
ICS 52: Introduction to Software Engineering

Winter Quarter 2004

Professor Richard N. Taylor

Lecture Notes

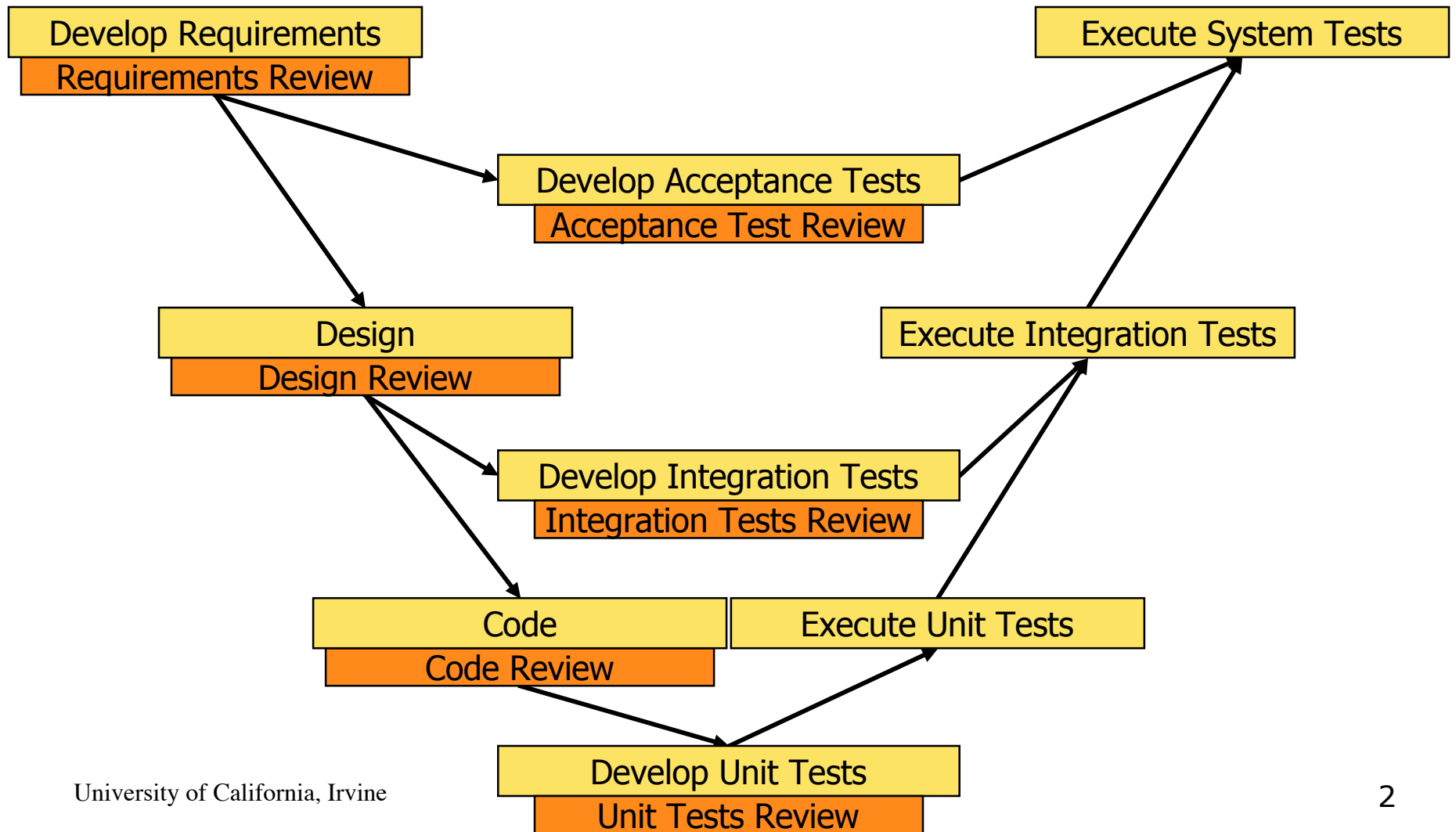
Week 7 Integration Testing and Implementation Issues

http://www.ics.uci.edu/~taylor/ICS_52_WQ04/syllabus.html



Copyright 2004, Richard N. Taylor. Duplication of course material for any commercial purpose without written permission is prohibited.

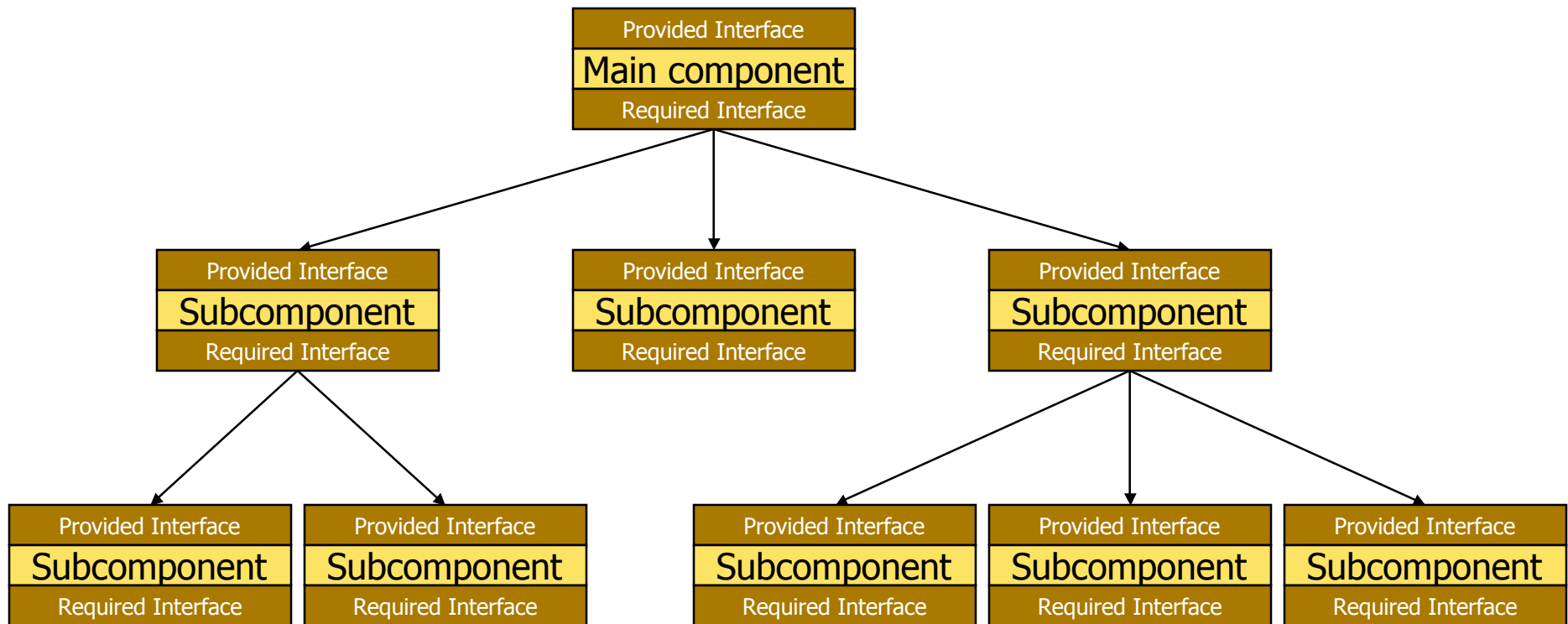
V-Model of Development and Testing



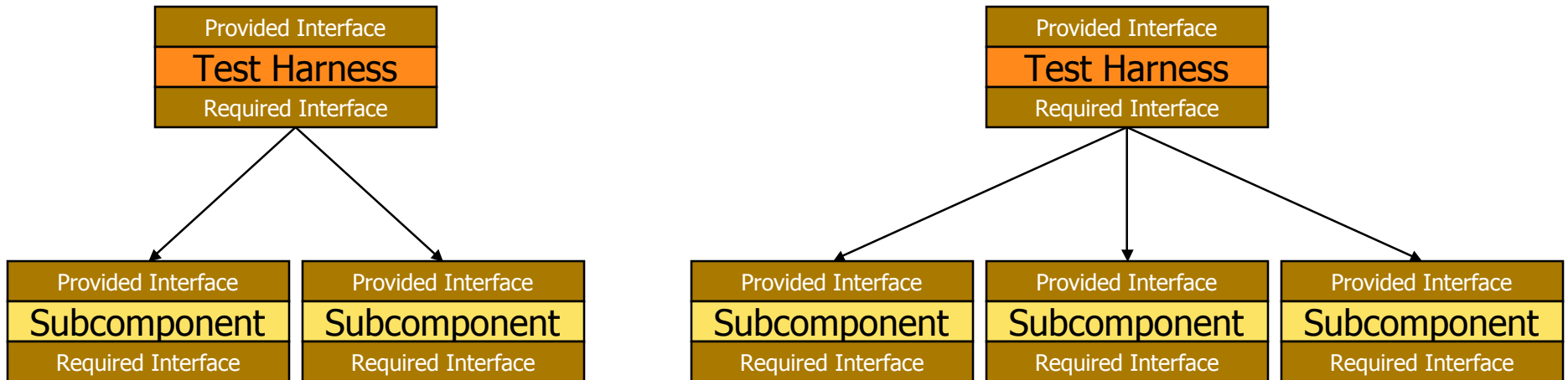
Integration Test Plan

- ◆ Ensures module implementations adhere to assumptions and interfaces as designed
 - Uncovering interactions that highlight problems with assumptions is difficult
- ◆ Approach
 - Combine more and more modules
 - Use USES hierarchy
 - » Work up from level zero
 - ◆ Use test harnesses to test each group of modules
 - » Work down from highest number
 - ◆ Use stubs as mockups to test each group of modules
- ◆ Can be done during implementation effort

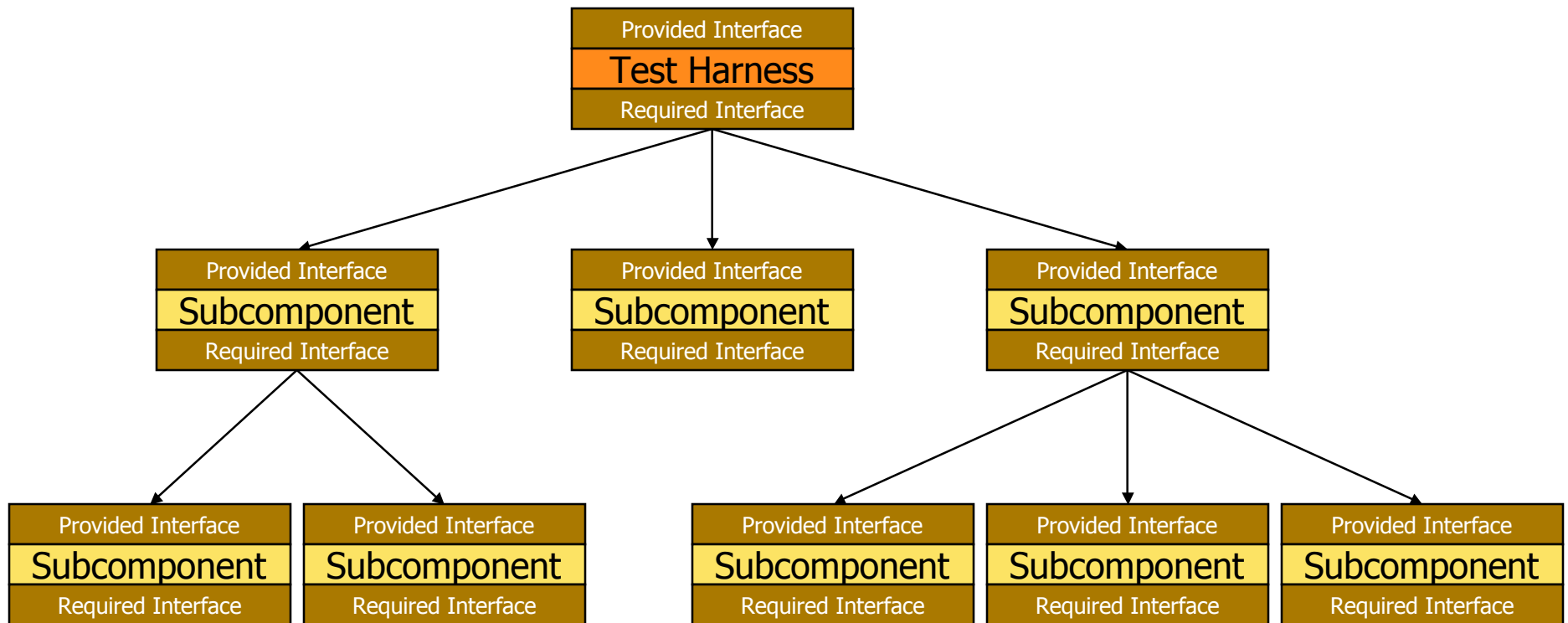
Integration Test Example



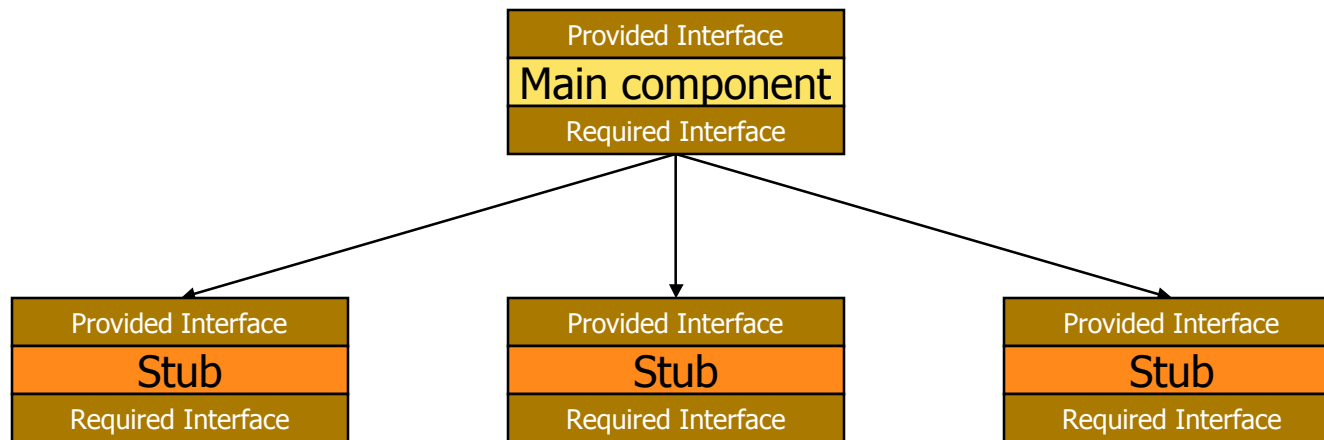
Test Harnesses



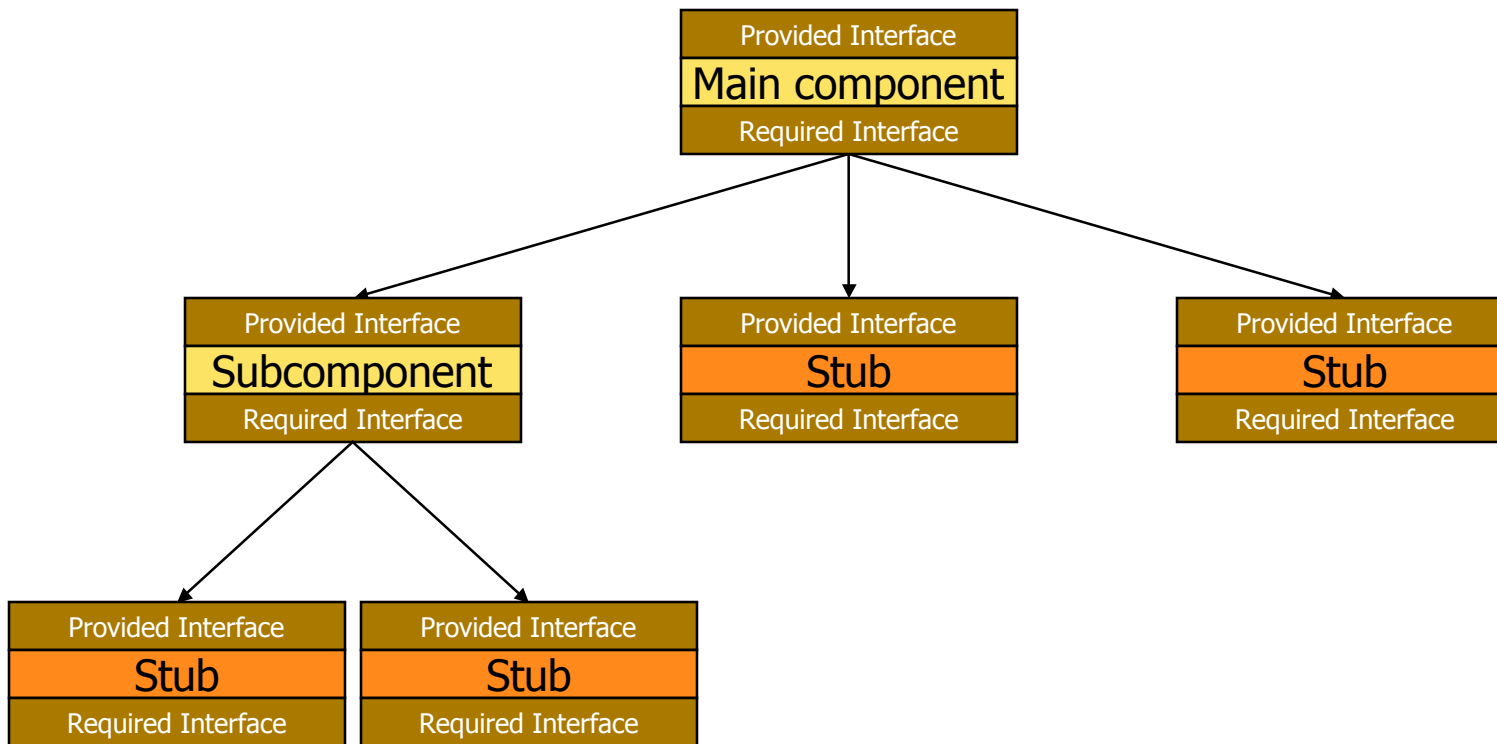
Test Harnesses



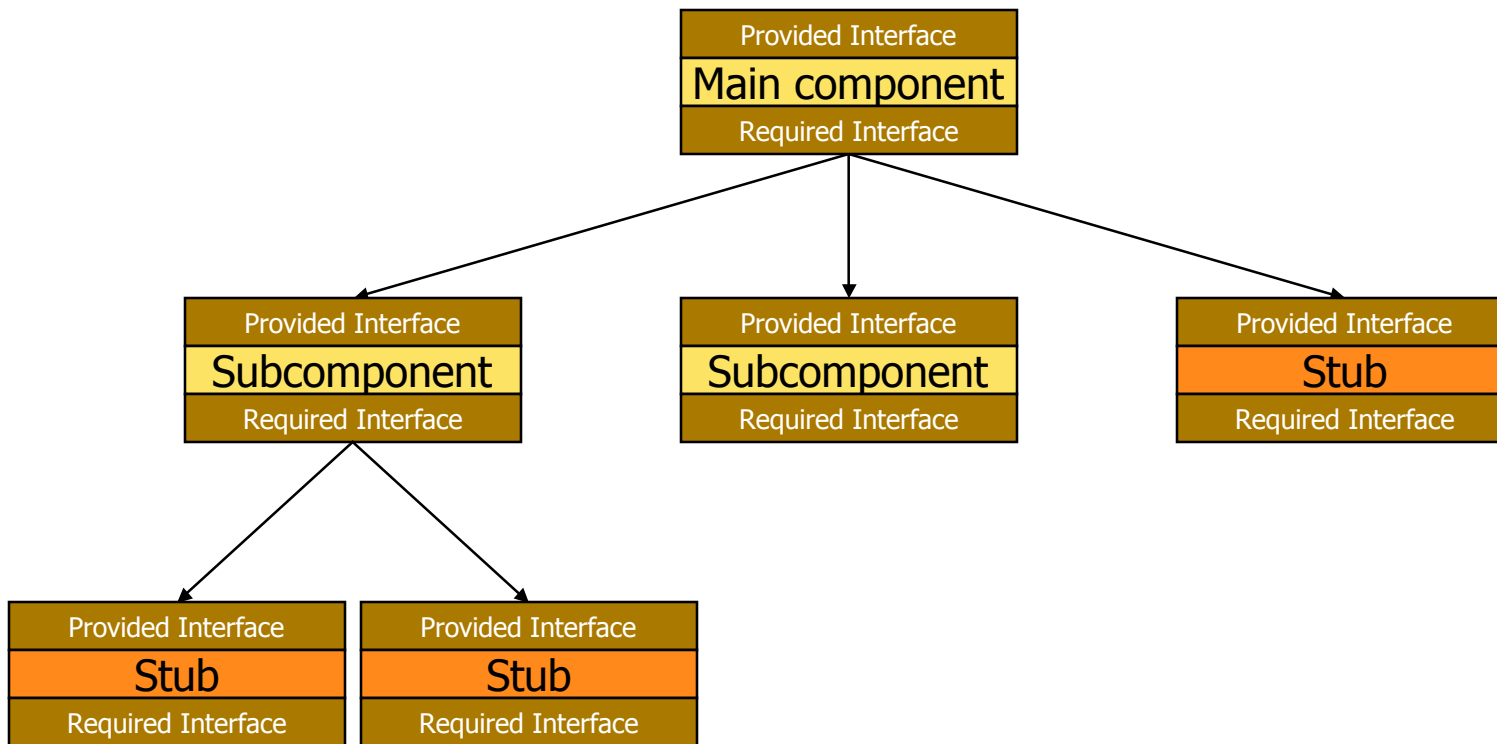
Stubs



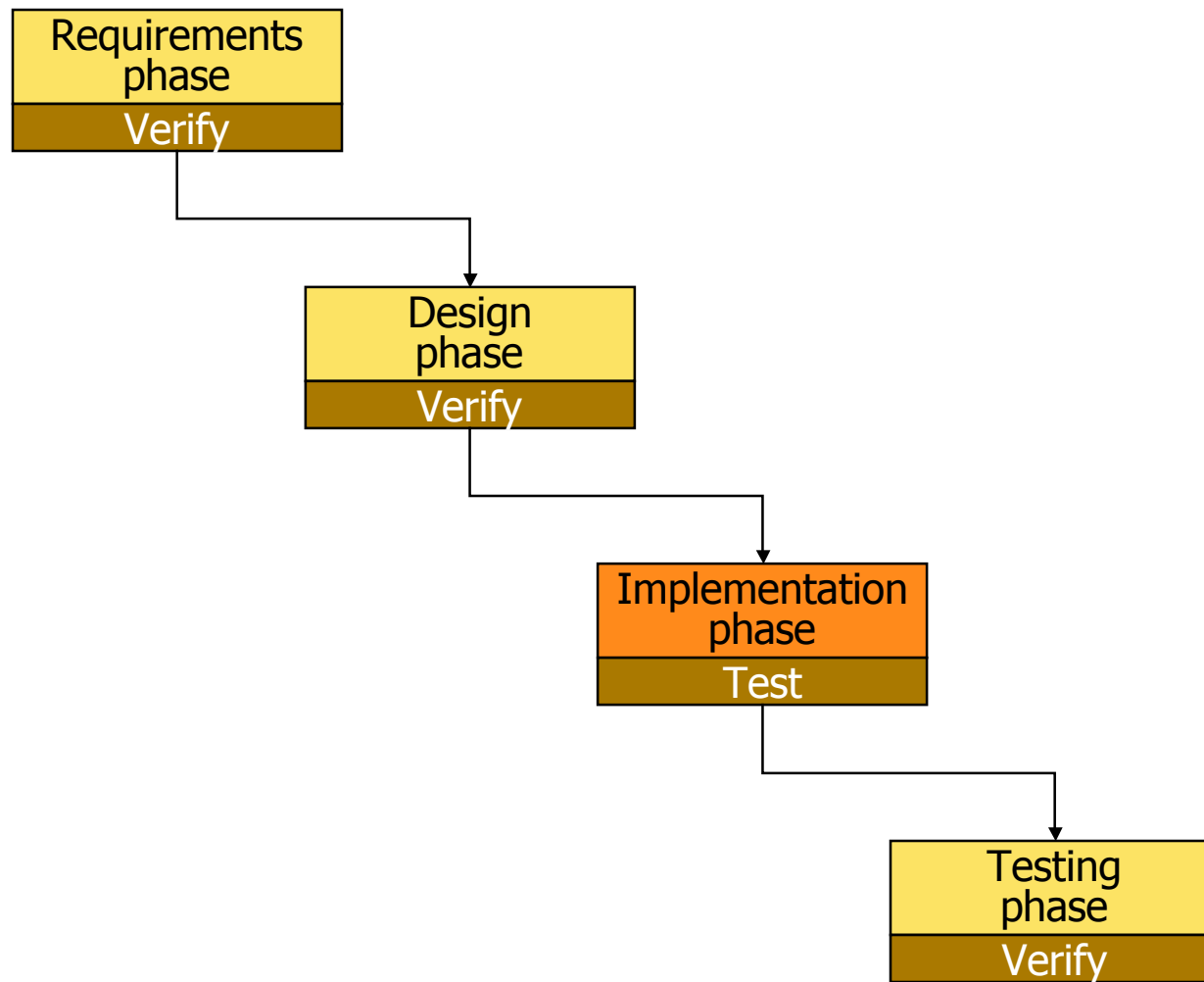
Stubs



Stubs

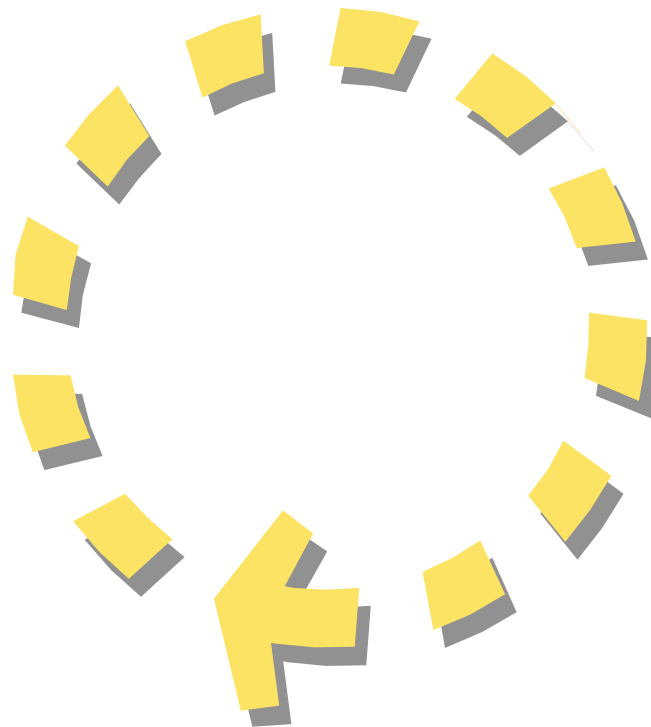


ICS 52 Life Cycle



Design/Implementation Interaction

Design
(previous lectures)



Implementation
(this lecture)

A Good Design...

- ◆ ...is half the implementation effort!
 - Rigor ensures all requirements are addressed
 - Separation of concerns
 - » Modularity allows work in isolation because components are independent of each other
 - » Abstraction allows work in isolation because interfaces guarantee that components will work together
 - Anticipation of change allows changes to be absorbed seamlessly
 - Generality allows components to be reused throughout the system
 - Incrementality allows the software to be developed with intermediate working results

A Bad Design...

- ◆ ...will never be implemented!
 - Lack of rigor leads to missing functionality
 - Separation of concerns
 - » Lack of modularity leads to conflicts among developers
 - » Lack of abstraction leads to massive integration problems (and headaches)
 - Lack of anticipation of change leads to redesigns and reimplementations
 - Lack of generality leads to “code bloat”
 - Lack of incrementality leads to a big-bang approach that is likely to “bomb”

From Design to Implementation

- o Choose a suitable implementation language
- o Establish coding conventions
- o Divide work effort
- o Implement
 - o Code
 - o Unit tests
 - o Code reviews
 - o Inspections
- o Perform integration tests

Choose a Suitable Language

- ◆ 4th Generation language
 - Databases
 - Visual Basic
 - Forms
- ◆ “Real” programming language
 - Java + Class Libraries
 - C++/C + STL (Standard Template Library)
 - Cobol
 - Fortran
- ◆ Assembly language
 - Machine specific

Choose a Suitable Language

- ◆ Maintain the design “picture”
 - Mapping of design elements onto implementation
 - Module inside versus outside
 - » Does the language enforce a boundary?
 - » Interfaces!
 - Explicit representation of uses relationship
 - » Just function calls?
- ◆ Error handling
 - Return values
 - Exceptions

Establish Coding Conventions

- ◆ Naming
 - Avoid confusing characters
 - » 1, l, L, o, O, 0, S, 5, G, 6
 - Avoid misleading names
 - Avoid names with similar meaning
 - Use capitalization wisely -- and consistently
- ◆ Code layout
 - White space / blank lines
 - Grouping
 - Alignment
 - Indentation
 - Parentheses

Divide Work Effort

- ◆ Assign different modules to different developers
 - Assignments can be incremental
 - Assignments change
 - » Illness
 - » New employees
 - » Employees who quit
 - » Schedule adjustments
 - » Star programmers
- ◆ Interfaces are tremendously important
 - “Contracts” among modules

Coding

- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY
- ◆ FIRST MAKE IT WORK CLEANLY

Code Optimizations

- ◆ Only make optimizations to a cleanly working module if absolutely necessary
 - Performance
 - Memory usage
- ◆ Isolate these optimizations
- ◆ Document these optimizations

Empirical evidence has proven that these optimizations are rarely needed and that if they are needed, they are only needed in a few critical places

Defensive Programming

- ◆ Make your code robust and reliable
 - Use assertions
 - Use tracing
 - Handle, do not ignore, exceptions
 - » Contain the damage caused
 - » Garbage in does not mean garbage out
 - Anticipate changes
 - Check return values
- ◆ Plan to be able to remove debugging aids in the final, deliverable version

Do not sacrifice any of these when facing a deadline

Comments

- ◆ Self documenting code does not exist!
 - Meaningful variable names, crisp code layout, and small and simple modules all help...
 - ...but they are not enough
- ◆ Every module needs a description of its purpose
- ◆ Every function needs a description of its purpose, input and output parameters, return values, and exceptions
- ◆ Every piece of code that remotely may need explanation should be explained

Unit Tests

- ◆ Developer tests the code just produced
 - Needs to ensure that the code functions properly before releasing it to the other developers
- ◆ Benefits
 - Knows the code best
 - Has easy access to the code
- ◆ Drawbacks
 - Bias
 - » “I trust my code”
 - » “I always write correct code”
 - Blind spots

Code Reviews (“Walk-throughs”)

- ◆ Developer presents the code to a small group of colleagues
 - Developer describes software
 - Developer describes how it works
 - » “Walks through the code”
 - Free-form commentary/questioning by colleagues
- ◆ Benefits
 - Many eyes, many minds
 - Effective
- ◆ Drawbacks
 - Can lead to problems between developer and colleagues

Inspections

- ◆ Developer presents the code to a small group of colleagues
 - Colleagues look for predefined types of errors
 - » Checklists
 - Colleagues read code beforehand
 - Moderator leads discussion
- ◆ Benefits
 - Avoids personal “attacks”
 - Effective
- ◆ Drawbacks
 - Only verifies code with respect to a predefined list of problem areas

Use the Principles

- ◆ Rigor and formality
- ◆ Separation of concerns
 - Modularity
 - Abstraction
- ◆ Anticipation of change
- ◆ Generality
- ◆ Incrementality