Testing Object-Oriented Software

7 December 1993

Presented by

Donald G. Firesmith

President, Advanced Software Technology Specialists (ASTS)

4018 South Harrison Street

Fort Wayne, IN 46807

voice: (219) 745-7928

fax: (219) 745-7928 or 747-9389

e-mail: 73664,3515@compuserve.com

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Introduction

Object-oriented software:

- Has a **radically different structure:**
  - Objects
  - Classes
  - Inheritance
  - Aggregation
- Is developed according to a **radically different development cycle:**
  - Incremental (a.k.a., recursive)
  - Iterative

These changes significantly impact software testing.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Impact of Errors on Software Engineering Goals

Testing finds errors that may cause deficiencies in the following goals of software engineering:

- **Correctness** – the degree to which the software meets its specified requirements
- **Efficiency** – the degree to which the software uses hardware resources effectively
- **Interoperability** – the degree to which the software correctly interacts with other software
- **Portability** – the ease with which the software can be transitioned to another hardware or software environment
- **Reliability** – the degree to which the software behaves correctly over time

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Impact of Errors on Software Engineering Goals - 2

- **Robustness** – the degree to which the software continues to function correctly under abnormal circumstances.
- **Safety** – the degree to which the software works without accidental harm to life or property.
- **Security** (a.k.a., integrity) – the degree to which the software protects itself from unauthorized access or modification.
- **Testability** – the degree to which, and the ease with which, the software can be effectively tested.
- **User-friendliness** – the ease with which humans can use the software.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Levels of Testing

Testing can be performed at different levels of abstraction:

- The **resources** of objects and classes (sub-unit testing)
- Individual **objects and classes** (unit testing)
- **Subassemblies** of interacting objects and classes (integration testing)
- Hierarchies or graphs (integration testing):
  - **Inheritance hierarchies or graphs** of classes
  - **Aggregation hierarchies** of objects and classes
  - **Scenarios** of collaborating objects
- **Assemblies** of subassemblies (SW integration and acceptance testing)
- **Systems** of assemblies, hardware, people, and documentation. (system acceptance testing)

Copyright © 1992, 1993 by Advanced Software Technology Specialists

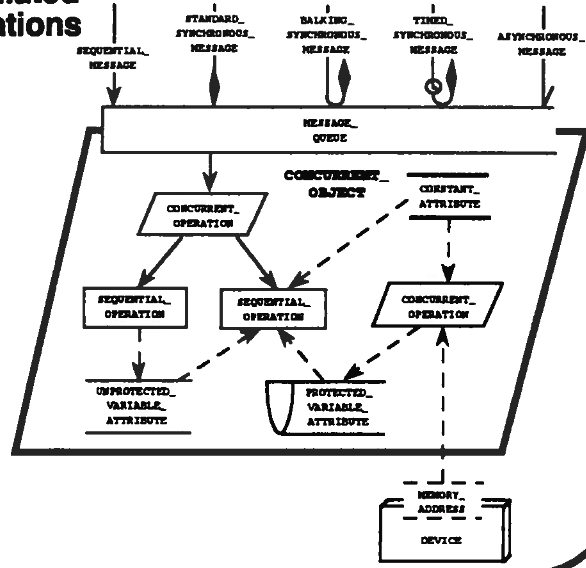
Levels of Testing - 2

Object-oriented software primarily impacts testing at the level of:

- Individual **objects and classes**
- Subassemblies of interacting objects and classes (integration testing)
- Hierarchies or graphs (integration testing):
  - **Inheritance hierarchies or graphs** of classes
  - **Aggregation hierarchies** of objects and classes
  - **Scenarios** of collaborating objects

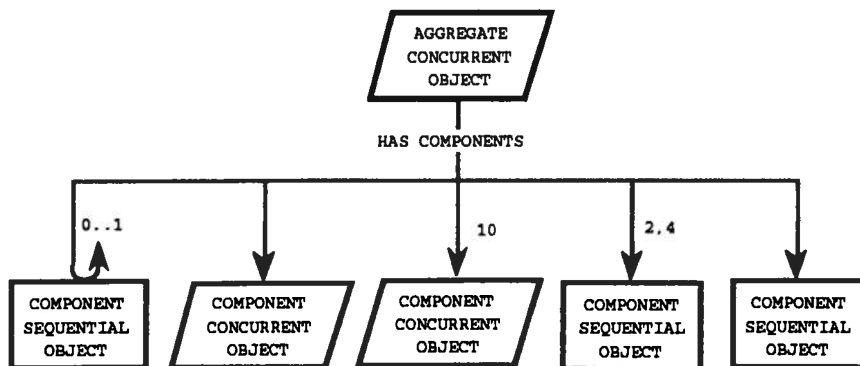
Copyright © 1992, 1993 by Advanced Software Technology Specialists

Structure of Objects in Terms of Encapsulated Attributes and Operations



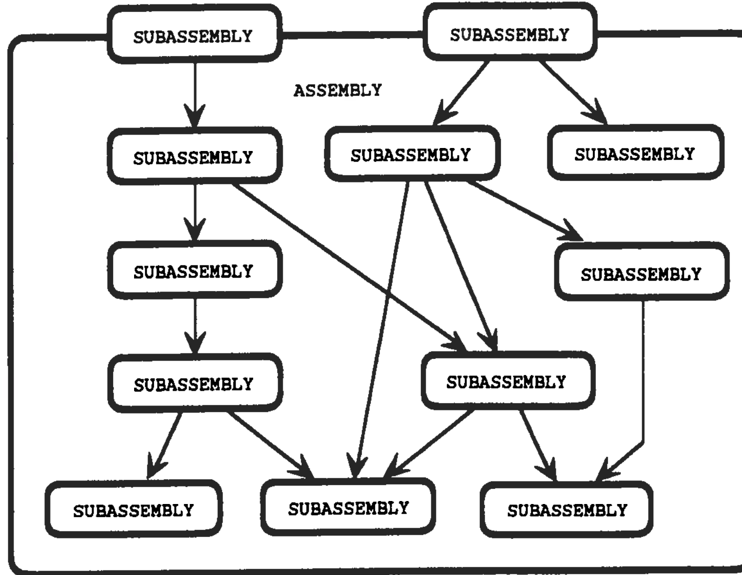
Copyright © 1992, 1993 by Advanced Software Technology Specialists

Structure of Aggregate Object in Terms of Encapsulated Objects



Copyright © 1992, 1993 by Advanced Software Technology Specialists

Structure of OOSW in Terms of Subassemblies



Copyright © 1992, 1993 by Advanced Software Technology Specialists

Procedural Software	Object-Oriented Software
Units = Functions	Units = Classes (and Objects)
Unit Testing = Function Testing	Unit Testing = Class and Object Testing
White-box Testing Emphasized	Black-box Testing Emphasized
Common Global Data	Common Local Attributes, Global Objects and Classes
Functional Decomposition	Recursion and Inheritance
Functional Cohesion	Class/Object/Assembly Cohesion
Data and Call Coupling	Inheritance and Message Coupling
Subroutine (Function) Calls	Message Passing
Entity Relationship Diagrams	Semantic Nets
Data Flow Diagrams	Blackbox Interaction Diagrams
Structure Charts	Whitebox Interaction Diagrams
Reuse is Primarily Incidental	Reuse is Central
Primarily Top-Down Development	Top-Down and Bottom-Up Development
Waterfall Development Cycle	Incremental Iterative Development Cycle

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Structure Impacts Unit Testing

Procedural Software:

Unit testing of functionally decomposed software was the testing of individual, functionally-cohesive operations

Object-Oriented Software:

The testing of individual operations and attributes is only part of the unit testing of objects and classes because the meaning and behavior of these encapsulated resources depend on the other resources with which they are encapsulated

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Integration Testing Becomes Unit Testing

Procedural Software:

Integration testing of functionally decomposed software was the testing of an integrated set of operations and common global data

Object-oriented unit testing is the:

- Independent testing of individual objects and classes using test software (e.g., test drivers and test stubs) and lowerCASE tools (e.g., debuggers, profilers, performance analyzers)
- Testing of a logically related set of operations and data within an object or class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Additional Testing Implications of OO Software

Component objects are encapsulated within aggregate objects making them difficult to test directly.

Production-quality debuggers are critical because objects encapsulate their resources, which are ordinarily accessible only via their interface protocol consisting of messages, visible attribute types, and exceptions.

Test drivers must send appropriate messages to the associated objects under test and must be able to properly handle both data returned as parameters of messages and exceptions raised by the objects under test.

Test stubs must handle and potentially log messages sent by the objects under test.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

4.1) Testing Objects

Objects are models of **application domain entities** that typically:

- Are **instances of classes**
- **Encapsulate** (i.e., localize and hide):
  - Attribute types
  - **Attributes (data)**
  - **Operations (functionality)**
  - Exceptions
  - Other objects
- Send and receive **messages**

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Built-In Test (BIT)

Objects should:

- Be **responsible for maintaining their own abstraction**
- **Implement their own built-in test (BIT)**
- Not rely on some massive BIT function with its tenticles violating the encapsulation of all objects

At least three object-oriented approaches to BIT, each with its own pros and cons, are currently used in industry:

1) **concurrent BIT operations**
2) **assertions and exceptions**
3) **BIT messages**

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Concurrent BIT Operations

First approach:

- Objects actively and continually perform self checks using concurrent BIT operations
- This approach may be necessary for certain life-critical, real-time applications (e.g., where objects model critical hardware devices that may fail)
- This involves significant overhead and typically not cost-effective for most objects and applications

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Objects	Priority	Verification Technique(s)
Failure to meet the allocated requirements	Critical	Black-box testing
Abstraction violated	Critical	Black-box and white-box testing of assertions and exceptions, and stress testing using multiple messages
Invariants violated or missing	Critical	White-box testing of assertions
Timing problems such as: – Deadline missed – Inaccurate timing – Imprecise timing	Critical	Testing with a performance analyzer (Also verified at the operation level and during integration testing of assemblies and scenarios)
State modeling problems such as: – Missing or improper states, transitions, or guard conditions – State model inconsistent with state attributes, links, and component objects – Transitions inconsistent with operations – Unreachable states – Transition from end state	Critical	Inspection, black-box testing, white-box testing of preconditions and postconditions, and stress testing using multiple messages
Garbage collection errors such as: – Early reclamation (e.g., dangling pointers in C++) – Late or missing reclamation (i.e., memory leakage or component objects not reclaimed)	Critical	Inspection and debugging with run-time tools

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Objects - 2	Priority	Verification Technique(s)
Inadequate achievement of software engineering goals or inadequate use of software engineering principles	High	Inspection and black-box testing (Typically verified at the class level)
Concurrency problems such as: – Inappropriate concurrency – Missing concurrency – Incorrect priorities – Unnecessary polling	High	Inspection and testing with a concurrent debugger and a performance analyzer (Also verified at the class level and during integration testing of assemblies and scenarios)
Incorrect initialization or missing reinitialization	High	Black-box testing (Typically verified at the class level)
Failures associated with messages, exceptions, attributes, or operations	High	See appropriate tables
Persistence problems: – Object not stored when necessary – Object stored unnecessarily	Medium	Inspection and black-box testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the object	Medium	Inspection
Links not implemented by messages or attributes	Medium	Inspection and black-box testing (error guessing)
Violation of design or coding standards	Medium	Inspection, standard checker, and pretty printer (Typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Testing Issues Concerning Classes - 2

Classes may not be executable.

- Templates only
- Generic (or parameterized class)
- Must test indirectly via instances
- Regression testing

Testing of classes can be **black-box** and **white-box**:

- Black-box testing involves testing the class's interface (a.k.a., protocol)
- White-box testing involves testing the class's implementation (e.g., attributes and operations)

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Classes	Priority	Verification Technique(s)
Failure to meet the allocated requirements	Critical	Black-box testing
Abstraction Violated	Critical	Black-box and white-box testing of assertions and exceptions, and stress testing using multiple messages
Invariants violated or missing	Critical	White-box testing of assertions
Incorrect state model	Critical	Inspection, black-box testing, and white-box testing of preconditions and postconditions
Invariants violated	Critical	White-box testing of assertions
State modeling problems such as: – Missing or improper states, transitions, or guard conditions – State model inconsistent with state attributes, links, and component objects – Transitions inconsistent with operations – Unreachable states – Transition from end state	Critical	Inspection, black-box testing, white-box testing of preconditions and postconditions, and stress testing using multiple messages
Failures associated with inheritance	Critical	Integration testing (See inheritance table)
Failures associated with both instantiation (e.g., incompatible generic parameters)	Critical	Black-box testing (See classification table)
Garbage collection errors such as: – Early reclamation (e.g., dangling pointers in C++) – Late or missing reclamation (i.e., memory leakage or component objects not reclaimed)	Critical	Inspection and debugging with run-time tools

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Exported Resources

A **message** is a call, possibly bound at runtime, to an object or class that:

- Requests a **service**
- Provides **data**
- Provides notification of an **event**

An **exception** is an error condition identified and raised by the operations of objects and classes so that they can be properly handled by calling operations, possibly in other objects or classes.

A **role** is an inheritable subset of the protocol that can be accessed as a group and used to restrict client access to only certain resources in the protocol.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Sender Errors Associated with Messages	Priority	Verification Technique(s)
Failure to meet the allocated requirements	Critical	Integration testing
Message sent to wrong receiver(s)	Critical	Integration testing
Wrong type of message (e.g., sequential, synchronous, asynchronous)	High	Inspection and integration testing
Concurrency problems such as unnecessary polling	High	Inspection, integration testing, and testing with a performance analyzer
Incorrect message priority	High	Inspection and integration testing
Incompatible actual parameters in the message for all formal parameters required by the corresponding message exported by the receiver	Medium	Inspection, compilation, and integration testing
Message not declared in the interface (i.e., protocol) of the receiver(s)	Medium	Inspection, compilation, and integration testing
Association not implemented by message	Medium	Inspection and black-box testing (error guessing)
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the message	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (Typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

4.4) White-Box Testing of Implementations

White-box testing is the testing of an individual unit (e.g., object or class) using knowledge of both the unit's implementation (including subunits) and its interface.

White-box testing tests:

- Individual encapsulated **attributes** and **operations**
- Their interactions via control flows and data flows

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Encapsulated Attributes and State

An **attribute** is a single, discrete, and inherent data abstraction that captures a characteristic, property, trait, quantity, quality, or association of an object or class.

An attribute either:

1) Describes its object or class
2) Stores [part of] the state of its object or class
3) Implements a simple binary association between its object or class and another object or class

Attributes can be either constants or variables.

An attribute is an instance of an attribute type.

State is either the value(s) of:

- Only those attribute(s) that determine the overall behavior of an object or class
- All attributes

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Encapsulated Operations

An **operation** is a discrete activity, action, or behavior that:

- Implements a functional (i.e., sequential) or process (i.e., concurrent) abstraction
- Is typically performed by, belongs to, and is encapsulated in an object or class
- Is either a:
  - Constructor
  - Destructor
  - Modifier
  - Preserver

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Operations	Priority	Verification Technique(s)
Failure to meet requirements regarding operations	Critical	White-box unit testing
Message not received by correct operation or message received by incorrect operation	Critical	White-box unit testing or compilation (if language requires one-to-one mapping)
Precondition, postcondition, or invariants violated (e.g., operation executed while in an incompatible state)	Critical	Assertions and white-box unit testing
Operation incorrectly performed	Critical	White-box unit testing
Corruption of attributes due to lack-of critical regions or concurrent access	Critical	Integration testing
Inaccurate timing or missed deadline	Critical	White-box unit testing using a performance analyzer
Proper value not returned or improper value returned by operation	High	White-box unit testing
Correct exception not raised by correct operation	High	White-box unit testing
Correct exception not handled by correct operation	High	White-box unit testing
Improper state following raising of exception	High	White-box unit testing
Improper priority of operation	High	Integration testing
Unreachable statements (e.g., dead code)	Medium	White-box unit testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the operation	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (Typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

4.5.1) Testing Classification and Inheritance

Definition: **Classification** is the mechanism that allows an object to reuse the resources from one or more classes (i.e., the relationship between an instance and each class that defines it).

Objects are classified into (i.e., are instances of) one or more classes.

Classes are classified into one or more metaclasses.

Classification may be:

- Dynamic or static
- Multiple or single

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Inheritance

Definition: **Inheritance** is the mechanism by which a class reuses the resources from one or more other classes (i.e., by which the subclass inherits the resources of its superclasses).

The subclass may:

- Add additional resources
- Modify or replace inherited resources
- Delete inherited resources

Inheritance is usually used to build new extensions or specializations of existing generalizations.

Inheritance may be:

- Dynamic or static
- Multiple or single

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Inheritance and Classification - 2	Priority	Verification Technique(s)
Original resource in superclass not overwritten in subclass	High	Integration testing
Original resource in superclass not deleted in subclass	High	Integration testing
Ad hoc support for dynamic classification fails	High	Integration testing
Deferred resource not provided by subclass of deferred superclass	High	Integration testing
Incompatibility between subclass resources and existing superclass resources (e.g., common local data problems with attributes or violation of the state model)	High	Regression testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the operation	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (Typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

4.5.2) Testing Scenarios and Subassemblies

Definition: A **scenario** is a logically cohesive set of interactions (e.g., message passing and operation execution) among objects and classes.

Scenarios often provide a required capability.

Scenarios are used for analysis, design, and/or testing purposes.

Scenarios may involve more than one thread of control.

Scenarios are often, but need not be, initiated by some client terminator.

Scenarios are sometimes called *use cases*.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Subassemblies – 2

Subassemblies are sometimes called:

- Clusters
- Kits
- Subjects
- Subsystems

Subassemblies are developed:

- Recursively (i.e., incrementally):
  - Top-Down
  - Bottom-Up
  - Outside-In
- Iteratively

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Order of Testing

Subassemblies are often tested in the order in which they are developed:

- Top-Down
- Bottom-Up
- Outside-In

Individual subassemblies may be incrementally tested if any of their objects and classes must be temporarily stubbed due to top-down recursive development.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

4.5.3) Testing Aggregates

Objects may be aggregates, composed of component objects.

Component objects may be exported or they may be encapsulated within the aggregate object.

Encapsulated component objects are typically not accessible directly, but only via messages to the aggregate objects that contain them.

The state of the aggregate object may be determined by the states of its component objects.

The component objects must be unit tested, and the aggregate object must be integration tested to find errors and inconsistencies in the integration of the components into the aggregate.

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Errors Associated with Aggregates	Priority	Verification Technique(s)
Failure to meet requirements of the aggregate	Critical	Integration testing
Components missing	Critical	Inspection and integration testing
Inconsistent components	Critical	Inspection, assertions, and integration testing
Components not created or initialized when aggregate created or initialized	Critical	Assertion and integration testing
Incorrect visibility of components	High	Inspection and black-box testing of aggregate
Components not destroyed when aggregate destroyed	Medium	Assertion and integration testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements to the parts of the aggregate	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (Typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Copyright © 1992, 1993 by Advanced Software Technology Specialists

5) Test Completeness Criteria

Testing cannot identify and locate all errors.

When is testing complete?

When should testing stop?

- Completion criteria
- Budget and schedule

This section covers test completion criteria for:

- Unit Testing (Objects and Classes)
- Integration Testing:
  - Inheritance Hierarchies
  - Scenarios and Subassemblies
  - Aggregations

Copyright © 1992, 1993 by Advanced Software Technology Specialists

5.1) Test Completeness Criteria for Objects and Classes

- **Inspection**:
  - —Code compared with the relevant documentation (e.g., Software Requirements Specification and Design Document)
  - —Code manually inspected including:
  - – Interface
  - – Implementation
  - – Separate subunits (e.g., operations)
- **Quality tools** used:
  - —Pretty printer for source code
  - —Standards checker
  - —Compiler for syntax checking

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Test Completeness Criteria for Objects and Classes - 4

— **Performance testing.** Timing information is captured as objects are executed using performance analyzers, profilers, and (concurrent) debuggers.

— **Instance testing.** If a class has only a single instance, it is tested. If a class has multiple instances, a representative instance will be chosen and tested. If a generic (i.e., parameterized) class has multiple instances, then a representative instance from each equivalence class (based on parameters) is chosen and tested.

— **Context testing.** Objects shall be tested in the context in which they will occur (e.g., during integration testing):

- Objects and classes shall accept all relevant messages from their clients
- Objects and classes shall raise all relevant exceptions to their clients
- Objects and classes shall send all relevant messages to their servers
- Objects and classes shall handle all relevant exceptions raised by their servers
- Objects and classes shall protect their attributes from corruption in an concurrent environment due to the interleaving of operation execution because of time-slicing

Copyright © 1992, 1993 by Advanced Software Technology Specialists

Test Completeness Criteria for Objects and Classes - 5

- **White-box testing**:

  — **Completeness testing.** Based on the resources encapsulated in the object, test cases should be developed to meet the following completeness criteria:

  - Every **operation** should be executed (e.g., dead operation elimination)
  - Every **variable attribute** should be updated by the appropriate operation(s)
  - Every **constant attribute** should be read by the appropriate operation(s)
  - Every **object-internal scenario** involving multiple operations and attributes should be executed (e.g., basis path testing using operation and possible attributes as nodes)
  - Every **out-going exception** should be raised
  - Every **in-coming exception** should be properly handled

Copyright © 1992, 1993 by Advanced Software Technology Specialists