sign of the date
  - The most recent year in which Memorial Day was celebrated on Monday
  - The most recent Friday the 13th

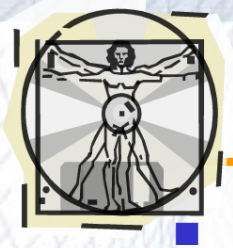




Functional Decomposition

Calendar program

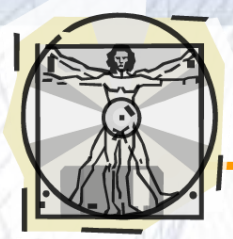




Top–Down Integration

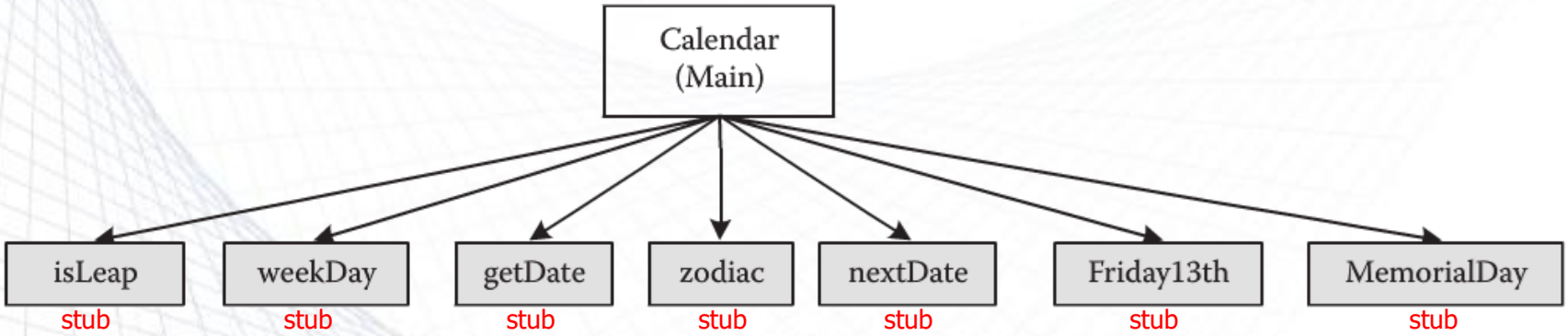
- If we perform top–down integration testing for the Calendar program, the first step is developing stubs for all the units called by the main program — isLeap, weekDay, getDate, Zodiac, nextDate, Friday13th, and memorialDay

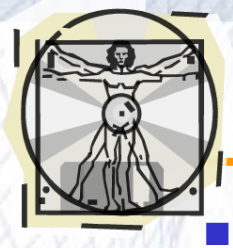
- In a stub for any unit, we hard code correct response to the request from the calling/invoking unit
- E.g., in the stub for Zodiac, if the main program calls Zodiac with the date digits [05, 27, 2017], the zodiacStub would return “Gemini”



Top–Down: First Step

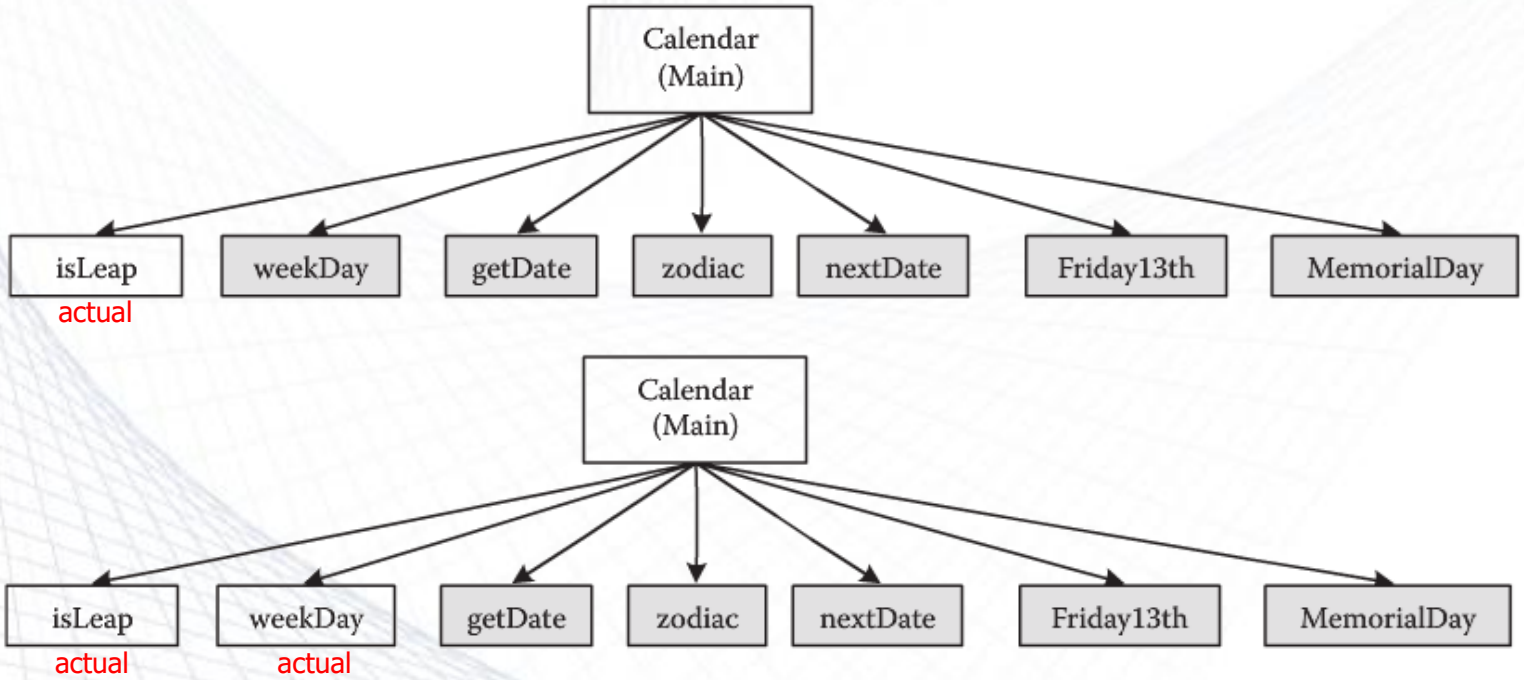
- The gray-shaded units are all stubs. The goal of the first step is to check that the main program functionality is correct

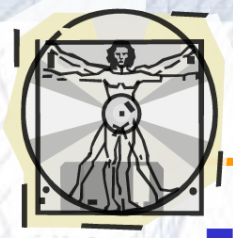




Top–Down: Next 2 Steps

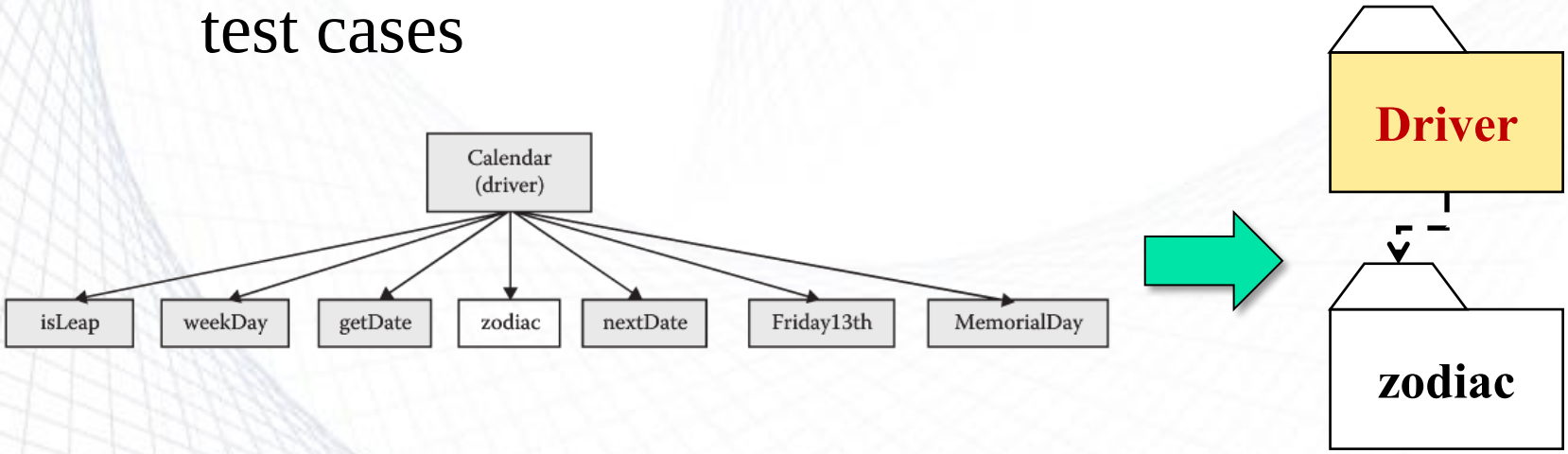
- The "theory" of top–down integration is that, as stubs are replaced one at a time, if there is a problem, it must be with the interface to the most recently replaced stub



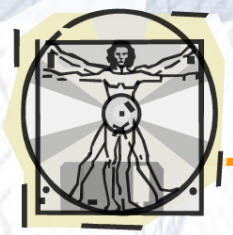


Bottom–Up Integration

- Bottom–up integration begins with the leaves of the decomposition tree, and use a driver version of the unit that would normally call it to provide it with test cases

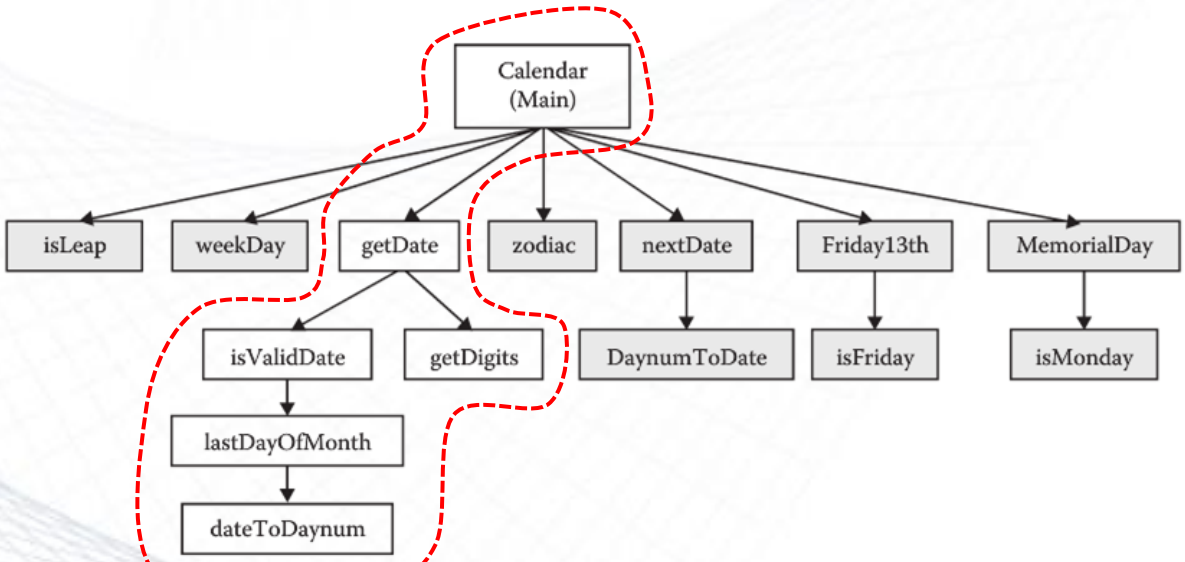


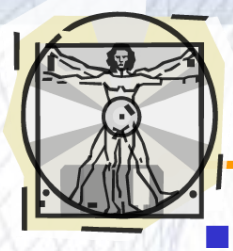
Bottom–up integration for zodioc



Sandwich Integration

- Sandwich integration: doing big-bang integration on a subtree
- There will be less stub and driver development effort, but this will be offset to some extent by the added difficulty of fault isolation that is a consequence of big bang integration





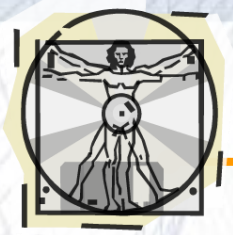
Continuous Integration

- Continuous Integration (CI) is a development practice that requires developers check-in their code into a shared repository several times a day. Each check-in is then verified by an automated build

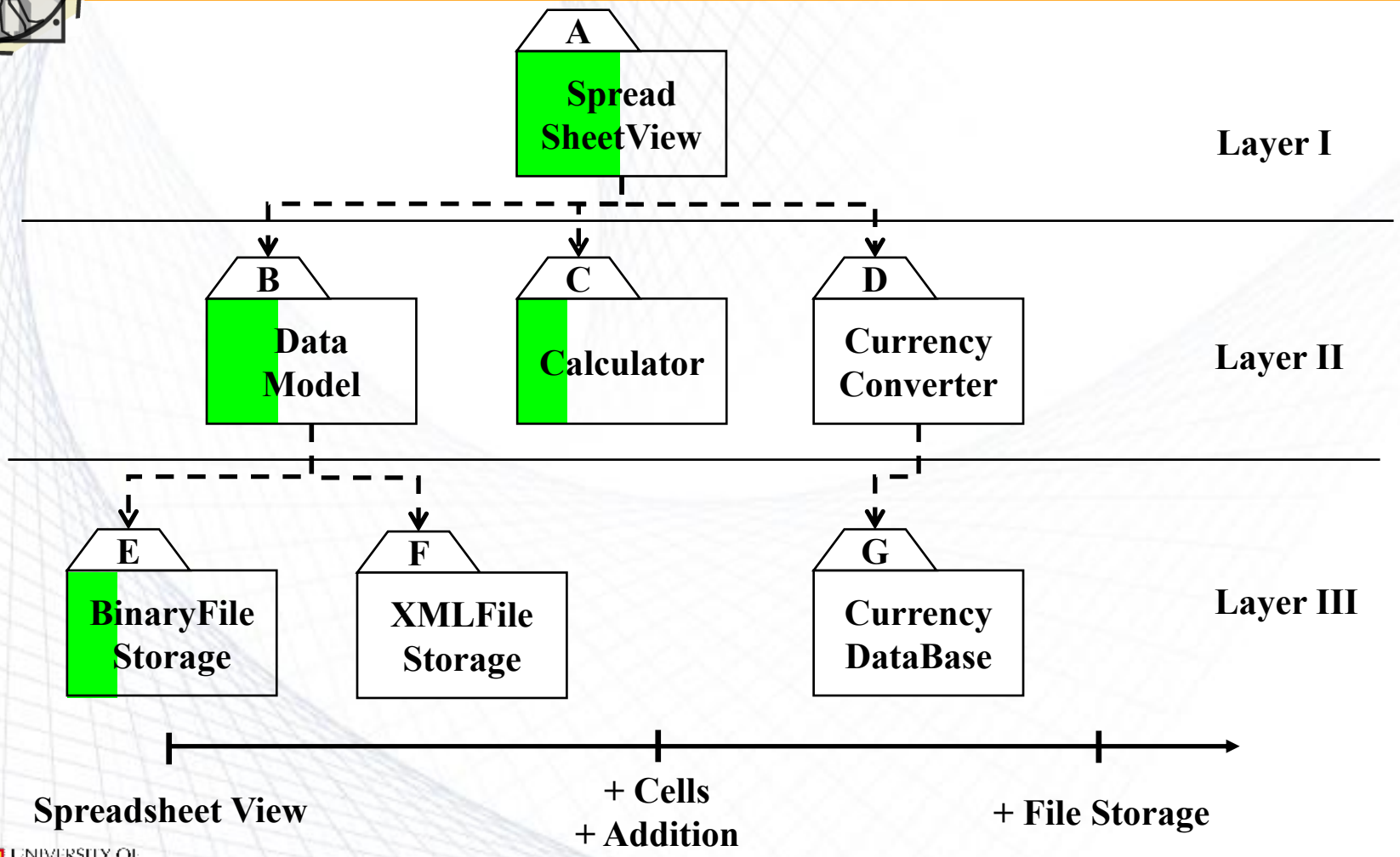
- Continuous build:
  - Build from day one
  - Test from day one
  - Integrate from day one
  - Assessment from day one

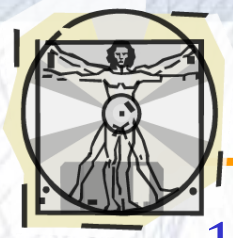
⇒ System is always runnable

- Requires integrated tool support:
  - Continuous build server
  - Automated tests
  - Configuration management
  - Issue tracking
  - Quality assessment tracking



Continuous Testing Strategy





Steps in Integration Testing

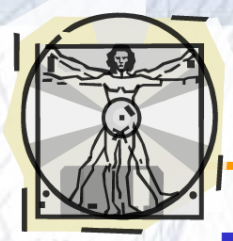
1. Based on the integration strategy, select a component to be tested; Unit test all the classes in the component
2. Put selected components together; do any preliminary fix-up necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements: Define test cases that exercise all use-cases with the selected component
4. Test subsystem decomposition: Define test cases that exercise all dependencies
5. Keep records of the test cases and testing activities
6. Repeat steps 1 to 5 until the full system is tested

SCHULICH School of Engineering

UNIVERSITY OF CALGARY

Section 3

System Testing

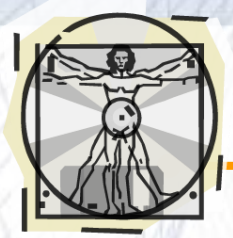


System Testing

Let's watch a video

- Last integration step
- Functional
  - Functional requirements and requirements-based testing
- Non-functional
  - As important as functional requirements
  - Often poorly specified
  - Must be tested
- Often done by an independent test group

https://www.youtube.com/watch?v=CC0He-SS-Do



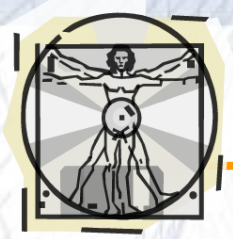
System vs. Integration Testing

#	System Testing	Integration Testing
1.	Check the system as a whole	Check the interfacing between the inter-connected components
2.	Is performed after integration testing	Is performed after unit testing
3.	Is carried out for performing both functional and non-functional testing (performance, usability etc.)	Is generally limited to functional aspects of the integrated components
4.	Since the testing is limited to evaluation of functional requirements, hence, it includes black-box testing techniques only	Since the interfacing logic is required to perform this testing, hence, it requires white/grey box testing techniques along with black-box techniques
5.	Different type of system testing are: functional testing, performance testing, usability testing, reliability testing, security testing, scalability testing, etc.	Different approaches of performing integration testing are: top down, bottom up, big bang, sandwich, etc.

Are We All Done?

- Do not forget non-functionals during system testing!

- The nonfunctional aspects of a software system (e.g., performance, security, compatibility, concurrency and usability) may require considerable effort to test and perfect



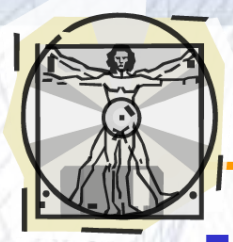
What Are Non-functionals?

- Main non-functions for hardware/software systems are:
  - Performance
  - Compatibility
  - Security
  - Usability
  - Concurrency

Question:

How do we usually handle non-functions?

- **A good architectural design of the system can handle non-functions**
- **A good post release analysis will help improve handling the non-functions**



Documenting Non-functions

- Nonfunctional requirements are usually documented in two ways:
- A system-wide specification is created that defines nonfunctional requirements for all ***use-cases*** in the system **Example:** "The user interface of the Web system must be compatible with Firefox 50.x or higher and Microsoft Internet Explorer 12.x or higher"
- Each requirement description contains a section titled "Nonfunctional Requirements," which documents any specific nonfunctional needs of that particular requirement that differ from the system-wide specifications

SCHULICH School of Engineering

UNIVERSITY OF CALGARY

Section 4

Acceptance Testing

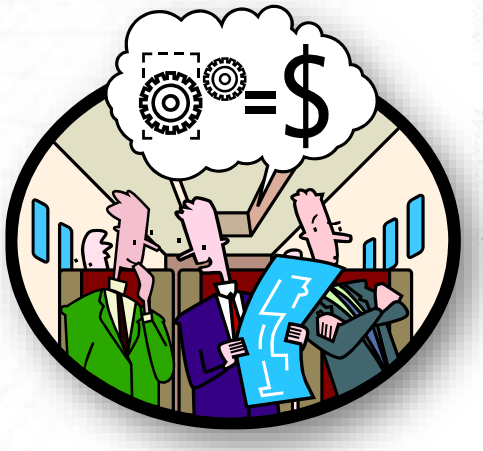
Acceptance Testing

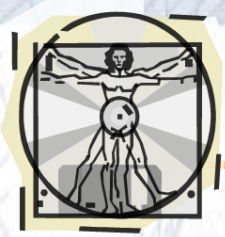
- Final stage of validation
  - Customer (user) should perform or be closely involved
  - Customer can perform any test they wish, usually based on their business processes
  - Final user sign-off
- **Goal:** Demonstrate system is ready for operational use
- Acceptance testing motto:

If you don't have patience to test the system, the system will surely test your patience

Why Customer Involvement?

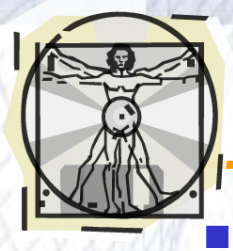
- Users know:
  - What really happens in business situations
  - Complexity of business relationships
  - How users would do their work using the system
  - Variants to standard tasks (e.g. country-specific)
  - Examples of real cases
  - How to identify sensible work-arounds
- Benefit for customer:
  - Detailed understanding of the new system
- Disadvantage for developers:
  - Potential customers might get discouraged





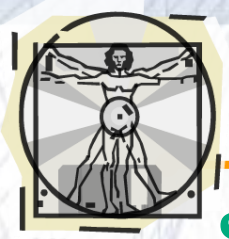
External Function Test: Alpha

- In-house testers mimic the end use of the system
- Sometimes known as an ***Alpha*** test
- Black-box test
- Verify the system correctly implements specified functions



External Function Test: Beta

- Environment reflects end use
  - Includes hardware, database size, system complexity, external factors
- Also known as ***Beta*** testing
- Completed system tested by end users
- More realistic test usage than 'system testing' phase
- Validates system with user expectations
- Determine if the system is ready for deployment
- Can more accurately test nonfunctionals (performance, security etc.)



Alpha and Beta Tests

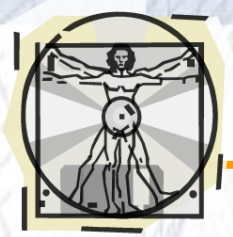
Similarities

- Testing by [potential] customers or representatives of your market
- When software is stable
- Use the product in a realistic way in its operational environment
- Give comments back on the product
  - Failures found
  - How the product meets their expectations
  - Improvement / enhancement suggestions?

Alpha and Beta Tests

Differences

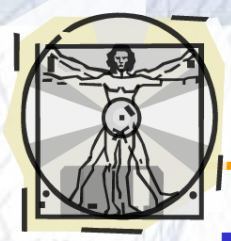
- Alpha testing
  - Simulated/actual operational testing at an in-house site
  - Client uses the software at the developer's environment
  - Software used in a controlled setting, with the developer always ready to fix bugs
- Beta testing
  - Conducted at client's environment
  - Software gets a realistic workout in target environment
  - There will be time lag between reporting a bug and fixing it



Post-Release Activities

- **Do we conduct "testing" in post-release?**

- No. Post-release activity is usually called **quality assessment** or **maintenance** and includes techniques, such as
  - Reliability-Availability-Maintainability (RAM)
  - Fault Tree Analysis (FTA) ← **already covered**
  - Failure Modes and Effect Analysis (FMEA)
  - HAZOP
  - etc.



Summary

- Testing is still a black art, but many rules, best practices and heuristics are available
- Testing consists minimally of
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing

- Testing should be accompanied by quality assessment techniques
- Testing has its own lifecycle