CS4125
SYSTEMS ANALYSIS
SPRING SEMESTER 2010-2011

J.J. Collins
Dept of CSIS
University of Limerick

Lecture B (W1L2)

Characteristics of “good” software

Good Software is .....

- Good systems should be usable, reliable (bug free), affordable, flexible, and available.
  - These are quality attributes that are not specific to software.
- Affordable – be cognizant of maintenance
  - AKA evolution support
- Maintenance consumes 50-70% of budgetary resources in software lifecycle
- 50% of costs of maintenance associated with comprehension
  - Diagram support might reduce cost of comprehension!

Good Software: Modules

Good Systems Consist of Modules

- Fundamental problem: limit to how much a human can conceptualise.
  - Problem: increasing development team size exponentially increases overheads.
    Brooks, F. The Mythical Man-Month. Addison-Wesley. 1975.
- Enterprise development v. *heroic programming*.
- David Parnas advocated modular decomposition as far back as the early 1970s.
  - Conceptualise the system as a set of disjoint collaborating blocks
  - Blocks can be hierarchically decomposed.
- A system is a collection of modules: files, subroutines, library functions, classes in an object oriented language, other constructs known as module or similar, independent or semi-independent programs or subsystems.
  - Modular decomposition: cluster software "units" together.
  - What characteristics can be used to guide the clustering process?
  - Metrics: can these characteristics be measured?

Good Software: Dependency

Good modules: must consider dependency, coupling, cohesion, interface, encapsulation and abstraction.

Good Systems Have Minimal Dependency/Coupling

- Module A depends on Module B if it possible for some change to Module B to require modifications to Module A. Module A is a client of Module B, or that Module B acts as a server to Module A.

- Circular dependency: two modules are both clients and servers to each other. To be avoided.
- Good systems have low coupling, and thus, changes in one part of the system do not propagate throughout the rest of the system.
- Must identify which modules are coupled.
- Two types of facts that may be useful:
  - Which modules are clients of a given module?
  - What assumptions may clients of a given module make about it?

Good Software: Interfaces

Good Systems Have Interfaces

- An interface to a module defines some features of the module on which its clients may rely.
- "The connection between modules are the assumptions which the modules make about each other"

Software Fundamentals. Collected Papers by David L. Parnas.
Ed. By Daniel. M. Hoffman and David M. Weiss. Addison-Wesley.
2001.

- Document such assumptions in interfaces and check their correctness.
- Should be automatic checks to ensure that no client makes any assumptions about its server that is not documented in the interface.

Good Software: Context Dependencies

- The services a module requires are called its context dependencies. May be expressed in terms of an interface.
  - i.e Libraries, operating system, etc.

- The module's own interface(s) and its context dependencies constitute a contract describing the module's responsibilities.
- If a module changes internally without changing its interface, this change will not impact other modules that depend on it.

Good Software: Interfaces

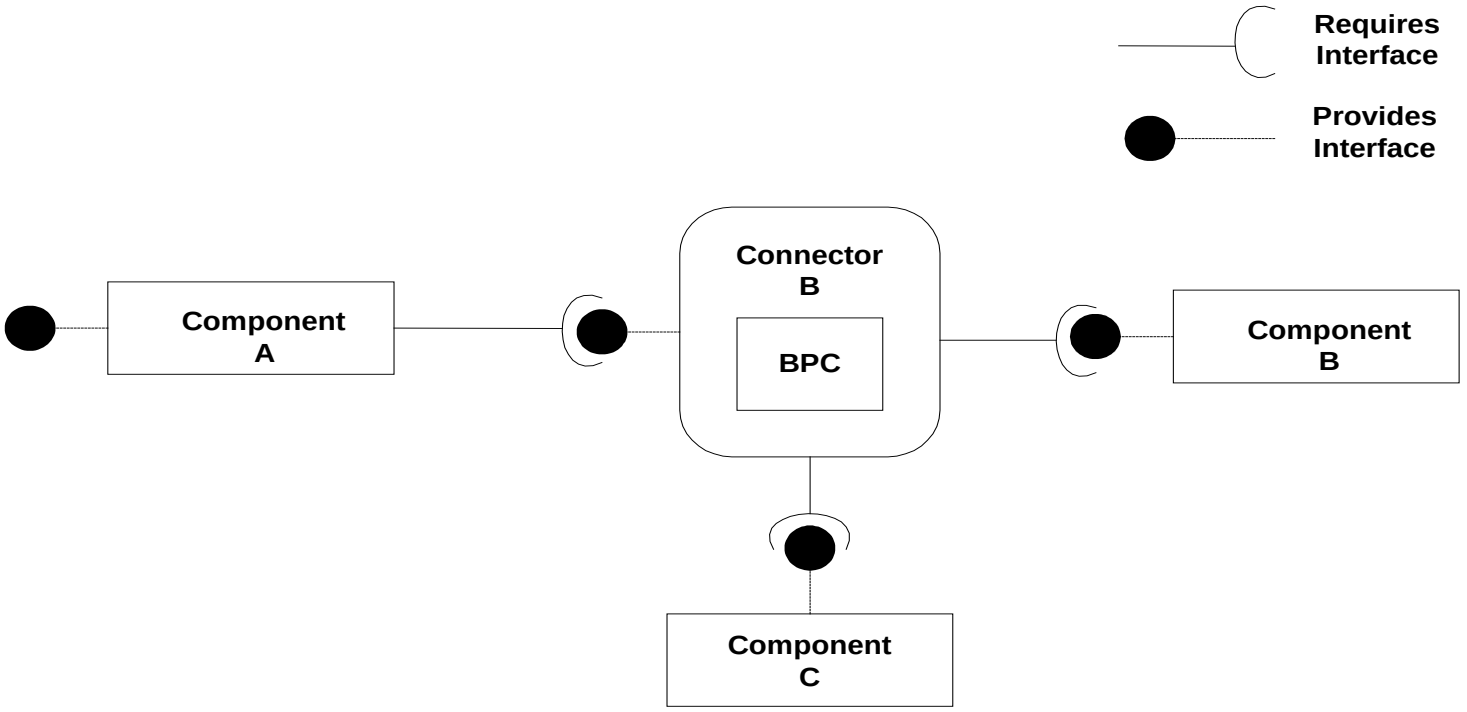


Figure 1. Requires and Provides interfaces in the ConnX framework, developed in a research programme in Dept. of CSIS, UL..

Interfaces

- Specification of a contract between a service provider (software) and a client (client developer who constructs software that uses the service provider).
- Contracts must be enforceable
  - Syntactic: signature, parameters, return type, etc., match.
  - Semantic: the meaning of the arguments supplied in the client call correlates with the intent of the parameters in the method heading.
    - Not currently doable.

Good Software

Benefits of Modularity with Defined Interfaces

- Easier to maintain.
- Easier to upgrade a modular system.
- Easier to build a system that is reliable because modules can be tested more thoroughly in isolation.
- Modules can be coded and tested concurrently, thus speeding up time-to-market.
- Team members only need to know other module interfaces, not how they works.
- Can reuse modules because of the property of documented interface(s).

Good Software

Cohesion, Abstraction, Encapsulation, Information Hiding.

- Good modules have the property that their interfaces provide an abstraction of some intuitively understood concept.
- Such modules are said to have high cohesion.
- Abstraction is when a client of a module does not need to know more than is documented in the interface.
- Encapsulation is when a client is not allowed to know more than is in the interface.
- Information hiding - encapsulation and abstraction

Good Software

- "A good system consists of encapsulated modules with high cohesion that are at the correct level of abstraction. The services provided by the system are clearly specified through interfaces."

  (Stevens and Pooley, 2000).

Good Software: CBD

Serves as a basis for Component Based Development (CBD)

- A module may have several interfaces
- If a module is a good abstraction - high cohesion and low coupling - may be able to reuse it or replace it.
- Pluggable component.
- Also depends on the architecture in which the component is developed.
- Architecture-centric CBD.

Definitions: Components

"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Syzperski, C. Component Software. Beyond Object-Oriented Programming. Addison-Wesley. 1999.

Software Architecture & UML

- UML diagrams taken from “CaseStudy” example provided for illustration in Component Factory distribution from Aonix.

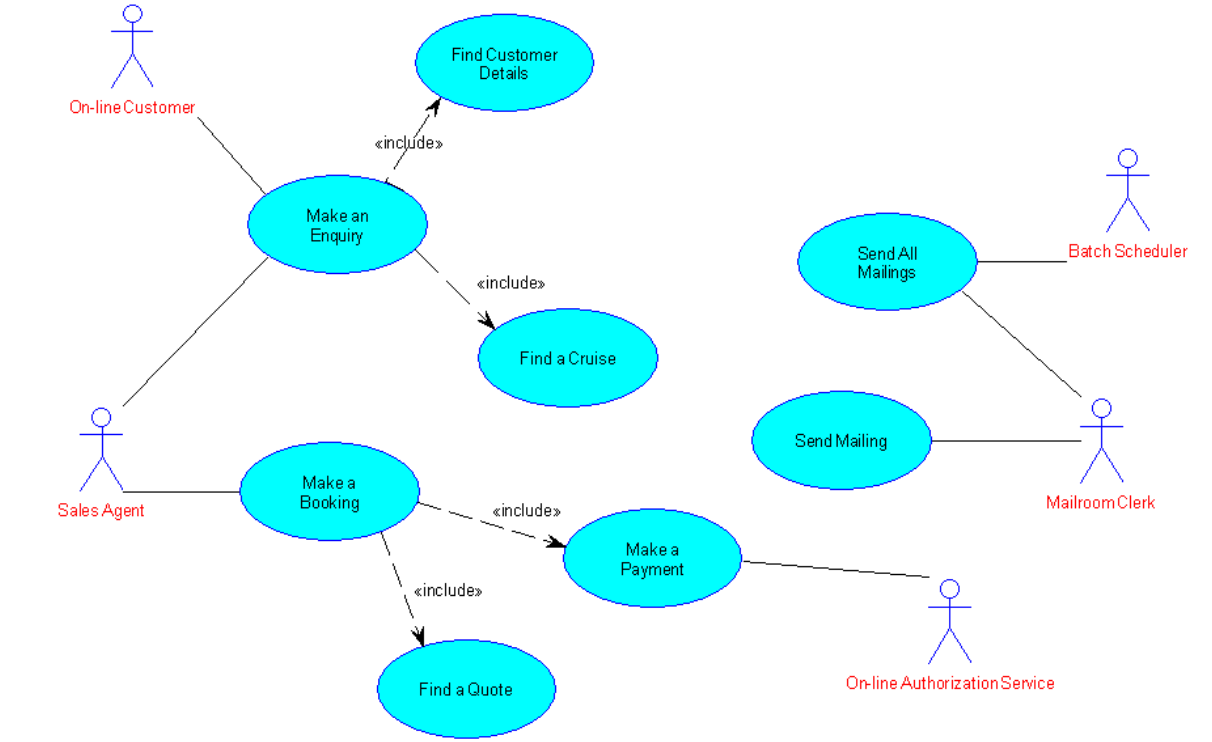


Figure 1. Use Case (Requirements) model from Select Solutions Factory "Case Study".

Software Architecture & UML

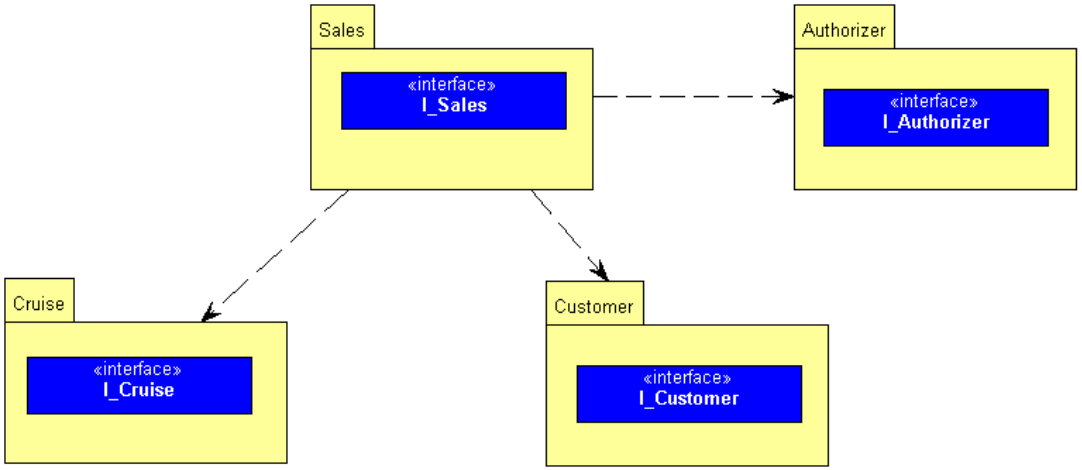


Figure 2. Business Architecture.

Software Architecture and UML

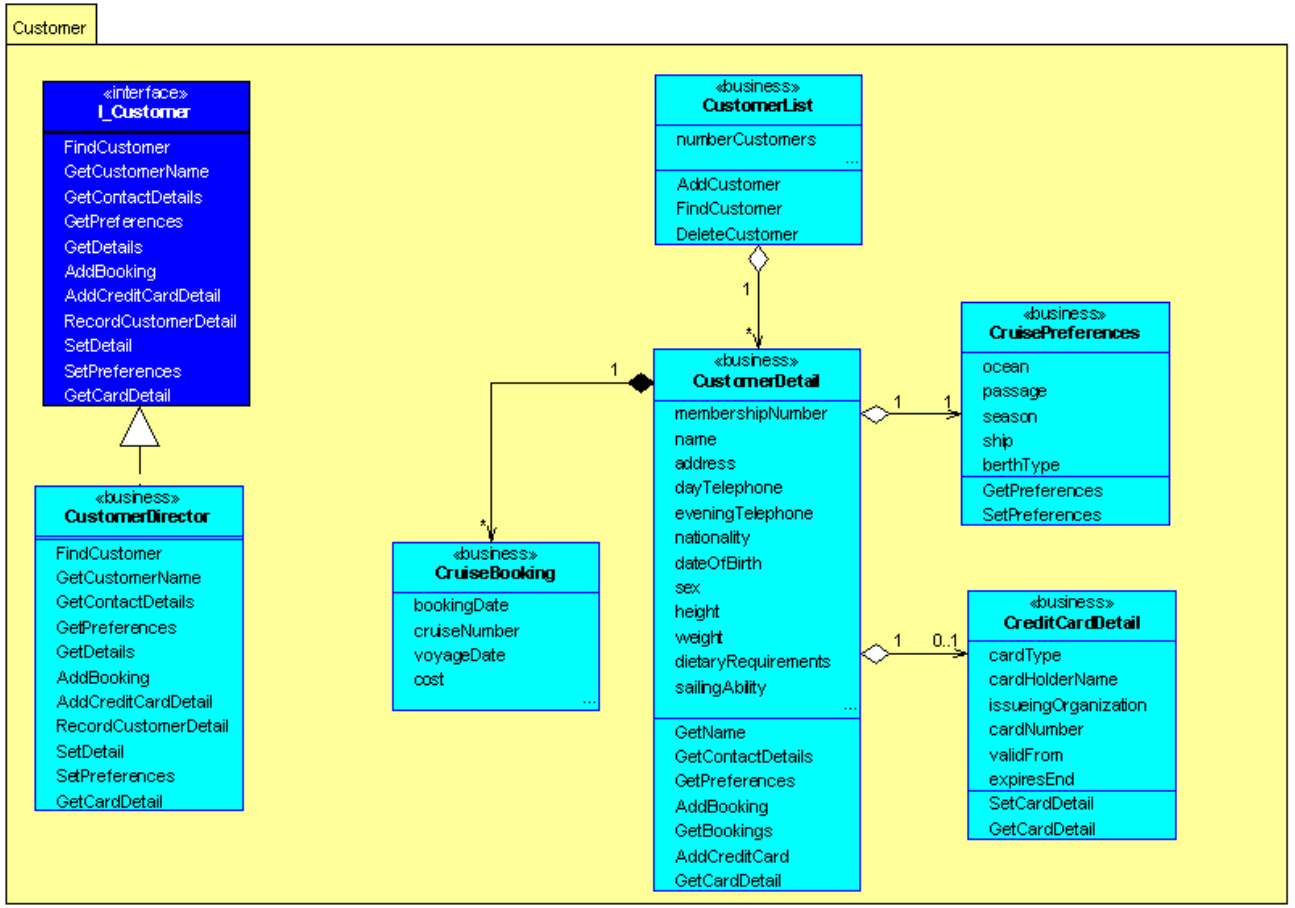


Figure 3. Customer component.

Software Architecture and UML

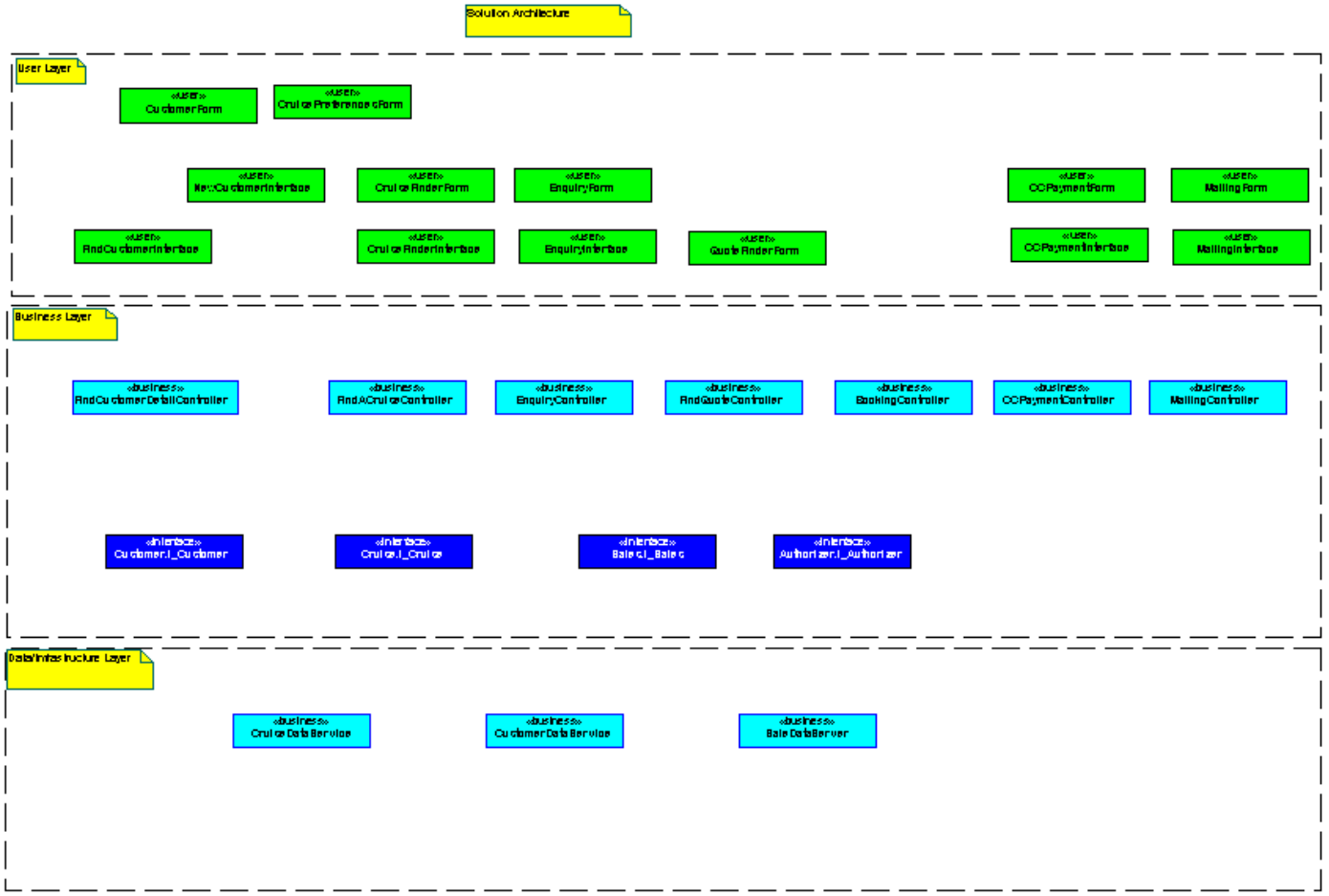


Figure 4. Technical solutions architecture from Select Solutions Factory "Case Study".

This module is focused on

- Learning and applying a generic OOAD method in order to create models of software during the requirements, analysis, and design phases of a software lifecycle
- Using the UML
- Based on architectural and design patterns where appropriate
- Leading to:
  - High quality software that is
    - REUSABLE
    - Easier/Cheaper to Maintain (dependency management).

Finally

- **Methodology:**
  - A methodology is a abstract description.
  - Systematic description of the sequence of activities required to solve the problem.
  - Provides techniques usually in the form of a formal graphical language to model a system.
- **Model of system:**
  - Simplified representation of some aspect of the real world.
  - Used to analyse this aspect and facilitate communication and understanding to others.
- **Diagramming:**
  - Profession has associated tools.
  - Formal techniques.
  - Form of Language.
  - Forms a good basis for understanding and communication.
  - Interchange of ideas.
  - Clear aid to thinking.
  - Speed up and improve quality of work.
  - Visualisation: events, processes, objects.
  - Clear diagrams play an essential part in designing complex systems and developing programs.
  - Essential aid to maintenance.

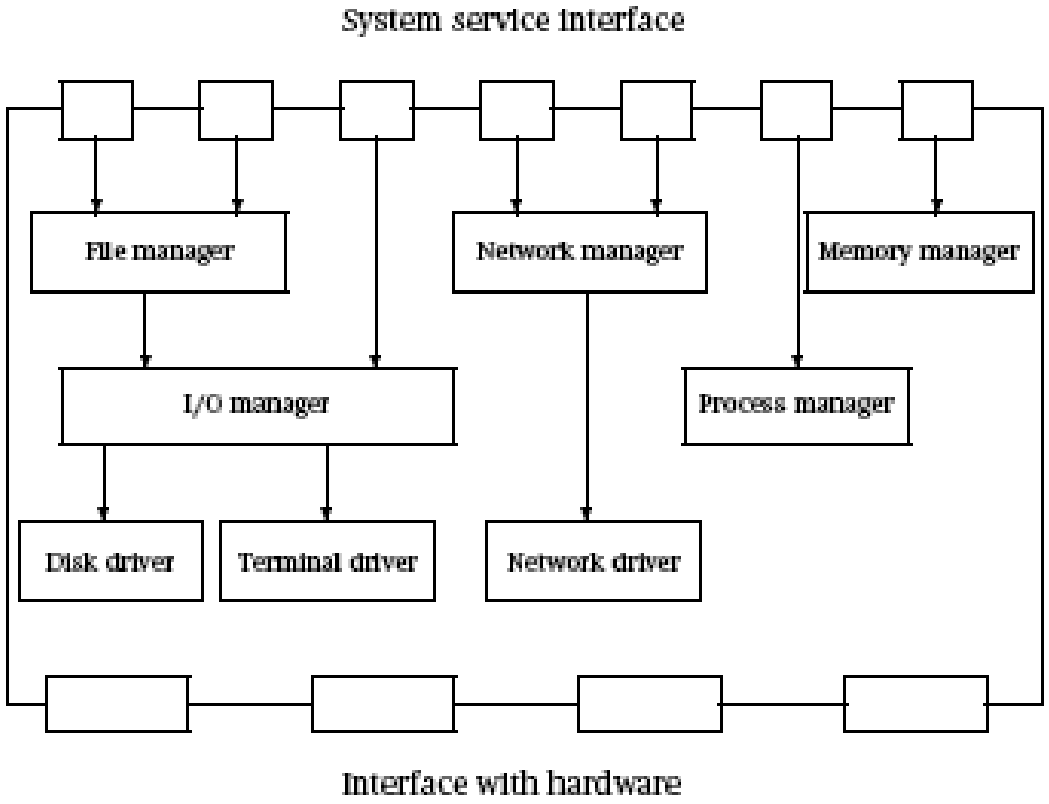
Review: Lectures 1 and 2

- Problems and solution (focus on architecture).
- Characteristics of good software
- Architecture Centric Development

- READING:
  - Bennett, McRobb, and Farmer: chapters 1 and 2.
  - Stevens and Pooley: chapter 1.

Question:

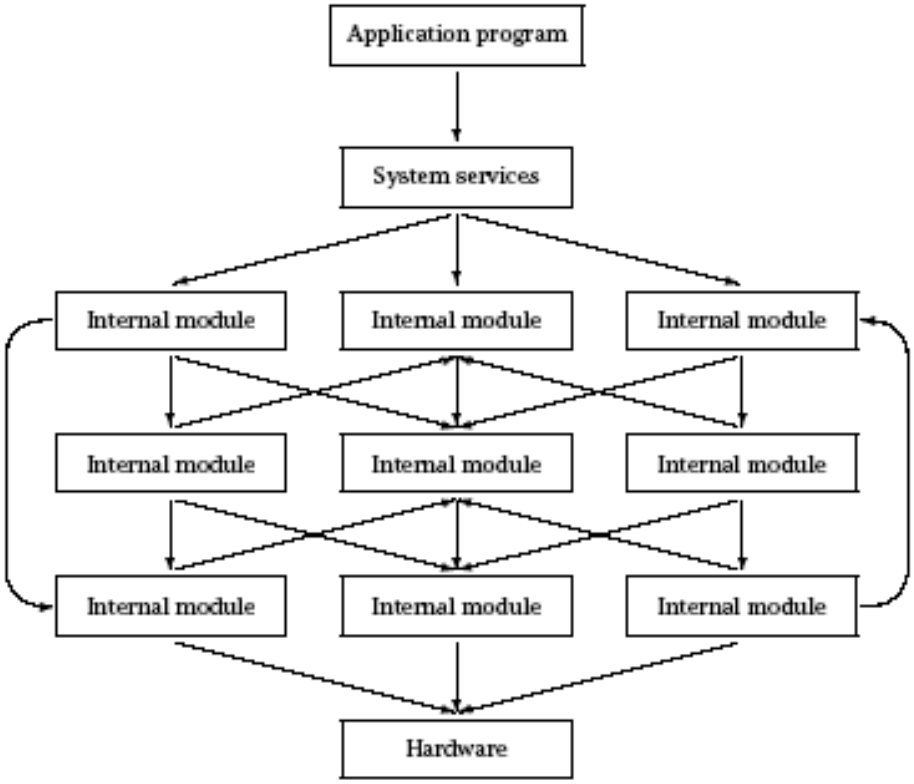
- Is this your design?



From O'Gorman, J. Operating Systems with Linux. Wiley. 2001

Question

- Or more likely, the following ??



From O'Gorman, J. Operating Systems with Linux. Wiley. 2001