Testing Object-Oriented Applications

Chapter 19: Lecture Title

Subject: Software Engineering

Program: BTech Computer Science and Engineering

Duration: 1 Hour

The Object-Oriented Shift

Testing Collaborating Objects in Complex Systems

Context: Introduction

- Transitioning from traditional, function-based testing to the unique challenges of the OO paradigm.

A New Perspective on Testing

Moving Beyond the Function

The Traditional Approach

Until now, we have mastered testing conventional software by focusing on individual functions, isolated modules, and specific execution paths.

The OO Difference

Object-Oriented software fundamentally changes the architecture. It is built from collaborating objects and introduces complex dynamics like:

- Inheritance
- Polymorphism
- Dynamic Binding

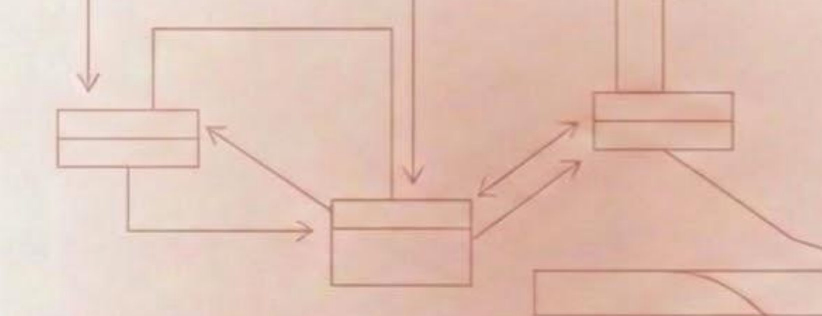
The Broadened View

To effectively test OO software, we must shift our focus from testing standalone functions to testing complete classes, clusters of classes, and the intricate interactions between objects.

Learning Objectives

What You Will Master Today

By the end of this lecture, you will be equipped to:



- **Broaden the View:** Expand your understanding of testing to fully encompass OO concepts and artifacts.
- **Evaluate Models:** Describe effective strategies for testing OO analysis and design models before code is even written.
- **Adapt Core Strategies:** Explain how unit, integration, and validation testing strategies must change within an OO context.
- **Apply OO Methods:** Utilize specific OO testing methods, including fault-based testing, scenario-based testing, and the testing of complex class hierarchies.
- **Design Class Tests:** Successfully design tests at both the class level (using random and partition testing) and the interclass level.

A Paradigm Shift

Adapting Testing for Object-Oriented Architecture

Part 1 – Why conventional testing falls short and why we must start earlier in the lifecycle.

Conventional vs. OO Testing

Changing the Target

Conventional testing and Object-Oriented testing approach the software from entirely different angles.

Conventional Testing

Feature	
Primary Focus	Processes, isolated functions, subroutines, and individual modules.
Starting Point	Typically begins at the code implementation level.
Complexity Drivers	Control flow and data flow.

Object-Oriented Testing

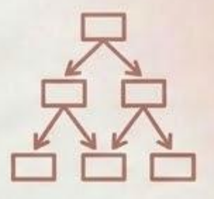
Feature	
Primary Focus	Collaborating objects, complete classes, and inter-object communication.
Starting Point	Must begin much earlier at the Analysis and Design (OOA/OOD) stages.
Complexity Drivers	Object state, inheritance trees, and runtime binding.

> Key Insight: Errors embedded in OO models seamlessly propagate into the code. Therefore, OO testing must begin at the design stage to prevent catastrophic, systemic bugs.

The Four Challenges of OO Testing

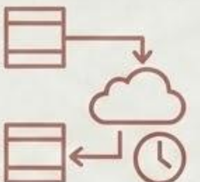
Testing the Unique Characteristics

Encapsulation:
Methods and state data are bundled together. We can no longer test a method in a vacuum; we must test the class as a whole.



Inheritance:
Subclasses inherit behaviors from their parents. A change in a parent class immediately affects all children, requiring broad regression testing.

Polymorphism:
The exact same method name may invoke completely different implementations. Tests must cover all possible variants.



Dynamic Binding:
Method binding occurs at runtime rather than compile time. The testing environment must ensure the correct binding executes under live conditions.

Testing OOA and OOD Models

Catching Expensive Errors Early

Before a single line of code is written, we must test the Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) models.

Errors in requirements, analysis, and design are structurally the most expensive to fix later.

We evaluate these models on two strict criteria:

Correctness:
Does the model accurately represent the reality of the problem domain?

Consistency:
Do the different views and diagrams within the model completely agree with each other?

Verifying Correctness

Does it Match Reality?

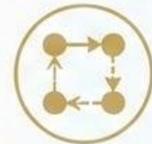
To validate the correctness of OOA and OOD models, we must ask strict, domain-focused questions:

Requirement Traceability

- Are absolutely all classes directly derived from the stated requirements?

Domain Accuracy

- Do the modeled attributes and operations correctly represent the actual business or problem domain?



Relationship Integrity

- Do the defined associations, inheritances, and aggregations correctly reflect real-world connections and constraints?

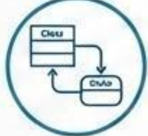
Ensuring Consistency

Cross-Checking the Blueprints

Consistency ensures that our architectural blueprints do not contradict one another. We must perform rigorous cross-checks:

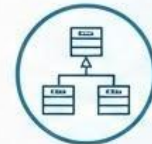
Use Cases ↔ Class Diagram

- Do all the scenarios outlined in the use cases map flawlessly to sequences of class collaborations?



Class Diagram ↔ State Diagrams

- Do the documented state transitions perfectly align with the behavior defined in the classes?



Inheritance Hierarchy

- Do the subclass attributes and methods strictly respect the Liskov Substitution Principle (meaning a subclass can stand in for a parent without breaking the system)?

FIGURE 19.1

An example CRC index card used for review

class name: credit sale

class type: transaction event

class characteristics: nontangible, atomic, sequential, permanent, guarded

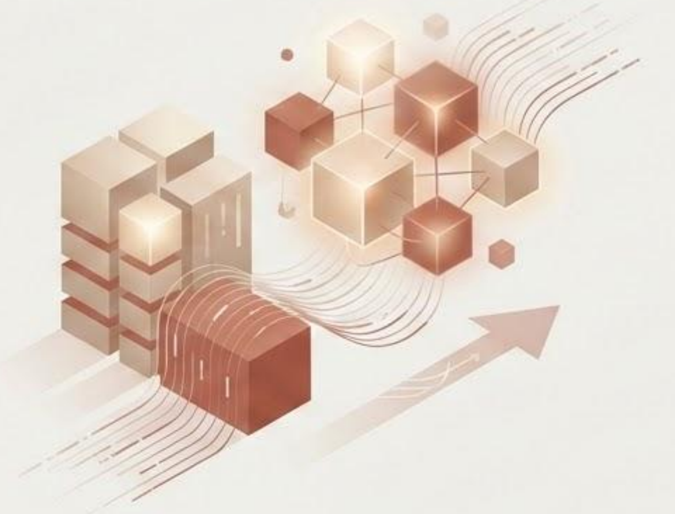
responsibilities:	collaborators:
read credit card	credit card
get authorization	credit authority
post purchase amount	product ticket
	sales ledger
	audit file
generate bill	bill

Part 2: Object-Oriented Testing Strategies

SHIFTING THE TESTING PARADIGM

Unit, Integration, and Validation in an OO World

Context: Part 2 - How we adapt standard testing levels to handle encapsulated, collaborating objects.



Unit Testing in the OO Context

The Class is the Unit

The Shift & The Reason

- In conventional software, a unit is a function. In OO software, the unit is the Class.
- Methods within a class share the same internal state. Testing a method in complete isolation ignores this critical shared context and can mask serious bugs.

The Class Testing Approach

- Design test cases that exercise the class as a complete, cohesive entity.
- Test the constructor, the destructor, and all methods in various sequential combinations.
- Verify that the class strictly maintains its correct internal state across multiple method calls.

Tooling

- Drivers: Still required to instantiate the objects, invoke methods, and check the resulting state.
- Stubs: Often needed to stand in for other classes that have not yet been implemented.

OO Integration Testing

Testing the Collaborations

The Shift

- Conventional integration focuses on the data interfaces between isolated modules.
- OO integration focuses on the dynamic collaborations and message message passing between active objects.

Approach 1: Thread-Based Testing

- The Concept: Integrate the specific set of classes that are required to respond to one specific input or event.
- Example: Testing all the individual classes involved in "processing a customer order" together at the same time.
- The Advantage: It provides the development team with early visibility into complete, end-to-end functionality.

Use-Based and Cluster Testing

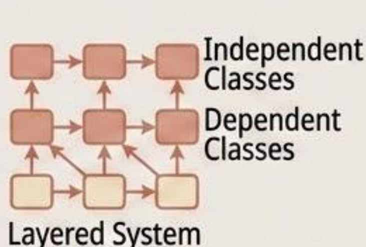
Layering the Integrations

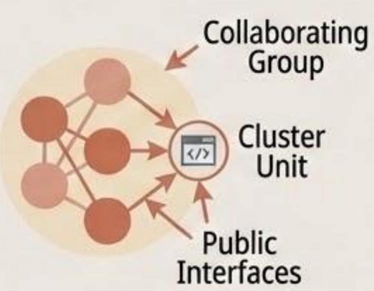
Approach 2: Use-Based Testing

- The Concept: Integrate classes in distinct layers based entirely on their usage dependencies.
- The Process: Start by testing the most independent classes (those that do not rely on many other classes). Once verified, integrate the dependent classes that utilize those independent ones.
- Note: This closely mirrors a traditional bottom-up integration approach.

Cluster Testing:

- A "cluster" is defined as a specific, collaborating group of classes.
- The strategy is to test the entire cluster as a single, larger unit by strictly exercising its public-facing interfaces.





Validation Testing in an OO Context

Fulfilling the User's Reality

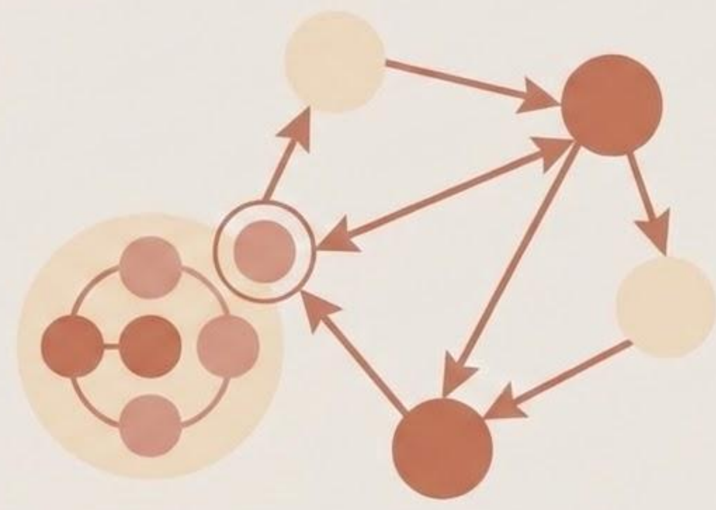
- The Focus: Validating that the final, integrated system actually meets the documented user requirements.
- The Core Method: This remains identical to conventional validation—it is strictly black-box testing from the user's perspective.
- The OO Twist: The Use Cases built during the design phase become your primary test specifications. Every single use case scenario must be tested end-to-end to ensure the hidden network of collaborating objects delivers the correct result to the user.

Part 3: Object-Oriented Testing Methods

Navigating OO Testing Methods

Designing Tests for Complex Object Behaviors

Part 3 - Exploring specific methodologies tailored for encapsulation, inheritance, and polymorphism.



Slide 2: Test-Case Design Implications

How OO Concepts Change the Rules

Encapsulation

Test designers must thoroughly understand the class's public interface, treating the internal implementation as a black box.

Inheritance

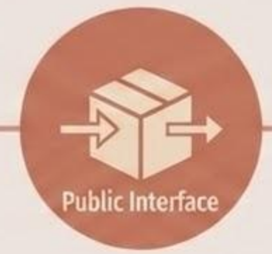
We cannot assume an inherited method works perfectly in a new context. Subclasses require retesting of inherited methods, especially if they are overridden or interact with newly added state variables.

Polymorphism

A single method call might trigger different code depending on the object type. Test cases must explicitly force the system to exercise all possible polymorphic implementations.

Slide 3: Conventional Methods in an OO World

What Still Works and What Doesn't



White-Box Methods

Still highly applicable at the micro-level. We use basis path, condition, and loop testing for the code inside individual methods.

Black-Box Methods

Still highly applicable for testing the public interfaces of classes and validating overarching use case scenarios.

The Limitation

Conventional methods fall short because they do not account for OO-specific architectures like inheritance trees, dynamic binding, and complex state behaviors.

Slide 4: Fault-Based Testing

Hunting the Usual Suspects

The Premise:

Instead of guessing where bugs might be, we identify the most likely OO-specific faults and design tests specifically to trigger them.

Common OO Faults to Target:

Improper object initialization (Constructor errors).

Incorrect method implementation or failure to handle boundaries.

Misuse of inheritance (e.g., a subclass that violates the behavior expected from its parent).

Polymorphism errors (e.g., incorrect dynamic binding at runtime).

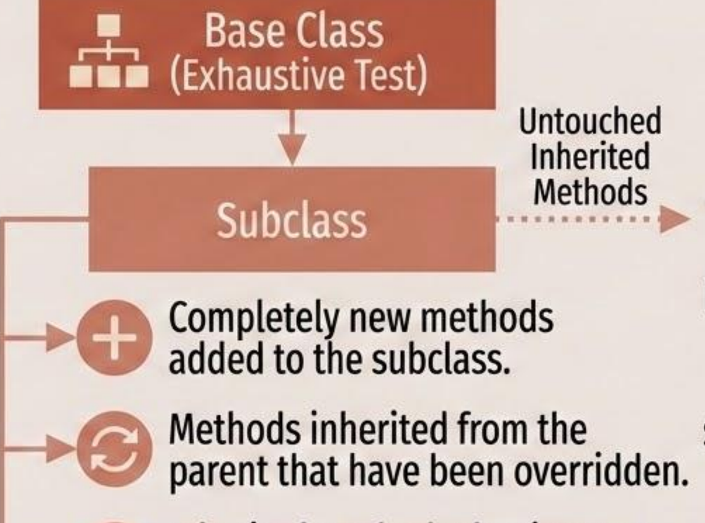
Slide 5: Testing the Class Hierarchy

Maximizing Coverage, Minimizing Redundancy

The Strategy

The Challenge

How do we test a deep inheritance hierarchy without wasting time retesting the exact same code at every level?



The Optimization

Inherited methods that are untouched and do not interact with any new subclass state generally do not require full retesting.

Slide 6: Scenario-Based Test Design

Testing the Interactions

The Focus:

Testing the software from the user's perspective by executing complete Use Case scenarios.

The "Why":

In OO systems, the most severe faults rarely occur within a single object; they occur in the complex interactions and messaging between collaborating objects.

The Process:

Select a core use case from the requirements.

↓

Map out the main success scenario and any alternate flows.

↓

Execute the scenario end-to-end, actively verifying that the objects collaborate correctly and state changes occur exactly as designed.

Slide 7: Surface Structure vs. Deep Structure

Two Levels of Verification

Surface Structure

What It Is: The external view—the public interfaces and observable behaviors of the classes.

How We Test It:

- Black-Box Methods: Equivalence partitioning and Boundary Value Analysis on public methods.

Deep Structure

What It Is: The internal view—the underlying object interactions, state mutations, and hidden collaborations.

How We Test It:

- Advanced Methods: Scenario-based testing, fault-based testing, and strict state-based testing.

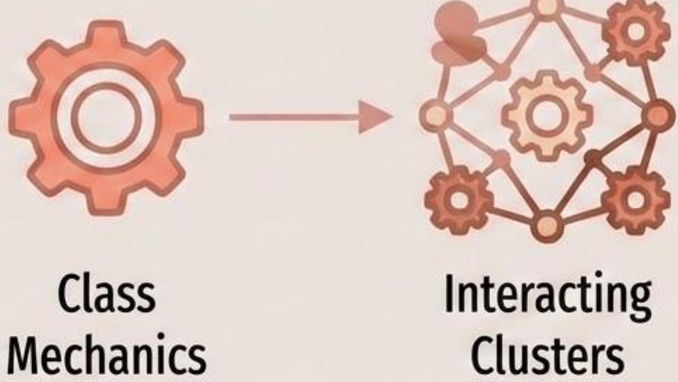
Part 4: Testing at Class and Interclass Levels

Title: Scaling Up the Scope

Subtitle: Validating Individual Objects and Their Collaborations

Context: Part 4

Moving from internal class mechanics to how clusters of classes interact in the wild.



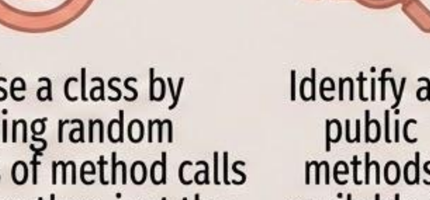
Random Testing for OO Classes

Expecting the Unexpected

The Goal

Exercise a class by throwing random sequences of method calls at it, rather than just the “happy path” expected by the developer.

The Approach



Identify all public methods available in the class.

Generate entirely random sequences of these method calls, supplying random parameters within valid ranges.

After every single call, verify that the object’s internal state remains valid.

For state-dependent classes, use state transition diagrams to guide the random generation.

The Advantage

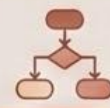
Highly effective at uncovering unexpected interaction errors between methods that a human tester might never think to execute sequentially.

Partition Testing at the Class Level

Dividing and Conquering the Input Space

The Goal

Divide the massive input space of a class into manageable, logical groups to reduce the total number of test cases while maintaining maximum coverage.



State-Based

How It Groups Test Cases: By the specific state the object is in immediately prior to the method invocation.

Examples:

- Before Initialization
- After Open
- After Close

Attribute-Based

How It Groups Test Cases: By the specific values held by internal attributes.

Examples:

- Account Balance > $0
- Account Balance < $0

Category-Based

How It Groups Test Cases: By the fundamental function or category of the method itself.

Examples:

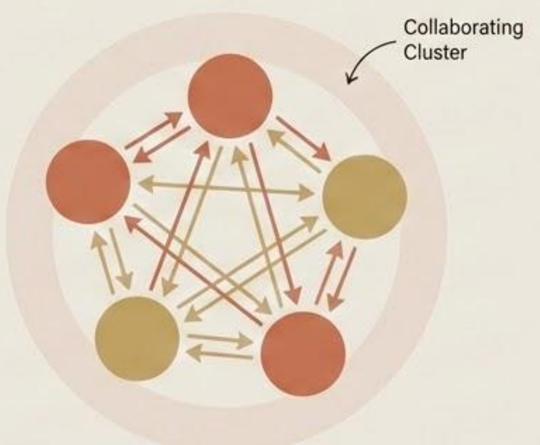
- Constructors
- Modifiers
- Accessors

Interclass Test-Case Design

Testing the Cluster

The Focus

Moving beyond the single class to test the dynamic collaborations between multiple classes.



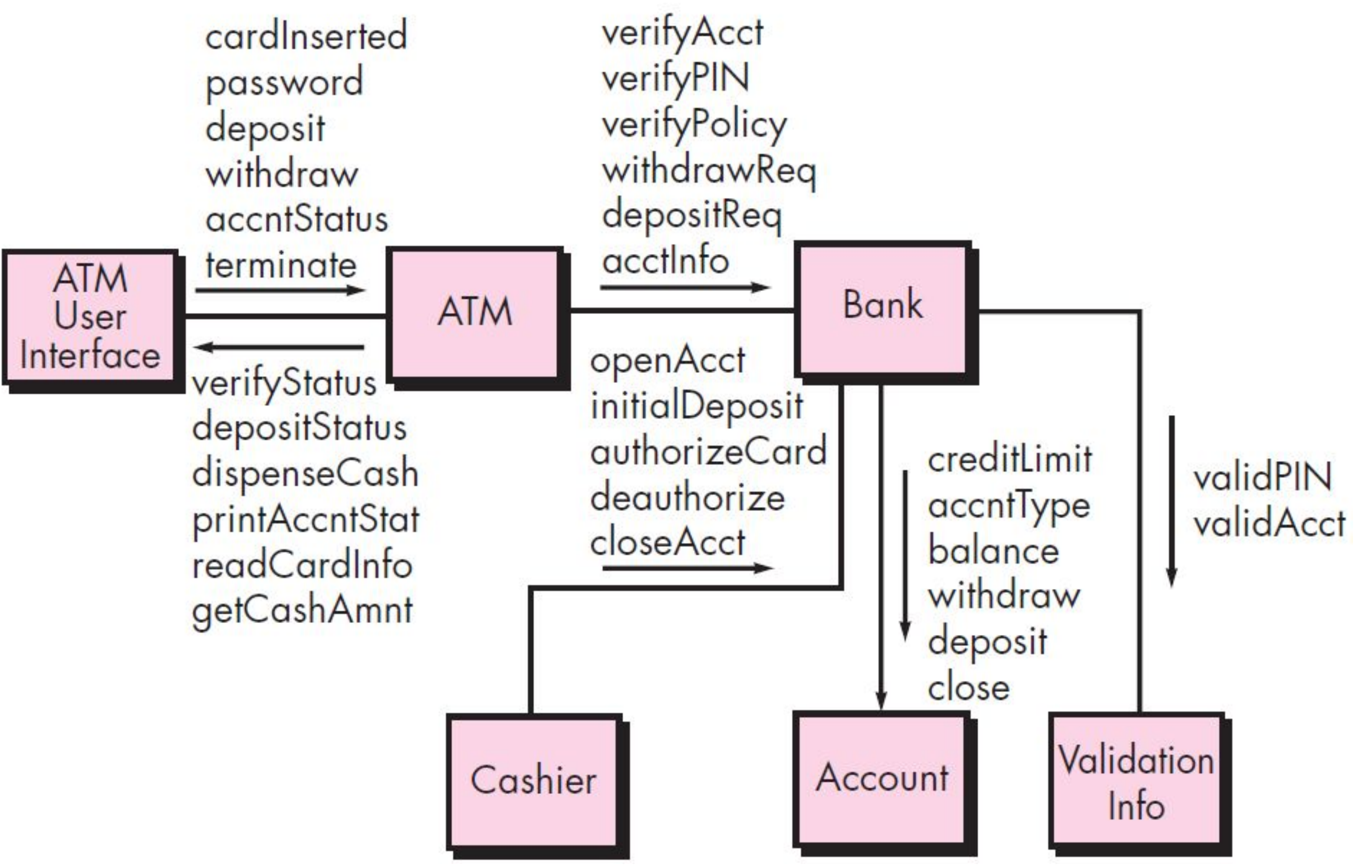
The Approach

1. Identify specific clusters of collaborating classes (easily pulled from sequence or collaboration diagrams).
2. Design test cases that strictly exercise the cluster's public-facing interfaces.
3. Utilize scenario-based testing to drive real-world interactions through the cluster.
4. Actively verify that all objects within the cluster maintain their correct states and structural relationships throughout the entire transaction.

FIGURE 19.2

Class collaboration diagram for banking application

Source: Adapted from [Kir94].

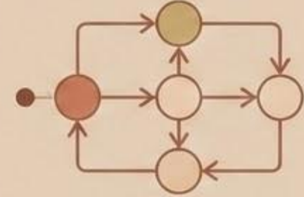


Testing Derived from Behavior Models

Using Blueprints as Test Specs

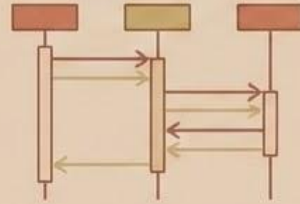
We can use our design models—specifically Statecharts and Sequence Diagrams—as direct specifications for our tests.

Statechart-Based Testing



- Map out the statechart for the system or class.
- Design test cases specifically built to cover every single state, every transition, every triggering event, and all primary paths through the chart.

Sequence Diagram Testing

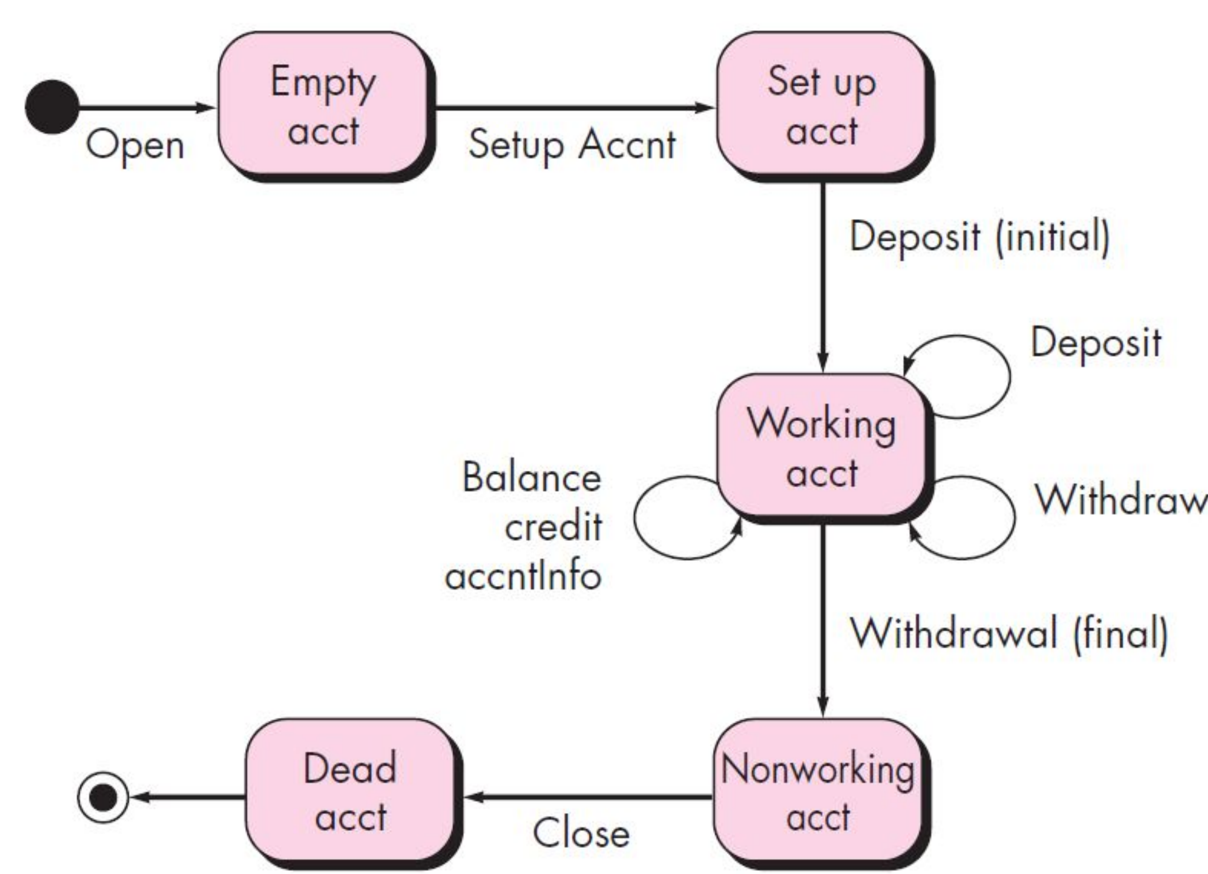


- Derive test cases directly from the chronological flow of each sequence diagram.
- Follow the exact message flow between the objects.
- Verify that messages are sent in the correct order, with the correct parameters, and that the objects return the exact expected results.

FIGURE 19.3

State diagram for the Account class

Source: Adapted from [Kir94].



Part 5

The Conversations Between Objects

Synthesizing Object-Oriented Testing

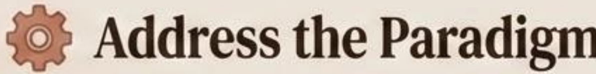
Summarizing the shift from traditional procedural testing to the OO paradigm.

The Broader View

Expanding the Scope of Testing

Start Early

- OO testing cannot wait for the code. It must begin at the Analysis and Design stages to catch fundamental architect-tural flaws.

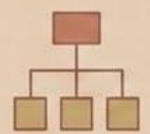


Address the Paradigm

- Your testing strategy must ex-plicitly account for the unique pillars of OO design:

Encapsulation

Hiding state requires testing the whole class.



Inheritance

Changes to parents ripple through subclasses.

Polymorphism & Dynamic Binding

Method execution is determined at runtime, requiring broader coverage.

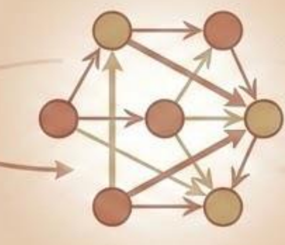
The OO Testing Strategy

Adapting the Core Levels

How we redefine the standard testing phases for an OO world:

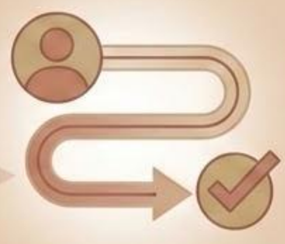
Unit Testing

The Class is the unit. We test the class as a whole, not individual methods in isolation.



Integration Testing

Focuses on collaborations. We use Thread-based (grouping by functionality) or Use-based (grouping by dependency layers) approaches.



Validation Testing

Strictly uses Use Case Scenarios to test end-to-end behavior from the user's perspective.

Targeted Testing Methods

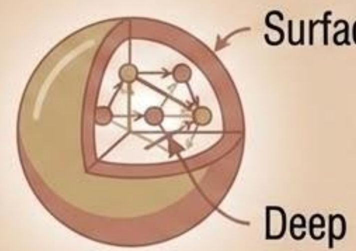
Hunting OO-Specific Faults

Fault-Based Testing

Proactively target the most likely OO-specific errors (e.g., initialization failures, improper inheritance).

Scenario-Based Testing

Execute real-world interactions driven directly by your use cases.

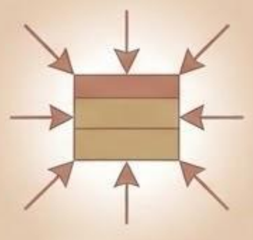


Surface vs. Deep Structure

Ensure you are testing both the external public interfaces (Surface) and the hidden internal collaborations (Deep).

Execution at the Class & Interclass Levels

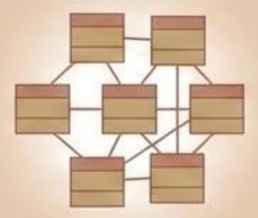
From Objects to Clusters



Class-Level Execution

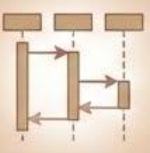
Random Testing: Throw random valid method sequences at an object to find unexpected interaction errors.

Partition Testing: Divide the input space by state, attribute, or category to test efficiently.



Interclass Execution

Focus heavily on clusters of collaborating classes.



Use established behavior models—like Statecharts and Sequence Diagrams—as your explicit test specifications.

The Final Thought

Bringing the System to Life

"Testing OO software is about more than testing code. It's about testing the interactions—the conversations between objects that bring the system to life."