**FUNCTIONAL REQUIREMENTS**

▸ Inputs▸ Outputs▸ Data (storage / other systems) File format used to temporarily backup

▸ Computations▸ Timing / Synchronization

**QUALITY REQUIREMENTS**

▸ Response time▸ Throughput▸ Resource usage▸ Reliability▸ Availability▸ Recovery from failure▸ Allowances for maintainability and enhancement▸ Allowances for reusability

**Platform:** Constraints on environment and tech of system **Process:** Constraints on project plan and development methods

**TEMPLATE OF A USE CASE**

A. **Name**: B. **Actors**: C. **Goals**:

D. **Preconditions**: E. **Summary**: F. **Related use cases**. G. **Steps**: Actor Response System Response

H. **Postconditions**: State of the system after completion.

**EXTENSIONS**

▸ Used to make optional interactions explicit or to handle

exceptional cases. ▸ Keep the description of the basic use case simple.

**GENERALIZATIONS**

▸ Much like super classes in a class diagram.

▸ A generalized use case represents several similar uses

cases. ▸ One or more specializations provide details of the similar

use cases.

**CHAPTER 6 – PATTERNS General Hierarchy Pattern**

Objects in a hierarchy can have one or more objects above them (superiors) and one or more objects below them (subordinates)

**The Player-Role Pattern**: A role is a particular set of properties associated with an object in a particular context

**The Singleton Pattern**: It is very common to find classes for which only one instance should exist (singleton)

When an association is created between two classes, the code for the classes becomes inseparable

**The Observer Pattern**: If you want to reuse one class, then you also have to reuse the other

**The Delegation Pattern:** You are designing a method in a class realize that another class has a method which provides the required service Inheritance is not appropriate

**The Adapter Pattern**: You are building an inheritance hierarchy and want to incorporate it into an existing class

**The Facade Pattern:** Often, an application contains several complex packages

**Immutable Pattern**: An immutable object is an object that has a state that never changes after creation

**Read-Only Interface Pattern:** You sometimes want certain privileged classes to be able to modify attributes that are otherwise immutable

**FACTORY:** A reusable framework needs to create objects; however the class of the created objects depends on the application

**CHAPTER 8 - UML BEHAVIOUR MODELS DESIGN SPACE** ▸ The space of possible designs that could be achieved by choosing different sets of alternatives is often called the **design space**

**TOP-DOWN DESIGN** ▸ Design very high-level structure ▸ Work down to detailed decisions about low-level constructs ▸ Arriving at detailed decisions such as: ▸ Data formats ▸ Algorithms ▸ API Interfaces

**BOTTOM-UP DESIGN** ▸ Decisions about re-usable low-level utilities ▸ Decisions on piecing them together ▸ Creating higher-level constructs

**MIXING TOP-DOWN AND BOTTOM-UP** ▸ A mixture of both normally used ▸ Top-Down gives a good structure ▸ Bottom-Up promotes re-usable components

**FUNCTIONAL COHESION** This is achieved when all the code that computes a particular result is kept together - and everything else is kept out ▸ i.e. when a module only performs a single computation, and returns a result, without having side-effects. ▸ Benefits to the system: ▸ Easier to understand ▸ More reusable ▸ Easier to replace ▸ Modules that update a database, create a new file or interact with the user are not functionally cohesive

**LAYER COHESION** All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out ▸ The layers should form a hierarchy ▸ Higher layers can access services of lower layers, ▸ Lower layers do not access higher layers ▸ The set of procedures through which a layer provides its services is the application programming interface (API) ▸ You can replace a layer without having any impact on the other layers ▸ You just replicate the API

**COMMUNICATIONAL COHESION** All the modules that access or manipulate certain data are kept together (e.g. in the same class) - and everything else is kept out ▸ A class would have good communicational cohesion ▸ if all the system’s facilities for storing and manipulating its data are contained in this class. ▸ if the class does not do anything other than manage its data. ▸ Main advantage: When you need to make changes to the data, you find all the code in one place

**SEQUENTIAL COHESION** Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out ▸ You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

**PROCEDURAL COHESION** Procedures that are used one after another are kept together ▸ Even if one does not necessarily provide input to the next. ▸ Weaker than sequential cohesion.

**TEMPORAL COHESION** Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out ▸ For example, placing together the code used during system start-up or initialization. ▸ Weaker than procedural cohesion.

**UTILITY COHESION** When related utilities which cannot be logically placed in other cohesive units are kept together ▸ A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable. ▸ For example, the **java.lang.Math** class

**CONTENT COUPLING** Occurs when one component surreptitiously modifies data that is internal to another component ▸ To reduce content coupling you should therefore encapsulate all instance variables ▸ declare them private ▸ and provide get and set methods ▸ A worse form of content coupling occurs when you directly modify an instance variable of an instance variable

**COMMON COUPLING** Using a global variable ‣ All components using the global are now coupled ‣ A weaker form is when a variable can be accessed by a subset of the system’s classes ‣ e.g. a Java package ‣ Acceptable for global variables of system-wide default values ‣ The Singleton pattern encapsulates global access to an object, but it is still a global.

**CONTROL COUPLING** Calling a procedure with a ‘**flag**’ or ‘**command**’ to control its behaviour ▸ To make a change you have to change both the ▸ Calling method (that send the command) and ▸ Called method (that interprets the command) ▸ Polymorphic operations a good way to avoid control coupling ▸ One way to reduce control coupling ▸ Using a look-up table ▸ Commands are mapped to a the callable method

**STAMP COUPLING** An application class is declared as the type of a method argument ▸ Since one class now uses the other, ▸ changing either becomes harder ▸ Reusing one class requires reusing the other ▸ Two ways to reduce stamp coupling, ▸ Use an interface as the argument type ▸ Passing simple variables (beware of content coupling)

**DATA COUPLING** The types of method arguments are primitive or simple library classes ▸ The more arguments, the higher the coupling ▸ Using that method means passing all those arguments ▸ Avoid it by limiting unnecessary arguments ▸ There is a trade-off (increasing one often decreases the other) between ▸ Data Coupling and ▸ Stamp coupling

**ROUTINE CALL COUPLING** One routine (or method) calls another ▸ The routines are coupled as they depend on each other’s behavior ▸ Routine call coupling is always present in any system. ▸ But should still be considered as coupling ▸ Is sequences of methods are use repeatedly then reduce call coupling by encapsulating into a single method

**TYPE USE COUPLING** A module uses a data type defined in another module ▸ Any time a class declares a variable of another class’ type ▸ If the type definition changes, then the declaring classes might be affected ▸ Always declare the type of a variable to be the most general possible class or interface that contains the required operations

**INCLUSION OR IMPORT COUPLING** One component imports a package (as in Java) or includes another (as in C++). ▸ The including/importing component is now ▸ exposed to everything in the included or imported component. ▸ If the included/imported component changes ▸ The including/importing component might have to change ▸ Can cause conflict forcing a change (e.g. duplicate names)

**EXTERNAL COUPLING** A module has a dependency on external things like the operating system, shared libraries or hardware ▸ Reduce the number of places where such dependencies exist (high cohesion). ▸ The Façade design pattern helps

**Chapter 10 Black-box vs white box testing**: white box is more time consuming than black box but removes much of the guesswork and allows the tester to be more thorough.

**BLACK-BOX TESTERS HAVE ACCESS TO … THE SYSTEM** ▸ Provide the system with inputs ▸ Observer outputs ▸ No access to ▸ Source code ▸ Internal data ▸ Documentation relating to the systems internals

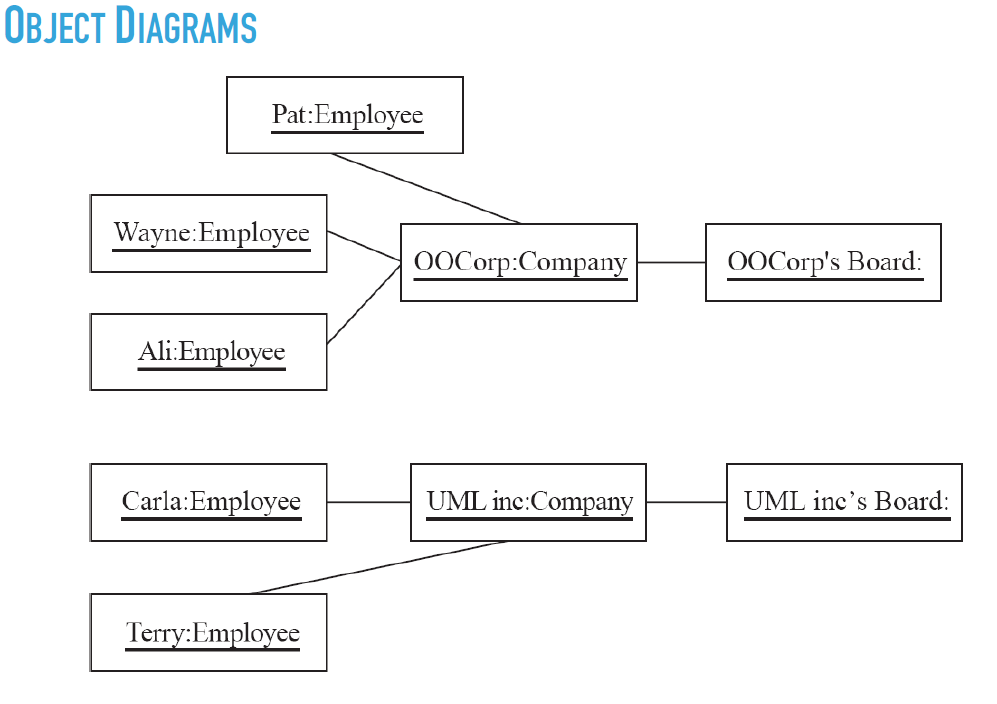
**QUIVALENCE CLASSES** ▸ Impossible to test by brute force using all input values ▸ e.g. every integer ▸ Instead, divide the possible inputs into groups ▸ Should be treated similarly ▸ Known as **equivalence classes** ▸ A tester needs only to run 1-3 tests per equivalence class ▸ You must understand the required input ▸ Appreciate how software may have been designed

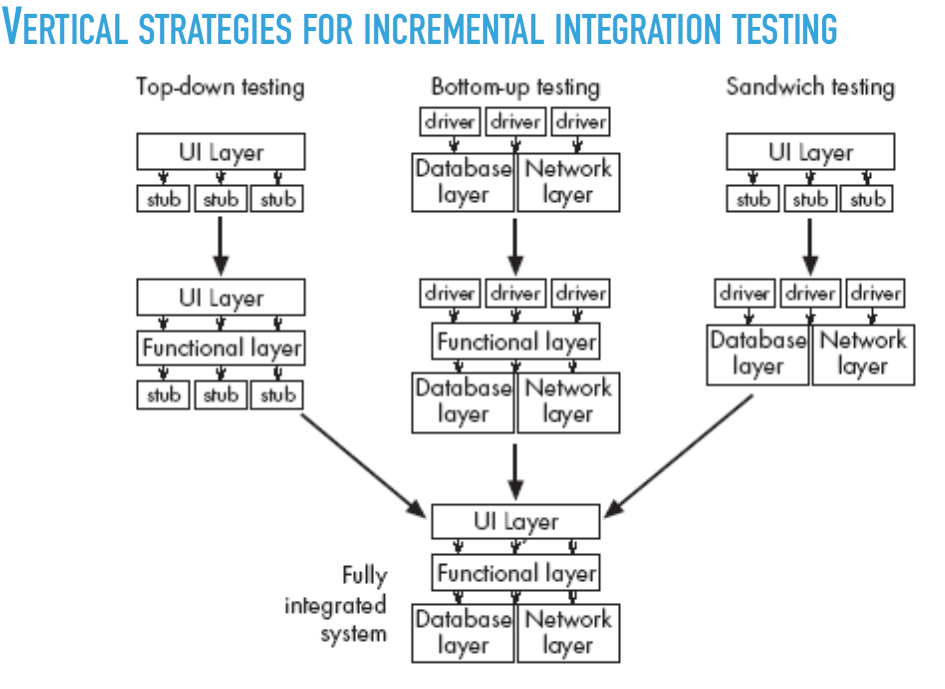
**BIG BANG TESTING VERSUS INTEGRATION TESTING**

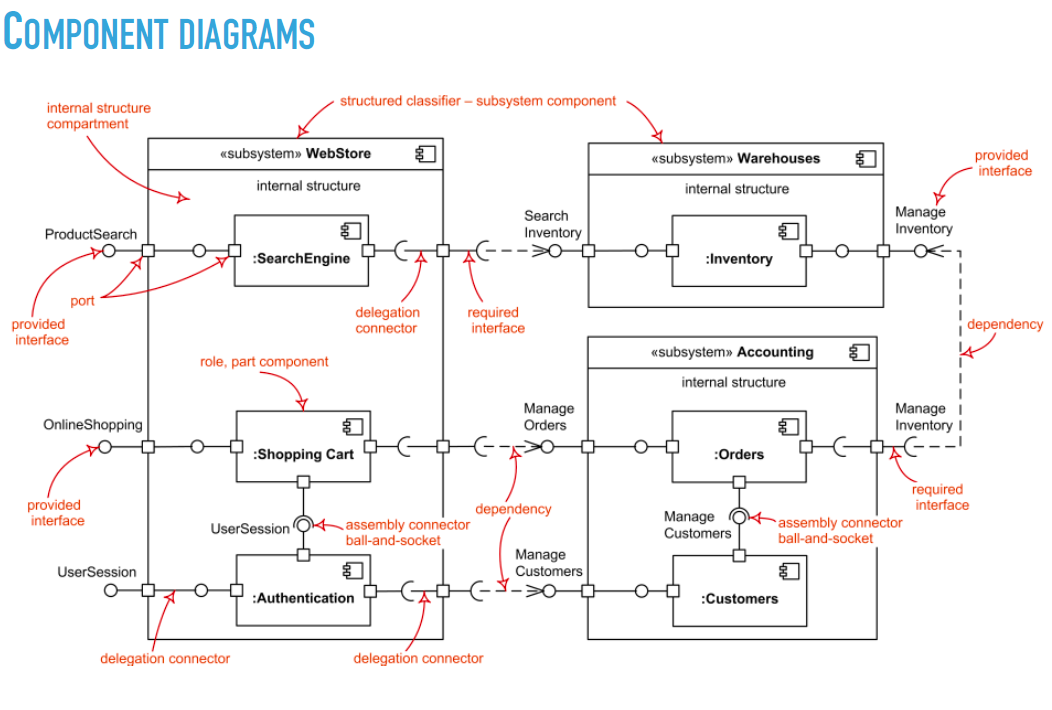
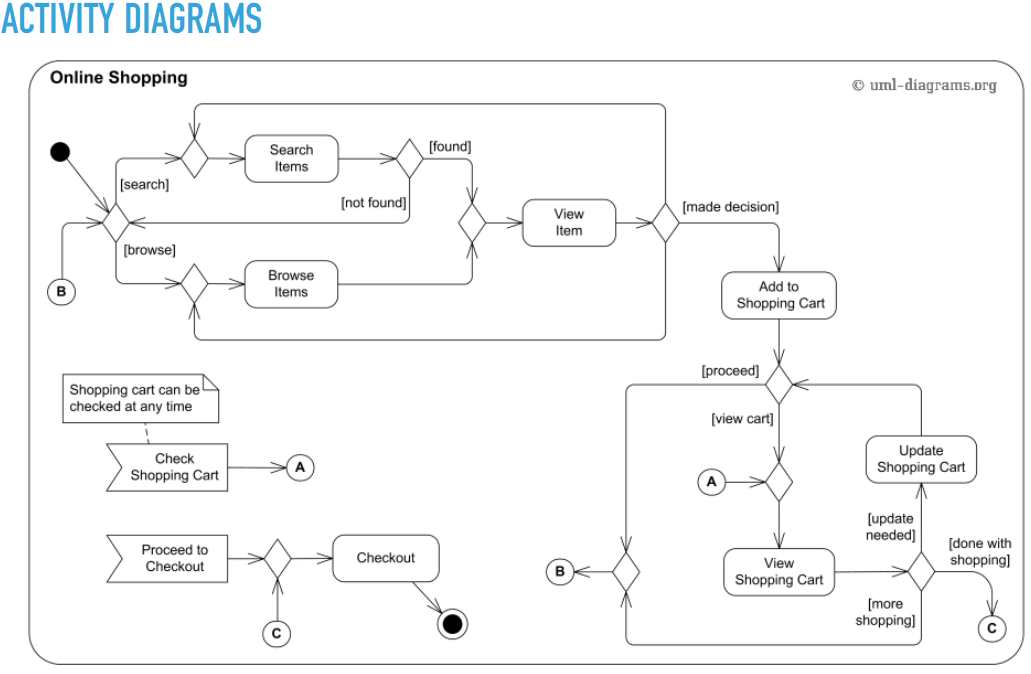
▸ In big bang testing, you take the entire system and test it as a unit A better strategy in most cases is incremental testing: You test each individual subsystem in isolation Continue testing as you add more and more subsystems to the final product▸ Incremental testing can be performed horizontally or vertically, depending on the architecture▸ Horizontal testing can be used when the system is divided into separate sub-applications

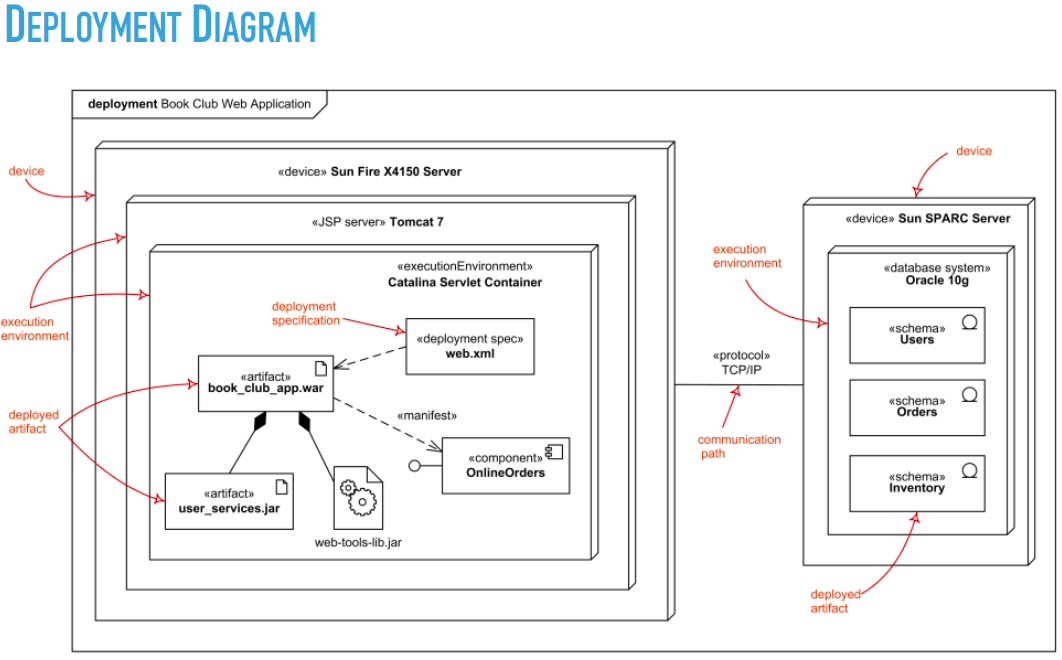
**REGRESSION TESTING**

▸ When testing manually ▸ Test a well-chosen subset of the previously successful test cases ▸ When testing automatically ▸ Run all tests, since it is generally fast









**BENEFITS OF DOMAIN ANALYSIS**

▸ Faster development▸ Better system

▸ Anticipation of extensions