Integration Testing
Integration Testing

- Software systems are built with components that must interoperate
- Primary purpose: To reveal component interoperability faults so that testing at system scope may proceed with the fewest possible interruptions
- Contrast:
  - Component scope testing – intracomponent faults
  - System scope testing – system scope responsibility
- An integration strategy must answer three questions
  - Which components are the focus of integration test?
  - In what sequence will component interfaces be exercised?
  - Which test design technique should be used to exercise each interface?
Integration Testing

- Incremental integration is the most effective technique
  - Add components a few at a time and then test their interoperability
  - Advantages:
    - Interfaces are systematically exercised
    - Observed failures are most likely to come from the most recently added component

- Strategy
  - A sequence of components should be identified using careful analysis of component dependencies
  - Testing must be planned and managed to follow these dependencies
Integration in OO Systems

- Integration in object-oriented development takes place at all scopes
  - Within a class
  - Within a class hierarchy
  - Between a client and its servers
  - Within a cluster of related classes
  - Within a subsystem
  - Within an application system
Dependency Analysis

- Components typically depend on each other in many ways
- Class and cluster scope dependencies result from explicit binding mechanisms
  - Composition and aggregation; Inheritance
  - Global variables; Server objects
  - Objects used as message parameters
  - ...
- Similar intercomponent dependencies occur at subsystem and system scopes.
- Dependencies often dictate the sequence of testing
Dependency Analysis
Dependency Tree for Example

FinancialService

Transaction
  Rates

Account
  Money
  AcctNum

Key: → Uses

How to handle dependency cycles?
Dependency Levels in Example

Level 2: Leaf Uses

<table>
<thead>
<tr>
<th>FinancialService</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction</td>
</tr>
<tr>
<td>Account</td>
</tr>
<tr>
<td>Rates</td>
</tr>
<tr>
<td>Money</td>
</tr>
<tr>
<td>AcctNum</td>
</tr>
</tbody>
</table>

Level 1: Single Uses

<table>
<thead>
<tr>
<th>FinancialService</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction</td>
</tr>
<tr>
<td>Account</td>
</tr>
<tr>
<td>Rates</td>
</tr>
<tr>
<td>Money</td>
</tr>
<tr>
<td>AcctNum</td>
</tr>
</tbody>
</table>

Level 0: Cluster Head

<table>
<thead>
<tr>
<th>FinancialService</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction</td>
</tr>
<tr>
<td>Account</td>
</tr>
<tr>
<td>Rates</td>
</tr>
<tr>
<td>Money</td>
</tr>
<tr>
<td>AcctNum</td>
</tr>
</tbody>
</table>
Integration Faults

- Integration testing can reveal component faults that cause failures when components interact.
- Typical interface bugs include the following:
  - Configuration/version control problems
  - Missing, overlapping, or conflicting functions
  - The client sends a message that violates the server's precondition or sequential constraints
  - Wrong object bound to message (polymorphic target)
  - Wrong parameters, or incorrect parameter values
  - Incorrect usage of virtual machine, ORB, or OS services
  - ...
Integration Patterns - 1

- Big Bang Integration: Attempt to demonstrate system stability by testing all components at the same time
- Bottom-up Integration: Interleave component and integration testing by following usage dependencies
- Top-down Integration: Interleave component and integration testing by following application control hierarchy
- Collaboration Integration: Choose an order of integration according to collaborations and their dependencies
- Layer Integration: Incrementally exercises interfaces and components in a layered architecture
Integration Patterns - 2

- Backbone Integration: Combine Top-down Integration, Bottom-up Integration and Big Bang Integration to reach a stable system that will support iterative development.
- Client/Server Integration: Exercise a loosely coupled network of components, all of which use a common server component.
- Distributed Services Integration: Exercise a loosely coupled network of peer-to-peer components.
- High-frequency Integration: Develop and rerun an integration test suite test hourly, daily or weekly.
Scope-specific Considerations

- Classes
  - The system is the class under test
  - The components to be integrated are CUT methods, superclass methods, instance variables, and parameters in messages received and sent by the CUT
  - Testing of class responsibilities is so closely coupled to class integration that it does not make sense to treat the two as separate test design patterns
Scope-specific Considerations

- **Classes**
  - Bottom-up intraclass integration is accomplished by an Alpha-Omega
  - Big Bang intraclass integration is accomplished by a Small Pop
  - Collaboration integration of a modal class may be accomplished with a Modal Class Test
  - Integration of superclass features can be accomplished with Polymorphic Server Test and Modal Hierarchy Test
  - There is no direct analogue to the other patterns
Cluster-specific Considerations

- The components to be integrated are server objects used by the CUT
  - Bottom-up Integration is the most widely used technique
  - Top-down Integration is possible if one class is the head of the cluster
  - Collaboration Integration of a modal class may be accomplished with Class Association Test, Round-trip Scenario Test, or Mode Machine Test
  - If the class under test catches exceptions, Controlled Exception Test may be useful
  - Big Bang integration is not recommended
  - There is no direct analogue to the other patterns
System-specific Considerations

- Subsystem/system
  - The scope of an implementation under test is often larger than of a single class or small cluster
  - At subsystem and system scopes, the goal of integration testing is achieving a sufficient stable system so that responsibility testing at system scope may proceed smoothly
System-specific Considerations - 2

- Bottom-up Integration works well for small to medium systems
- Top-down, Collaboration, and High-frequency Integration work for almost any scope or architecture
- Layer Integration applies to layered architectures
- Client/Server Integration is appropriate for client/server architectures
- Distributed Services Integration is appropriate for decentralized networks containing peer-to-peer nodes
- Backbone Integration is well suited to small to medium systems and especially useful for embedded systems
- Big Bang Integration is useful in a few limited circumstances
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
Integration Patterns

- Dependency tree that will be used to show successive configurations
Big Bang Integration

- **Intent**
  - Demonstrate stability by attempting to exercise an entire system with a few test runs

- **Context**
  - The SUT is stabilized and only a few components have been added or changed since the last time it passed a system scope test suite
  - The SUT is small and testable, and each of its components has passed adequate component scope tests
  - When components are so tightly coupled that they can not be exercised separately
Fault Model and Strategy

- **Fault Model**
  - The fault model is ambiguous and opportunistic
  - The hope is that the system will “run” and thereby demonstrate that system testing can begin

- **Strategy**
  - Dispenses incremental integration testing
  - The entire system is built and a test suite is applied to demonstrate minimal operability at system scope
  - Test suite may be developed at system scope by using an appropriate responsibility-based test design pattern
Big Bang Integration
Entry and Exit Criteria

- Entry Criteria
  - All components have passed component scope testing
  - The virtual machine to be used in the test environment is stable
  - A physical, functional, and environmental audit has been conducted and has not found any anomalies that would interfere with integration testing

- Exit Criteria
  - The test suite passes
Consequences

- Consequences
  - Debugging can be difficult because you receive fewer clues about fault locations
  - Even if SUT passes, many interfaces faults can hide and waylay subsequent system scope testing
  - Under favourable circumstances it can result in quick completion of integration testing
  - Few (if any) integration drivers or stubs are developed

- Known Uses
  - It is used in Backbone Integration
Bottom-up Integration

- **Intent**
  - Demonstrates stability by adding components to the SUT in uses-dependency order, beginning with components having the fewest dependences

- **Context**
  - Components with the least number of dependencies are tested first
  - When these components pass, their drivers are replaced with their clients; another round of testing then begins
  - It is often used to support unit scope testing in the iterative and incremental development of a subsystem’s components
Strategy - Test Model

- Develop a dependency tree
- The responsibility test suite for each component may be developed with any appropriate test design pattern
- The scope of the test suite for each driver is limited to the component under test
  - The driver does not attempt to exercise intercomponent interfaces
Strategy - Test Procedure

1. In the first stage, leaf-level components are coded.
2. Components on the next higher level are coded.
3. The entire system is exercised using the root-level component.

- Integration test cases: apply the appropriate responsibility-based test design pattern
Bottom-up Integration
Bottom-up Integration
Bottom-up Integration

FIGURE 13.9 Bottom-up Integration, third-stage configuration.
Bottom-up Integration
Bottom-up Integration
Bottom-up Integration

- Automation
  - It is necessary one driver for each component or component that is the root of a subtree in the dependency relationship
  - Avoid combining the test suites for components that can be tested by individual drivers into a single driver
  - The drivers may be revised as the class under test is revised
Entry and Exit Criteria

- **Entry Criteria**
  - The virtual machine to be used in the test environment is stable
  - The components to be integrated in each stage are minimally operable
  - A physical, functional, and environmental audit has been conducted and has not found any anomalies that would interfere with integration testing

- **Exit Criteria**
  - Each driver component meets the exit criteria for its test pattern
  - The interface to each component has been exercised at least once
  - Integration testing is complete when all root-level components pass their test suites
Consequences

● Disadvantages
  ● Driver development is the most significant cost
  ● The driver does not directly exercise intercomponent interfaces
  ● Postpones checking critical control interfaces and collaborations until the end of the development cycle
  ● If a fix, revision, or enhancement is made to previously tested component, the test configuration in which this component was first tested should be revised accordingly and rerun

● Advantages
  ● May begin as soon as any leaf-level component is ready
  ● Work may proceed in parallel
  ● Although this pattern reduces stubbing, stubs may still be needed to break a cycle or simulate exceptions
  ● It is suitable to responsibility-based design
Top-down Integration

- **Intent**
  - Demonstrate stability by adding components to the SUT in control hierarchy order, beginning with the top-level control objects

- **Context**
  - Control objects typically implement essential and nontrivial control strategies and therefore present relatively high risk
  - Top-down integration focuses on control components first, making the demonstration of system scope end-to-end operability a high priority
Strategy: Test Model

- The apex of control may be represented in a Collaboration Diagram, Sequence Diagram or Statechart of the SUT
- A responsibility test suite may be developed with any appropriate test design pattern
  - Modal Class Test or Mode Machine Test
  - Collaboration Integration, Round-trip Scenario Test or Covered in CRUD
Strategy: Test Procedure

- Model the control hierarchy as a dependency tree
- Develop a staged plan for implementation and testing
- Design a responsibility-based test suite at the system scope
  1. Develop and test the component(s) at the highest level of control first.
  2. Continue in breadth-first swath at each level, replacing the server stubs with a full implementation
  3. Continue in this manner until all servers in the SUT have been implemented and exercised
Top-down Integration

![Diagram of Top-down Integration](image-url)
Top-down Integration
Top-down Integration
Top-down Integration
Top-down Integration
Automation

- Requires a single driver for the control apex
- A stub is needed for each component in the layer below the current focus of integration
Entry and Exit Criteria

- **Entry Criteria**
  - The virtual machine to be used in the test environment is stable
  - The components to be integrated in each stage are minimally operable
  - A physical, functional, and environmental audit has been conducted and has not found any anomalies that would interfere with integration testing

- **Exit Criteria**
  - Each driven component meets the exit criteria of its test pattern
  - The interface to each subcomponent has been exercised at least one
  - Integration testing is complete when a build that includes all leaf-level components passes the system scope test suite
Consequences

- **Disadvantages**
  - Setting up a test requires that a large number of stubs be coded to provide the desired response
  - An unforeseen requirement in a lower-level component may necessitate last-minute changes to many top-components, breaking part of the test suite
  - The stubs are necessarily implementation-specific and likely to be brittle
  - It may be difficult to exercise lower-level components sufficiently

- **Advantages**
  - Testing and integration may begin early
  - The cost of driver development is reduced
  - Initially, components may be developed in parallel
Layer Integration

- **Intent**
  - Uses an incremental approach to verify stability in a layered architecture

- **Context**
  - For use with a pure layer architecture system

- **Fault Model**
  - Same as with general integration faults
  - Inadequate performance may present a critical failure in systems such as device drivers
Strategy: Test Model

- Combines top-down and bottom-up integration
- Identify layers and suitable integration test patterns for each
- Top layers are tested top-down
  - Use Mode Machine, Round-trip Scenario or Collaboration Integration
  - Use stubs for layer below
- Middle layer is developed bottom-up
- Bottom layer’s primary components tested in isolation
Strategy: Test Procedure

FIGURE 13.25 Layer integration, architecture and component test configuration.
Strategy: Test Procedure

- Layer integration may be top-down or bottom-up

- Top-down:
  - Test each layer in isolation
  - Perform top-down integration of layers

- Bottom-up:
  - Test each layer in isolation
  - Perform bottom-up integration of layers

- The test cases may be developed at system scope
Strategy: Test Procedure

FIGURE 13.26  Layer Integration, top-down configurations.
Strategy: Automation

- Requires drivers and stubs for each layer
- Top layer driver should be designed to exercise the entire system
Entry and Exit Criteria

- Entry Criteria
  - The virtual machine used in the test environment is stable
    - If not use Backbone Integration
  - Components of each collaboration are minimally operable
  - Physical, functional and environmental audit has been conducted

- Exit Criteria
  - Each driven component meets it’s exit criteria
  - Each collaboration traversing two or more layers has been exercised at least once
    - Verify with code coverage
Consequences

- Advantages and disadvantages are the same as for **Top-down Integration** (for the top-down variant) or **Bottom-up Integration** (for the bottom-up variant)
Client/Server Integration

- **Intent**
  - Demonstrate stability of client/server interaction
  - Begin by testing clients and servers in isolation and then use controlled increases in scope until all interfaces have been exercised

- **Context**
  - Achieves stepwise verification of interfaces between clients loosely coupled to servers
  - Concurrent development and testing possible
  - No single locus of control exists, each component of the system has its own control strategy
  - Components that have a single locus of control may be integrated using bottom-up, top-down or collaboration integration
Strategy: Test Model

- Identify clients and servers
- Use Extended Use Case Test or Round-trip Scenario Test to model client/server interaction
- Exercise all combinations of clients and servers
Strategy: Test Procedure

1. Each client is tested with a stub for the server
2. The server is tested with stubs for all client types
3. Test actual pairs of clients and servers
   - Server may retain stubs for other clients
4. Finally, all stubs are removed and the individual use cases are replayed
Strategy: Test Procedure

- How to integrate every unique client in large systems?
  - Identify client groups
  - Consider group as representative client
  - Test as having only the representative clients
Strategy: Test Procedure

Stage 1: Star Server

Stage 1: Client Group 1

Stage 1: Client Group 2

Stage 2: Client Group 1 Use Cases

Stage 3: Client Group 2 Use Cases

Stage 4: All Use Cases
Strategy: Automation

- The interface of the component under test will determine the kind of driver used
- It is difficult to debug threads that span several platforms
- Integration should be complete only after testing on an isomorphic environment
  - Same structure and architecture, if not same scale
Entry Criteria

- Virtual machine in the test environment is stable
- Components are minimally operable
- Physical, functional and environmental audit has been conducted
- Multiplatform test tool suite installed and working
- Multiplatform version control system has been installed
Exit Criteria

- Each driven component meets exit criteria for its pattern
- Interface to each subcomponent has been exercised at least once
  - Distribution limits coverage tools usefulness
  - Some effort should still be made to establish coverage of each end-to-end path
- The test suite passes in an environment isomorphic to the target environment
Consequences

- Disadvantages:
  - Cost of driver and stub development for clients and servers
  - Cannot exercise end-to-end use cases until midway or late in the testing cycle

- Advantages:
  - Avoids big bang integration problems
  - Order can be sequenced according to priority or risk
    - Client and server integration order has few constraints
  - Approach supports controllable and repeatable testing
Collaboration Integration

- **Intent**
  - Demonstrate stability by adding sets of components to the SUT that are required to support a given collaboration.

- **Context**
  - Exercises interfaces between participants of a collaboration and organizes integration according to collaborations.
  - Integration is complete when all components and interfaces have been tested, which may not require testing every collaboration.
  - Use when:
    - The system has clearly defined collaborations that cover all components.
    - Demonstrating a working collaboration early is important.
    - Credible demonstration requires more than just a pass through the control hierarchy.
Backbone Integration

- Intent
  - Combines Top-down, Bottom-up and Big Bang Integration to verify interoperability among tightly coupled subsystems

- Context
  - Embedded system applications and their infrastructure are often developed at the same time
    - The application cannot operate without the backbone
    - The backbone provides services that are essential for running tests and the application
    - Stub the backbone would be impractical
    - The high-level control strategy can be exercised with stubs for the upper and middle levels
    - The middle level consists of several clusters that have loose intercluster coupling, but tight intracluster coupling
Distributed Services Integration

- **Intent**
  - Demonstrate stability of interaction among loosely coupled peer components
  - Begin by testing some nodes in isolation, then use controlled increases in scope until all interfaces have been exercised

- **Context**
  - Components run concurrently
  - No single locus of control
  - No single hierarchy of servers
High Frequency Integration

- **Intent**
  - Integrate new code with a stabilized baseline frequently to prevent integration bugs from going undiscovered and to prevent divergence from the working, stabilized baseline

- **Context**
  - Rapid incremental development can result in missing or conflicting capabilities
    - Frequent integration prevents the festering of this
  - Necessary conditions for beginning a High-frequency integration regime
    - A stable increment is available
    - Meaningful increments can be produced in the frequency interval
    - Test suite is developed in parallel with code and kept current
    - High Frequency Integration must be automated
    - Configuration management tool must be installed and is routinely used