Testing Object-Oriented Software

3 March 1993

Presented by

Donald G. Firesmith

President, Advanced Software Technology Specialists (ASTS)

17124 Lutz Road

Ossian, Indiana 46777

voice: (219) 639-6305

fax: (219) 747-9389

Director of Object Technology, Software Consulting Specialists (SCS)

P.O. Box 80310

Fort Wayne, Indiana 46898

voice: (219) 432-3975

fax: (219) 432-3651

Introduction

Object-oriented software:

- Has a **radically different structure:**
  - Objects
  - Classes
  - Inheritance
- Is developed according to a **radically different development cycle**

These changes significantly impact software testing.

Topics Covered in this Paper

1) Definition and Purpose of Testing
2) Structure of Object-Oriented Software
3) Testing Object-Oriented Software
   - 3.1) Testing Objects
   - 3.2) Testing Classes
   - 3.3) White-Box Testing
   - 3.4) Black-Box Testing
   - 3.5) Integration Testing
4) Conclusion

1) Definition and Purpose of Testing

Testing cannot prove software is correct.

Testing can prove software is incorrect.

Definitions:

- **Software testing** is the execution of software for the explicit purpose of finding errors that cause failures
- **Failures** are deviations from the actual and intended structure or behavior of the software
- **Object-Oriented Testing (OOT)** is the appropriate testing of object-oriented software

OOT is designed to find specific types of errors that occur in object-oriented software because of its specific structures and behavior.

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

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.
- **User-friendliness** – the ease with which humans can use the software.

Limitation of Testing

Testing cannot find all errors.

Testing should therefore:

- Maximize the number of errors found (prioritized by probability of occurrence and severity of impact)
- Minimize the effort (in terms of cost and schedule) required to prepare, run, and evaluate the results of the testing

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)

Levels of Testing - 2

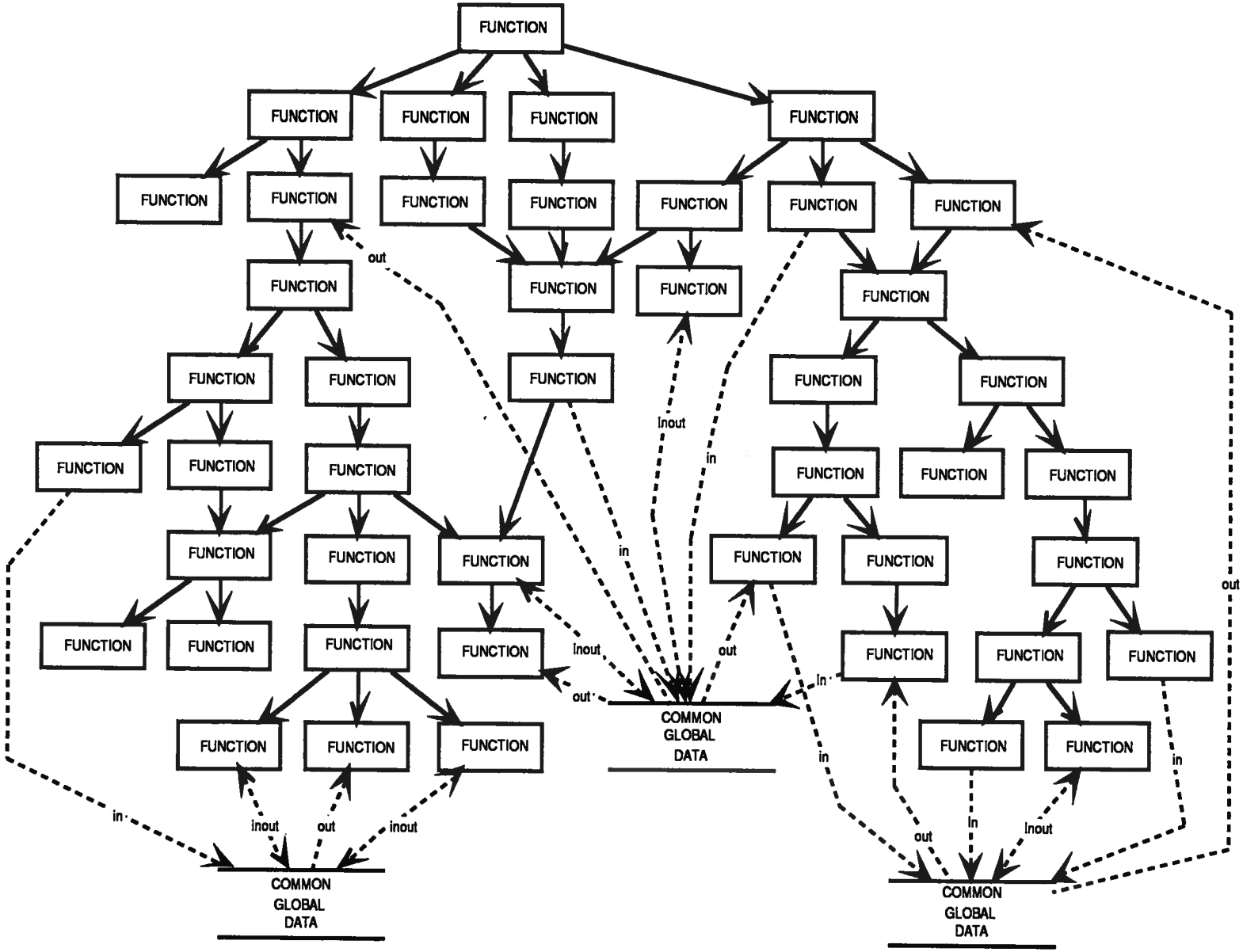
Object-oriented software primarily impacts testing at the level of:

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

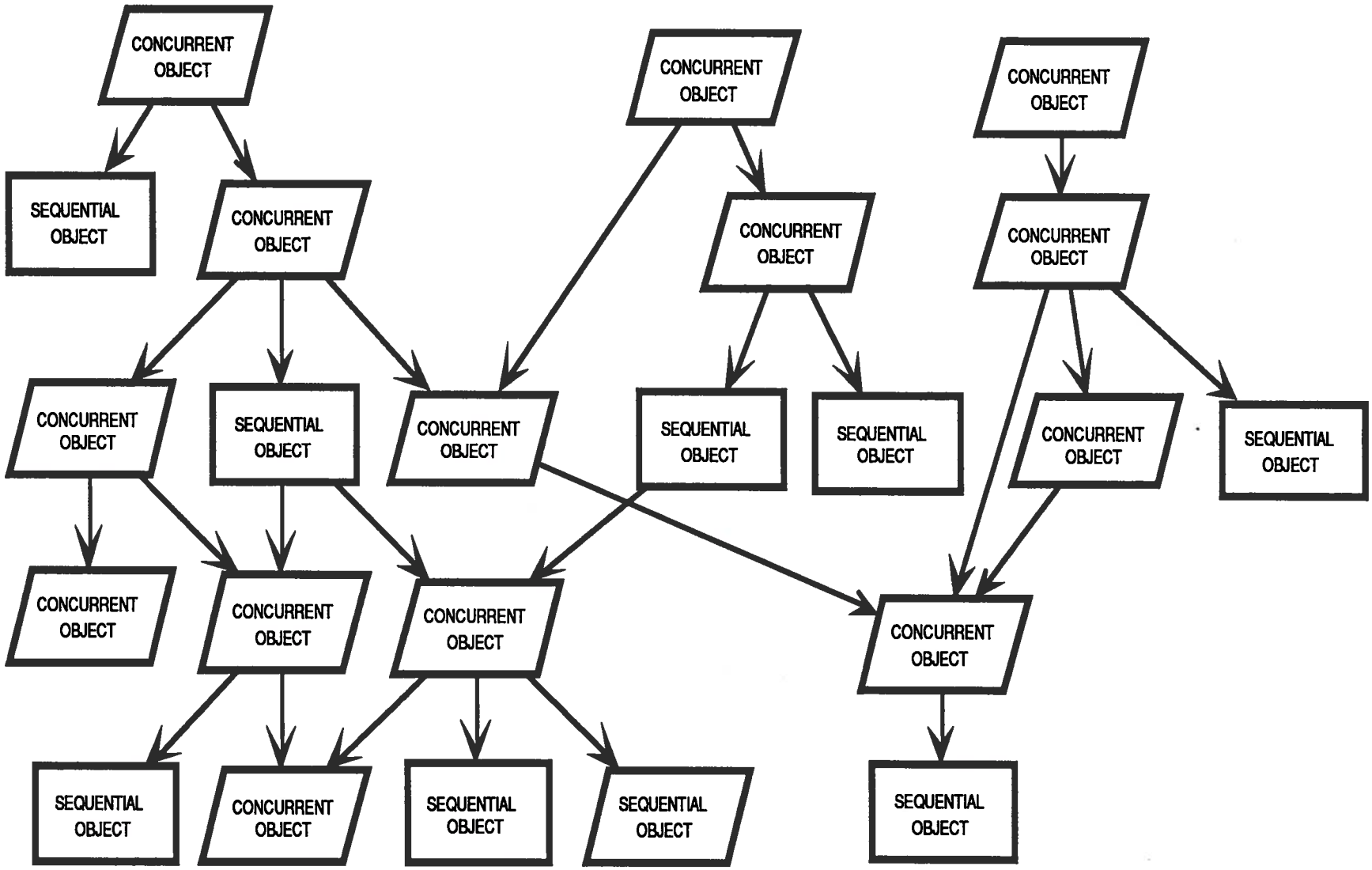
Procedural versus Object-Oriented Software

Procedural Software	Object-Oriented Software
Units = Functions	Units = Classes (and Objects)
Common Global Data	Common Local Attributes
Functional Decomposition	Recursion and Inheritance
Functional Cohesion	Object/Subassembly Cohesion
Data Coupling	Inheritance/Message Coupling
Subroutine (Function) Calls	Message Passing
Entity Relationship Diagrams	Semantic Nets
Data Flow Diagrams	Control Flow Diagrams
Structure Charts	Interaction Diagrams
Primarily Top-Down	Top-Down and Bottom-Up
Primarily Waterfall Development Cycle	Incremental Iterative Development Cycle

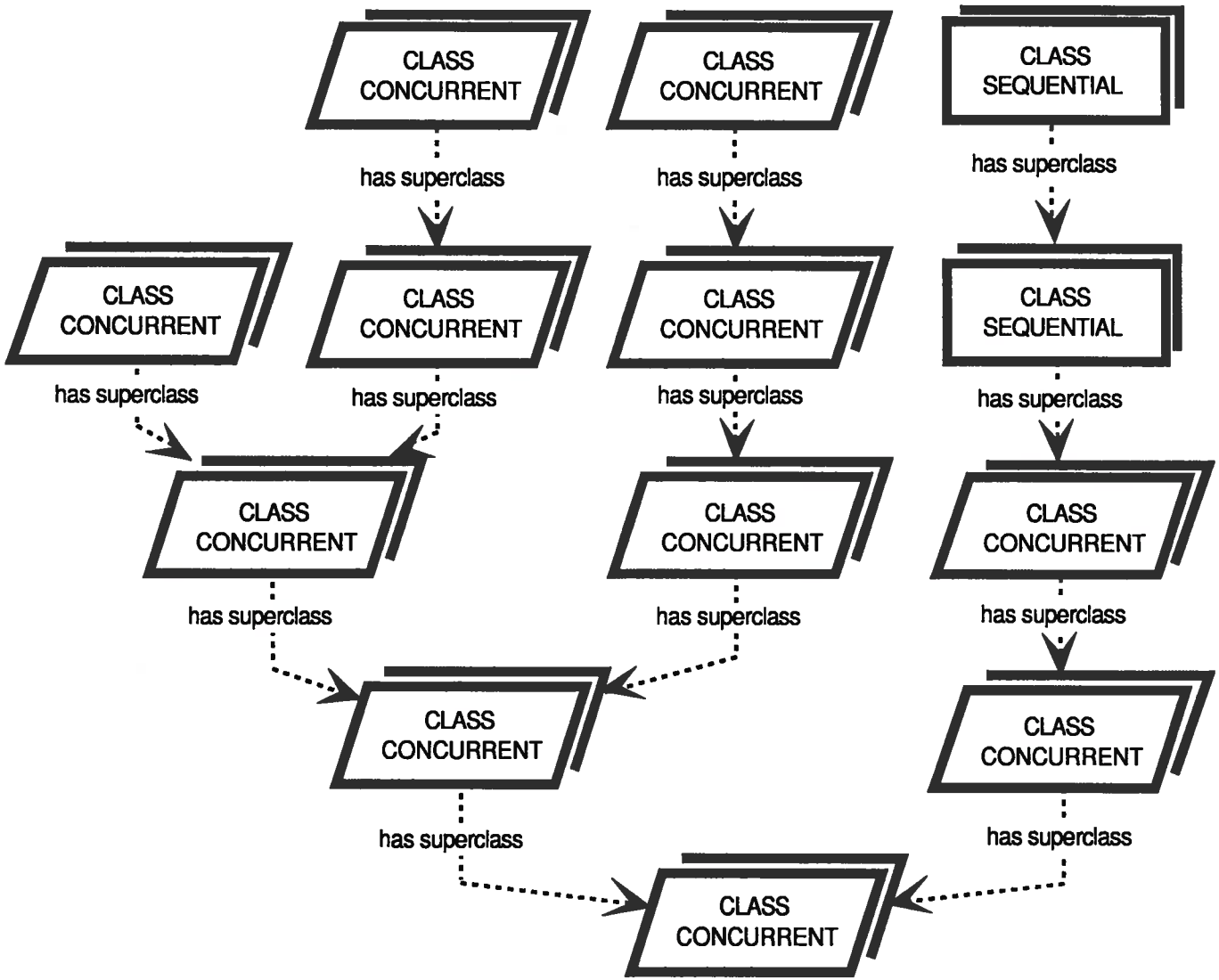
Structure of Functional Software



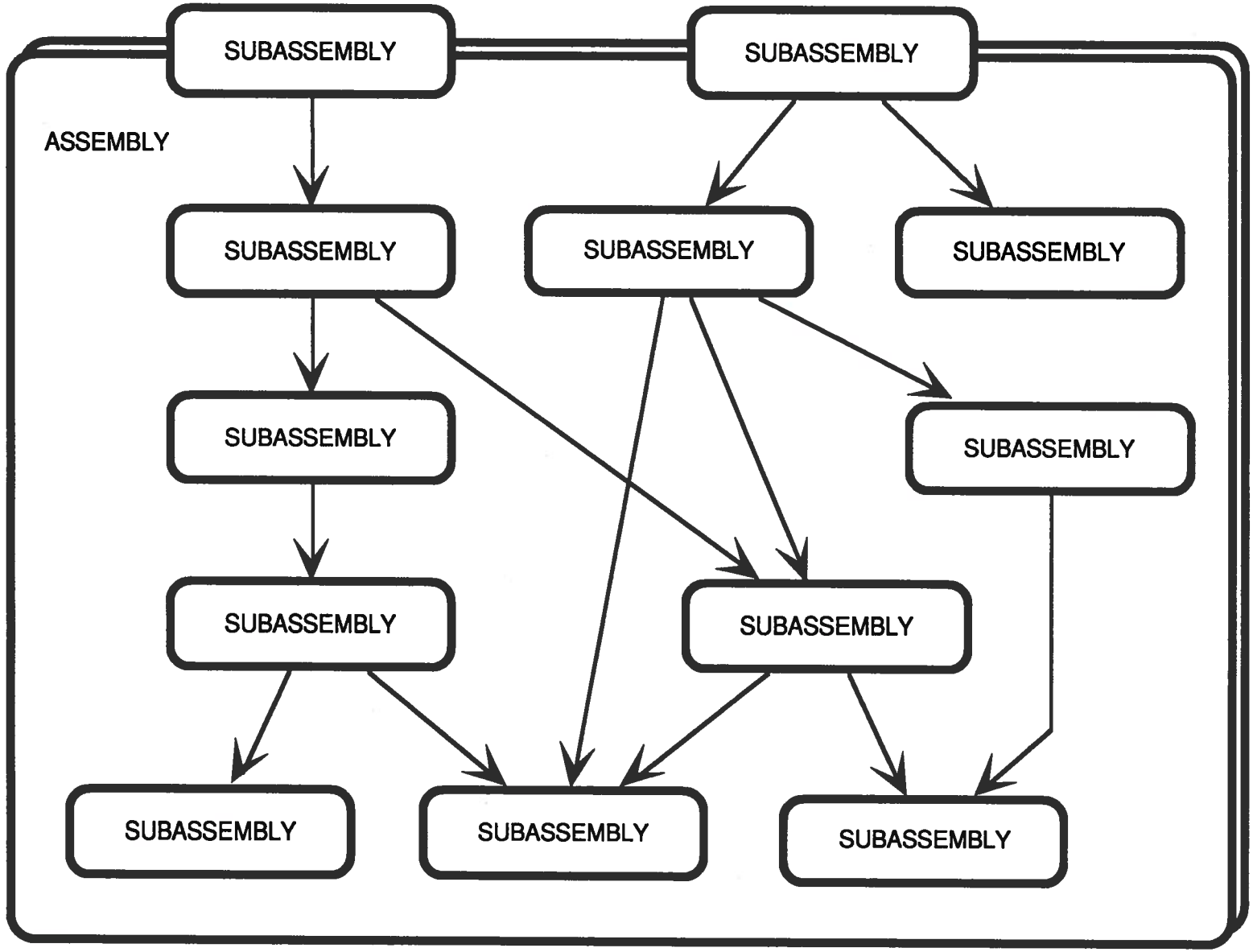
Structure of OOSW in Terms of Objects



Structure of OOSW in Terms of Classes



Structure of OOSW in Terms of Subassemblies



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

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

Testing Implications of Object-Oriented Software

Common global data bugs no longer exist because common global data no longer exists; all data (i.e., attributes) are now local within objects and classes.

Basis path (or structured) testing:

- Is limited to pieces of units (i.e., operations of objects)
- Must address exception handling and concurrency issues (if the object is concurrent)

Equivalence class testing:

- The importance of equivalence class and black-box testing is emphasized because objects can be considered software black-boxes
- Equivalence classes of test data are defined in terms of messages and the protocols of the objects

Testing Implications of Object-Oriented Software

Boundary value testing is of limited value when the object-oriented language is strongly typed and proper data abstraction is used to restrict the values of attributes.

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.

3) Testing Object-Oriented Software

1) Testing Objects
2) Testing Classes
3) Black-box Testing of Interfaces:
   - Messages
   - Exceptions
4) White-box Testing of Implementations:
   - Attributes and State
   - Operations
5) Integration Testing:
   - Subassemblies
   - Testing Scenarios
   - Classification and Inheritance

3.1) 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**

Testing Issues Concerning Objects

Objects are bigger units than functions.

Unit testing:

- Is primarily the testing of entire objects
- Is similar to the integration testing of functions and data
- Has test cases consist of messages (or possibly interrupts) rather than data.

Input-Processing-Output takes on new meaning.

Concurrent objects:

- Behave differently from sequential objects
- May not receive messages

Testing Issues Concerning Objects - 2

Encapsulation hides:

- Attributes
- Operations
- Component objects in aggregates

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

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

Common Errors Associated With Objects

Testers should test for the following errors associated with objects:

- **Abstraction violated**
- **Persistence problems:**
  - Object not stored when necessary
  - Object stored unnecessarily
- **Documentation** out of sync with the code
- **Incorrect state behavior** (operations versus state)
- **Invariants violated**
- Failures associated with:
  - **Instantiation**
  - **Destruction**

Common Errors Associated with Objects - 2

- **Concurrency problems** such as:
  - Unnecessary polling
  - Starvation
  - Deadlock
  - Priority inversion
  - Lack of mutually exclusive access to attributes
  - Incorrect message type:
    - Sequential
    - Synchronous
    - Asynchronous
  - Incorrect priority of:
    - Messages
    - Message queues
    - Concurrent operations

Common Errors Associated with Objects - 3

- **Inadequate performance:**
  - Excessive memory utilization
  - Memory not deallocated when finished
  - Inadequate performance
  - Missed deadlines
- Failures to meet:
  - **Requirements**
  - **Responsibilities**
- Failures due to:
  - **Polymorphism**
  - **Overloading**

Common Errors Associated with Objects -4

- Failures associated with:
  - **Messages**
  - **Exceptions**
  - **Attributes:**
    - Common local data
    - Type violations
  - **Operations:**
    - Preconditions
    - Postconditions
    - Correctness
    - Raising exceptions and exception handlers

Black-Box (Interface or Protocol) Testing

Allocated **Requirements and Responsibilities**

External state behavior

Attribute type visibility:

- Type available for message parameters
- Type implementation hidden

Message coverage:

- Every message (including interrupts) received
- Combinations of messages to determine synergistic effects
- **Equivalence classes** and **boundary values** of message parameters

Message correctness

Exception coverage:

- Every exported exception raised
- Every imported exception handled

Exception correctness

White-Box (Implementation) Testing

Operations:

- Operation coverage:
  - Basis path testing
  - State coverage
- Operation correctness
- Allocation of messages to operations
- Exceptions raised and handled

Attributes:

- Attribute coverage:
  - All attributes used
  - All variable attributes modified
- Attribute correctness
  - Type violations
  - Mutual exclusion

3.2) Testing Classes

Classes are templates for the construction (i.e., **instantiation**) of objects.

Classes:

- **Define and encapsulate** (i.e., localize and hide) the **resources of their instances**
- Send and receive **messages**

Classes may also have:

- **Class attributes** (e.g., number of instances)
- **Class operations** (e.g., construct an instance)

Testing Issues Concerning Classes

Classes are also bigger units than functions.

Unit testing:

- Is primarily the testing of entire classes
- Is similar to the integration testing of functions and data
- Unit test cases consist of messages (or possibly interrupts) rather than data

Encapsulation hides:

- Attributes (class and instance)
- Operations (class and instance)
- Component objects in aggregates

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)

Common Errors Associated With Classes

Testers should test for the following errors associated with classes:

- **Abstraction violated**
- **Persistence problems:**
  - Instances not stored when necessary
  - Instances stored unnecessarily
- **Documentation** out of sync with the code
- **Incorrect state behavior** (operations versus state)
- **Invariants violated**
- Failures associated with:
  - **Instantiation of instance**
  - **Instantiation of generic**

More Common Errors Associated with Classes

Testers should also test for the following errors associated with objects:

- Failures associated with **inheritance**
- Failure to meet **required behavior** of the class
- Failures associated with class and instance:
  - Messages
  - Exceptions
  - Attributes
  - Operations

3.3) Black-Box Testing of Interfaces

Black-box testing is the testing of a unit (e.g., object or class) using knowledge of only the unit's interface and not its implementation.

An **interface** is the outside, user view of an object or class, consisting of its context and protocol.

A **protocol** is the declaration of the exported resources of an object or class. It declares its visible attribute types, constant attributes, messages, exceptions, and roles.

Exported Resources

A **message** is a call, possibly bound at runtime, to an object or class that (1) requests a service, (2) provides data, or (3) 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.

3.4) 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

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, or (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

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

Testing Issues with Encapsulated Resources

Attributes and Operations cannot be tested in isolation because:

- They are hidden within the implementations of their objects and classes
- They depend on their environment:
  - Exception definitions
  - Synergistic interactions among resources

Good debuggers are needed.

Embedded test software has potential problems:

- Trapdoors
- Delivery of software modified after testing

3.5) Integration Testing

This section covers the integration of object-oriented software including the testing of:

- **Subassemblies**
- **Scenarios**
- **Classification and Inheritance**

Objects and classes are not islands.

Objects and classes **collaborate** by:

- **Sending messages**
- **Raising and handling exceptions**
- **Providing visibility to attribute types** (when strongly typed languages are used)

Several objects working together are often necessary to fulfill a single customer requirement.

3.5.1) Testing Subassemblies

Analyze a little, design a little, code a little, test a little, integrate a little, document a little.

Definition: A **subassembly** is a small, logically cohesive collection of objects, classes, and subassemblies that is analyzed, designed, coded, tested, integrated, and documented as a group.

Definition: An **assembly** is a logically-cohesive collection of software that:

- Is typically identified during system design
- Is a major software configuration item
- Typically implements a significant, cohesive set of system requirements
- Typically consists of multiple subassemblies identified during software requirements analysis and design

Testing Subassemblies – 2

Subassemblies are sometimes called:

- Clusters
- Kits
- Subjects
- Subsystems

Subassemblies are developed:

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

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.

Common Errors Associated with Subassemblies

Testers should test for the following errors associated with subassemblies:

- Correct message passed to the wrong object
- Incorrect message passed to the right object
- Correct exception raised to the wrong object
- Incorrect exception raised to the right object
- Concurrency problems such as:
  - Unnecessary polling
  - Starvation
  - Deadlock
  - Lack of mutually exclusive access to attributes
  - Priority inversion
  - Incorrect priorities and message types
- Inadequate performance and missed deadlines

3.5.2) Testing Scenarios

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*.

Common Errors Associated with Scenarios

Testers should test for the following errors associated with scenarios:

- Correct message passed to the wrong object
- Incorrect message passed to the right object
- Correct exception raised to the wrong object
- Incorrect exception raised to the right object
- Concurrency problems such as:
  - Unnecessary polling
  - Starvation
  - Deadlock
  - Lack of mutually exclusive access to attributes
  - Priority inversion
  - Incorrect priorities and message types
- Inadequate performance and missed deadlines

3.5.3) 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

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

More Background on Inheritance

Concrete classes can be used to instantiate objects.

Abstract classes are incomplete and are used as foundations on which to build subclasses.

Deferred classes are abstract superclasses that declare deferred resources that are to be supplied by their subclasses and which constrain the structure of their subclasses.

When inheritance hierarchies are large or when different variants of the same resource may be multiply inherited from two different superclasses, it is often difficult to determine what is being inherited and from where it is being inherited.

Integration Testing for Common Errors Associated with Classification and Inheritance

Superclass not replaceable by subclass.

Abstract or deferred class instantiated.

Deferred resource not provided by subclass of deferred superclass.

Wrong resource (e.g., attribute type, attribute, operation) inherited (e.g., due to confusion resulting from multiple inheritance).

Original resource in superclass not overwritten in subclass.

Original resource in superclass not deleted in subclass.

Generic classes improperly instantiated (e.g., with incorrect parameters).

Taxonomy not captured by inheritance.

Regression Testing for Common Errors Associated with Classification and Inheritance

Changing a superclass may invalidate subclasses.

Adding subclasses may invalidate inherited resources.

Attributes are local **common global data**.

4) Conclusion

Object-oriented software:

- Has a **radically different structure**
- Is developed according to a **radically different development cycle**

These changes significantly impact software testing

Object-Oriented Development (OOD) methods must address testing.

Developers must be adequately trained in Object-Oriented Testing (OOT).