



CSSE2003

Software Engineering Studio

Semester 2, 2009

**19: Object-oriented Testing &
Integration Testing**

Summary

- ◆ Features of object-oriented programs that affect testing.
- ◆ Integration testing:
 - integration testing concepts
 - integration test design patterns

Overview

- ◆ Systematic testing of object-oriented software is similar to systematic testing of procedural software, but the differences are worth considering:
 - typically much shorter methods
 - more encapsulated state
 - polymorphism and dynamic binding
 - generics
 - increased use of exception handling

State-dependent Behaviour

- ◆ OO programs are inherently dependent on state – methods typically depend not just on the parameter values but also the *state of the object*.
- ◆ This property makes testing more difficult since it is generally infeasible to test methods in isolation – the class becomes the unit for unit testing.

Encapsulation

- ◆ Encapsulating information via private or protected fields or methods makes it harder to design test cases and to decide whether the actual results are equivalent to the expected results.
- ◆ The effect of executing a method can be output, or a change of state that may not be externally observable.
- ◆ Test cases need to be able to cause objects to enter all possible states (controllability).
- ◆ Test oracles need to be able to check all outputs and states (observability).

Inheritance

- ◆ Inheritance allows us to define new classes as extensions or specialisations of existing classes.
- ◆ A child class can inherit fields and methods from its ancestors, *overwrite others*, and *add* new fields and methods.
- ◆ Test design must consider the effects of new or overridden methods, and distinguish methods that:
 - require new test cases
 - can be testing with existing test cases
 - need not be retested

Polymorphism and Dynamic Binding

- ◆ With object-orientation, variables can vary their type within a hierarchy defined by the declared type.
- ◆ Hence invocation of a method `var.f()` can be bound to different versions of `f()` depending on the type of the variable `var`.
- ◆ Such dynamic binding can affect the whole computation in which the invocation occurs.
- ◆ Tests must exercise **different bindings** to reveal potential failures that depend on a particular binding or on interactions between bindings for different calls.

Abstract Classes

- ◆ Abstract classes cannot be directly instantiated and testing.
- ◆ Testing must be applied to concrete subclasses.
- ◆ Not all subclasses may be defined if abstract class is part of a library that is to be sub-classed in the future.
- ◆ Testers need to be able to select *representative instances* of sub-classes to test an abstract class.

Exception Handling

- ◆ Exceptions were not originally introduced as an essential feature of object-orientation, but they are now considered central for modern object-oriented design methods.
- ◆ Exceptions cause non-trivial transfers of control (between the point where an exception is thrown and when it is handled), complicated by dynamic binding.
- ◆ Control flow associated with exception handling needs to be tested explicitly.

Concurrency

- ◆ Modern object-oriented languages and tool-kits encourage (if not require) multiple threads of control.
- ◆ Concurrency introduces new types of possible failures:
 - **deadlock** : two or more processes waiting for each other to release a resource
 - **livelock** : processes constantly change with regard to one another, none progressing
 - **race conditions** : process is dependent on the sequence or timing of other events
- ◆ Program behaviour can become non-deterministic, making testing much more problematic.

Integration Testing

- ◆ For systems with multiple components (classes, packages, subsystems , etc.), integration testing is concerned with how to organise testing of component interaction.
- ◆ It is normally preceded by unit testing of individual components and followed by system testing.
- ◆ Integration testing is a search for component faults that cause inter-component failures (required because testing individual components in isolation may not reveal such faults).
- ◆ Integration testing “is a dress rehearsal for system testing”.

Integration Strategy

- ◆ An integration strategy answers three questions:
 - Which components are the focus of the integration test?
 - In what sequence will component interfaces be exercised?
 - Which test design technique(s) should be used to exercise each interface?

Typical Integration Faults

- ◆ Wrong method called due to coding error or unexpected runtime binding.
- ◆ Violation of data integrity of a file or data database.
- ◆ Client sending a message that violates the precondition of a server's method.
- ◆ Wrong object bound to message.
- ◆ Wrong parameters or wrong parameter values.
- ◆ Inter-component conflicts.
- ◆ Resource contention.
- ◆ Configuration/version control problems.

Incremental Integration

- ◆ Experience has shown that incremental integration (add components a few at a time and then test their interoperability) is the most effective approach, since integrating all the components at one time makes debugging difficult since the defect may be in any interface.
- ◆ In contrast, with an incremental approach observed failures are likely to be in the most recently added components, making debugging more efficient.

Integration Testing and Architecture

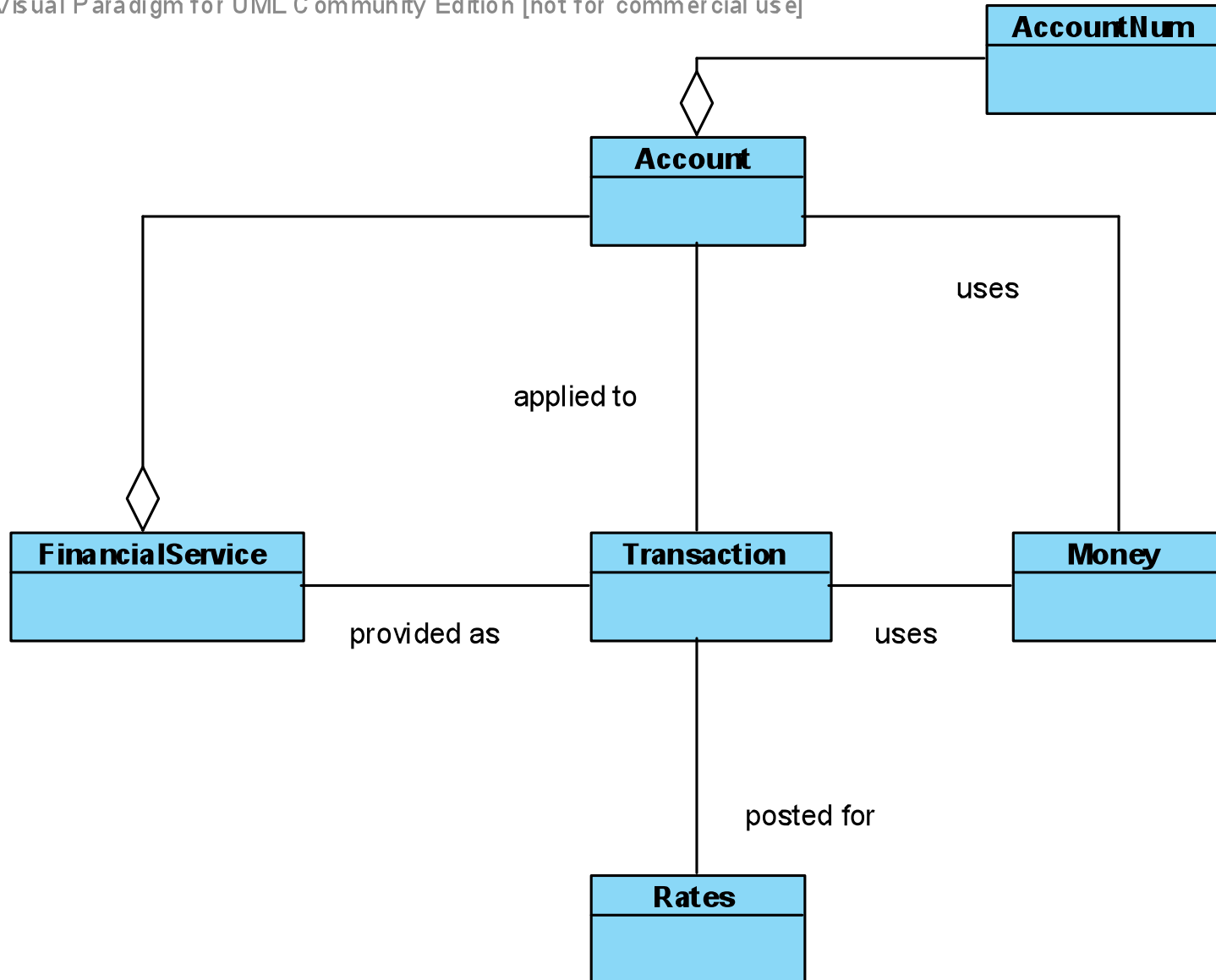
- ◆ Integration testing is closely tied to a system's architecture and its development strategy.
- ◆ The architecture organises the system into manageable components and subsystems.
- ◆ Integration test plan are usually based on inter-component implementation dependencies.
- ◆ Preparing the integration test plan often reveals errors, omissions and ambiguities in the requirements and architecture – motivation to start planning tests early in the project.

Dependency Analysis

- ◆ Component dependencies arise in many ways:
 - composition and aggregation
 - inheritance
 - calls to an API
 - objects used as message parameters
- ◆ Dependencies capture a “uses” relationship and can be represented in a dependency graph.

Example

Visual Paradigm for UML Community Edition [not for commercial use]



Integration Test Design Patterns

- ◆ Big-bang integration
 - all components at the same time
- ◆ Bottom-up integration
 - following usage dependencies
- ◆ Top-down integration
 - following application control hierarchy
- ◆ Collaboration integration
 - according to collaborations and their dependencies
- ◆ Layer integration
 - following layered architecture

Big-bang Integration

- ◆ Normally not recommended.
- ◆ Often adopted under schedule pressure at the end of a project.
- ◆ Applicable if:
 - system is stable and only a few component have been added or changed
 - system is small and components have been individually tested
 - components are tightly coupled so they cannot be exercised separately

Bottom-up Integration

- ◆ Add components to the system-under-test in uses-dependency order, beginning with components having the fewest dependencies.
- ◆ Works by moving from the leaves of the dependency tree to the root. If the tree contains n levels, there are n stages.
- ◆ Requires a driver for each component or component sub-tree.
- ◆ Minimises the need for test stubs, need to break dependency cycles or to simulate exceptions.
- ◆ Development and testing can often proceed in parallel.

Top-down Integration

- ◆ Add components to the system-under-test in control hierarchy order, beginning with the top-level control components.
- ◆ Implement the servers of this component as stubs.
- ◆ Continue by replacing stubs on one level by the corresponding full components with stubs for the next (lower) level of servers.
- ◆ A single driver is needed for the top-level.
- ◆ Stubs are required for all components except for the root(s) of the graph.
- ◆ Can be difficult to test lower-level component sufficiently.

Collaboration Integration

- ◆ Add sets of components to the system-under-test that are required to support a particular collaboration (often defined by use cases).
- ◆ Choose a set of collaborations that cover the dependency graph.
- ◆ Single top-level driver required.
- ◆ Stubs are required for untested components that do not participate in a collaboration.
- ◆ This approach may not exercise all component interfaces, focusing more on end-to-end functionality – important to explore exception cases.

Layer Integration

- ◆ Applicable to a system that can be modeled as a hierarchy that allows interfaces only between adjacent layers.
- ◆ Each layer is first tested in isolation.
- ◆ Perform top-down integration of layers, removing lower-level stubs in each stage (bottom-up integration is an alternative).
- ◆ Stubs and drivers are required for each layer.
- ◆ Popular for device interfaces and device drivers in embedded systems.

Follow-up Reading

- ◆ Mauro Pezzè and Michal Young, *Software Testing and Analysis: Process, principles and techniques*, Wiley, 2008 (Chapter 15)
- ◆ Robert V. Binder, *Testing object-oriented systems: models, patterns and tools*, Addison-Wesley, 2000