Testing Object-Oriented Programs

Dr. Durga Prasad Mohapatra

Professor

Department of Computer Science and Engineering

National Institute of Technology, Rourkela, India

Plan of the Talk

- Introduction
- Challenges in testing OO programs
- Test suite design using UML models
- Test design patterns
- Conclusion

Introduction

- More than 50% of development effort is being spent on testing.
- Quality and effective reuse of software depend to a large extent:
  - on thorough testing.
- It was expected during initial years that OO would help substantially reduce testing effort as object-orientation incorporates several good programming features:
  - But, as we find it out today --- it only complicates testing.

Complicacy in testing

- Soon it was realized that satisfactory testing object oriented programs is much more difficult.
- Requires much more cost and effort in comparison to procedural programs.
  - as the various object-oriented features introduce additional complications and scope of new type of bugs.
- Additional test cases are needed to be designed to detect the bugs.

Today's Focus

- Most reported research on OO paradigms focus on:
  - Analysis and design.
- We discuss some important issues in testing object-oriented systems.

Fault Model

- Different types of faults in a program:
  - Infinite for practical purposes .
- A fault model:
  - A map of possible types of faults.
  - Necessary to guide any rational testing strategy.
- A fault model may be constructed from analysis of failure data.

Places To Look For Faults

- Some places (error types) can trivially be omitted from a fault model:
  - I Cannot make English grammar mistakes when I am writing a C program.
  - Cannot commit errors due to side effects (change global variables) while writing a Java program.
- A **test strategy**:
  - Yields a test suite when applied to a unit under test.

Challenges in OO Testing

- What is an appropriate unit for testing?
- Implications of OO features:
  - Encapsulation
  - Inheritance
  - Polymorphism & Dynamic Binding, etc.
- State-based testing
- Test coverage analysis
- Integration strategies
- Test process strategy

What is a Suitable Unit for Testing?

- What is the fundamental unit of testing conventional programs?
  - A function.

- However, as far as OO programs are concerned:
  - Methods are not the basic unit of testing.

Weyukar's Anticomposition axiom

- Any amount of testing of individual methods can not ensure that a class has been satisfactorily tested.

Weyukar's Anticomposition axiom

- The main justification for anticomposition axiom is that
  - a method operates in the scope of the data and other methods of its object.
- So, it is necessary to test a method in the context of these.
- As objects can have significant states, the behavior of a method can be different based on the state of corresponding object.

Basic unit of testing

- Therefore a method has to be tested with all other methods and data of the corr. object.
- Moreover, a method needs to be tested at all states that the object can assume.
- So, it is improper to consider a method as the basic unit of testing an OOP.
- An object is the basic unit of testing of object-oriented programs.
- In object oriented testing, unit testing is conducted by testing each object in isolation.

Suitable Unit for Testing OO Programs

- **Class level:**
  - Testing interactions between attributes and methods must be addressed.
  - State of the object must be considered.

Suitable Unit of Testing

cont…

- **Cluster level:**
  - Tests the interactions among a group of cooperating classes.
- A sequence of interactions is typically required to implement a visible behavior (i.e. a use case).
- **System level:**
  - Tests an entire operational system.

Encapsulation

- Encapsulation is not a source of errors:
  - However, an obstacle to testing.
  - It prevents accessing attribute values by a debugger.
- While testing:
  - Precise current state information is necessary.

Solving Encapsulation-Related Problems

Several solutions are possible:

- Built-in or inherited state reporting methods.
- Low level probes to manually inspect object attributes.
- Proof-of-correctness technique (formal).

Solving Encapsulation-Related Problems

- Most feasible way: State reporting methods.
- Reliable verification of state reporting methods is a problem.

Inheritance

- Should inherited methods be retested?
  - Retesting of inherited methods is the rule, rather than an exception.
- Retesting required:
  - Because a new context of usage results when a subclass is derived.
- Correct behavior at an upper level:
  - Does not guarantee correct behavior at a lower level.

Inheritance --- Overriding

- In case of method overriding:
  - Need to retest the classes in the context of overriding.
  - An overridden method must be retested even when only minor syntactic changes are made.

Deep Inheritance Hierarchy

- A subclass at the bottom of a deep hierarchy:
  - may have only one or two lines of code,
  - but may inherit hundreds of features.
- This situation creates fault hazards:
  - similar to unrestricted access to global data in procedural programs.
- **Inheritance weakens encapsulation.**

Deep Inheritance Hierarchy cont...

- A deep and wide inheritance hierarchy can defy comprehension:
  - Lead to bugs and reduce testability.
  - Incorrect initialization and forgotten methods can result.
  - Class flattening may increase understandibility.
- Multiple Inheritance:
  - increases number of contexts to test.

Abstract and Generic Classes

- Unique to OO programming:
  - Provide important support for reuse.
- Must be extended and instantiated to be tested.
- May never be considered fully tested:
  - Since need retesting when new subclasses are created.

Polymorphism

- Each possible binding of a polymorphic component requires separate testing:
  - Often difficult to find all bindings that may occur.
  - Increases the chances of bugs .
  - An obstacle to reaching coverage goals.

Polymorphism

cont…

- Polymorphism complicates integration planning:
  - many server classes may need to be integrated before a client class can be tested.

Dynamic Binding

- Dynamic binding implies:
  - The code that implements a given function is unknown until run time.
  - Static analysis cannot be used to identify the precise dependencies in a program.
- It becomes difficult to identify all possible bindings and test them.

Object states

- Objects store data parmently and also they have significant states.
- The behavior of a object is usually different in different states.
- Hence the object has to be tested at all its possible states.

Object states

- All state transition functions also needs to be tested thoroughly.

- There should be no extra transitions or extra states other than those defined in the state model.

Why are traditional testing techniques considered unsatisfactory for testing OOPs?

- In procedural programs, procedures are the basic units where in OOPs objects are the basic units of testing.
- Statement coverage based testing is not meaningful for testing OOPs as the inherited methods have to be retested in the derived class.
- In fact, the different O-O features require different test cases to be designed compared to traditional testing.

Why are traditional testing techniques considered unsatisfactory for testing OOPs?

- These O-O features are explicit in design models, and it is usually difficult to extract from an analysis of the source code.
- Hence test cases are designed based on the design model.
- So, this approach is considered to be intermediate between white box and black box approach & is called grey box approach, which is important for testing OOPs.

Test Process Strategy

- Object-oriented development tends toward:
  - shorter incremental cycles.
- Development characterized as:
  - design a little, code a little, test a little.
- The distance between OO specification and implementation, therefore:
  - typically small compared to conventional systems.

Test Process Strategy

cont...

- The gap between white-box/black-box test is diminished.
  - Therefore lower importance of code-based testing.
  - Model-based testing has assumed importance (also called grey box testing).
- Conventional white-box testing can be used for testing individual methods.

Testing Based on UML Models

- UML has become the *de facto* standard for OO modeling and design:
- Though UML is intended to produce rigorous models:
  - Does incorporate many flexibilities.
  - incomplete, inconsistent, and ambiguous models often result.
- Never the less:
  - UML models are an important source of information for test design.

Test Approaches for UML-Based Designs

- Two Approaches:
  1. Interpret the generic test strategy to develop the test suite.
  2. Apply the related test design pattern.

Development of test suite by interpreting the generic test strategy

Use Case-based Testing

- Use cases roughly capture system level requirements.
- A collection of use cases defines complete functionality of the system.
- Each use case (UC) consists of a mainline scenario (sequence) and several alternate scenarios (sequences).
- For each UC, the mainline & all alternate scenarios (sequences) are tested to check, if any errors show up.

Use Case-based Testing

cont...

- Several general kinds of tests can be derived from use cases:
- **Scenario Coverage:**

1. Test cases for basic courses-- "the expected flow of events" mainline sequence.

2. Test cases of other courses-- "all other flows of events" alternate sequences.

- Also, test cases for the features described in user documentation, traceable to each use case, can be generated from use cases.

Use Case Diagram

cont...

- Generic test requirements include:
  - At least one test case should exercise:
    - Every Use Case when actor communicates with Use Case.
    - Every **extension** combination such as Use Case 1 extends Use Case 2.
    - Every **uses** combination such as Use Case 1 uses Use Case 2.

Class Diagram-based Testing

- Class diagram documents the structure of a system. It represents the entities and their inter-relationships.
- Testing derived classes: All derived classes have to be instantiated & tested. In addition to the new methods defined in the derived class, the inherited methods must be retested.
- The test cases should also exercise:
  - Each association relation (association testing)
  - Independent creation & destruction of the objects (container & components) in an aggregation relation (aggregation testing).

Testing Relations Among Classes

- Relations among classes are:
  - Inheritance, Association, Composition, and Dependency.
- Each relationship has corresponding generic test requirements
  - For example, a generalization relation is:
    - not Reflexive(R) and not Symmetric(S) but is Transitive(T)

Testing Relations

cont...

- For any relation, we can ask
  - which of the three (R, S, T) properties is required, excluded, or irrelevant.
- The answers leads to a simple but effective test suite.

Testing Relations

cont...

- For example, when reflexivity is excluded,
  - Any x that would make xRx true should be rejected.
- **Example:** Student *is member of* Dept-Library and Dept-Library is a part of Dept.
  - The relation *is member of* is not R, not S but is T.

State model-based testing

- State charts can be used to represents state-based behavior of :
  - a class, subsystem, or system.
- Statechart model:
  - provides most of the information necessary for class-level state-based testing.

State model-based testing

- The concept of control flow of a conventional program :
  - Does not map readily to an OO program.
- In a state model:
  - We specify how the object's state would change under certain conditions.

State model-based testing

- Flow of control in OO programs:
  - Message passing from one object to another.
  - Causes the receiving object to perform some operation, can lead to an alteration of its state.
- State Coverage: Each method of an object is tested at each state of the object.

State model-based testing

- The state model defines the allowable transitions at each state.
- States can be constructed:
  - Using equivalence classes defined on the instance variables.
- Jacobson's OOSE advocates:
  - Design test cases to cover all state transitions.

State model-based testing

- State transitions coverage: It is tested whether all transitions depicted in the state model work satisfactorily.
- State transition path coverage: All transition paths in the state model are tested.

State model-based testing

- Test cases can be derived from the state machine model of a class:
  - Methods result in state transitions.
  - Test cases are designed to exercise each transition at a state.
- However, the transitions are tied to user-selectable activation sequences:
  - Use Cases

Difficulty with state model-based testing

- The locus of state control is distributed over an entire OO application.
  - Cooperative control makes it difficult to achieve system state and transition coverage.
- A global state model becomes too complex for practical systems.
  - Rarely constructed by developers.
  - A global state model is needed to show how classes interact.

Activity diagram-based testing

- Activity diagrams are based on:
  - flow charts, state transition diagrams, flow graphs and Petri nets.
- Can be considered to be a flow chart that can represent two or more threads of concurrent execution.
- Supports all features of a basic flow graph,
  - Can be used to develop test models for control flow based techniques.

Activity diagram-based testing

- Activity diagram can be used to construct:
  - Decision tables
  - Composite control flow graph representing a collection of sequence diagrams.
  - Control flow graph at method scope:
    - this may be useful to analyze path coverage.

Activity diagram-based testing

- Generic test requirements:
  - At least one test case should exercise each different path:
    - Action 1 is followed by Action 2
    - Synch Point is followed by Action
    - Action is started after Signal
    - Synch Point is reached after Action

Sequence diagram-based Testing

- A sequence diagram (SD) documents message exchanges in time dimension.
- Generic tests include:
  - Message coverage: At least one test case should exercise each message.
  - Message path coverage: All end-to-end message paths in the SD should be identified and exercised.

Difficulties with Sequence Diagram-based testing

- Representing complex control is difficult:
  - For example: notation for selection and iteration is clumsy.
  - It becomes difficult to determine whether all scenarios are covered.
- Dynamic binding and the consequent superclass/subclass behavior :
  - Cannot be represented directly.

Difficulties with Sequence Diagram-based testing cont...

- A Sequence diagram shows only a single collaboration.
- Different bindings are shown on separate sheets.

Collaboration diagram-based testing

- Collaboration diagram represents interactions among objects.
- A collaboration diagram specifies:
  - The implementation of a use case.
- May also depict the participants in design patterns.

Collaboration diagram-based testing

- Each collaboration diagram represents only one slice of the interaction.
- A composite collaboration diagram would be necessary:
  - To develop a complete test suite for an implementation.
- Generic tests include:
  - Message coverage: At least one test case should exercise each message.
  - Other coverage like Condition Coverage

Component diagram-based testing

- Component diagram shows the dependency relationships among:
  - components,
  - physical containment of components,
  - interfaces and calling components.
- The basic testing practice with respect to component diagrams:
  - All call paths should be identified and exercised.

Deployment diagram-based testing

- Deployment diagram represents the hardware, software, and network architecture.
  - Useful in integration planning.
- When a component runs on a node,
  - test cases should exercise whether a component can be loaded and run on each designated host node.
- When a node communicates with another node,
  - test cases should exercise open, transmit, and close communication for each remote component.

Test Coverage

- Test coverage analysis:
  - helps determine the "thoroughness" of testing achieved.
- Several coverage analysis criteria for traditional programs have been proposed:
  - What is a coverage criterion?
- Tests that are adequate w.r.t a criterion:
  - Cover all elements of the domain determined by that criterion.

Test Coverage Criterion cont...

- But, what are the elements that appropriately characterize an object-oriented program?
  - Certainly different from procedural programs.
  - For example: Statement coverage is not appropriate due to inheritance and polymorphism.
- Appropriate test coverage criteria are needed.

Test Coverage Criteria Based on UML Models

- Generalization criterion
- Class attribute criterion
- Condition coverage criterion
- Full predicate coverage criterion
- Each message on link criterion
- All message paths criterion
- Collection coverage criterion

Generalization (GN) Criterion

- Given a test set T and a system model SM, T must cause:
  - every specialization defined in a generalization relationship to be created.
- GN criterion fault model:
  - likely to reveal faults that can arise from violation of the substitutability principle.

Substitutability Principle

"An instance of a subclass can be used anywhere an instance of its superclass is expected."

Class Attribute Criterion

- Given a test set T, a system model SM, and a class C, T must cause
  - a set of representative attribute value combinations in each instance of class C must be created.
- The value space of attributes provides an opportunity to develop test criteria.
- The value space of an attribute are usually restricted by OCL constraints.
- **Example:** *age* attribute in Customer class can have a constraint *age* >18

Condition Coverage Criterion

- Each message in a collaboration diagram occurs only under certain conditions.
- Given a test set T and collaboration diagram CD, T must cause:
  - Each condition in each decision to evaluate to both TRUE and FALSE.

Full Predicate Coverage Criterion

- Given a test set T and collaboration diagram CD, T must cause:
  - Each clause in every condition in CD to take the values of TRUE and FALSE.
- A condition may consist of more than one clause connected by Boolean operators (e.g., AND, OR).

Message On Each Link Criterion

- Given a test set T and collaboration diagram CD, T must cause:
  - Message on each link connecting two objects in CD to be exercised at least once.
- This criterion ensures:
  - All possible messages between two objects occur during tests.

All Message Paths Criterion

- Given a test set T and collaboration diagram CD, T must cause:
  - Each possible message path in CD to be taken at least once.
- A message path is a sequence of messages.
- All message paths is a stronger criterion than message on each link criterion.

Collection Coverage Criterion

- Given a test set T and collaboration diagram CD, T must test:
  - each interaction with a collection of objects of various representative sizes at least once.

- **Example:** An object *Staff* can be an aggregation of *Office staff*, *Faculty* and *Technical Staff*.

Development of test suite by applying test design patterns

Test Design Pattern

- A design pattern is a generalized (reusable) solution to a recurring problem.
- It is based on common sense and some good practices.
- It provides a concise description of:
  - common elements,
  - context, and
  - essential requirements for a solution.
- **Test design patterns are to testing what software design patterns are to programming.**

Test Design Pattern

cont...

Patterns developed under different scope of testing:

Scope	**Example Pattern**
Method scope	Polymorphic Message Test
Class Scope	Modal Class Test
Reusable Components	Abstract Class Test
Subsystem	Round-trip Scenario Test
Integration	Collaboration Integration
Application Scope	Extended Use Case Pattern
Regression Test	Retest Within Firewall

Extended Use Case (EUC)Test Pattern

- Difficulties with use case :
  - No variable definitions required
  - No business rules required
  - No input/output constraints
  - Narrative can be ambiguous
- In a test-ready model we need to
  - Define all input/output relationships
  - Define all operational variables

EUC Intent

- Model the input output relationships in a use case using a decision table.
- It then becomes possible to develop test cases:
  - by considering specific inputs for a use case and corresponding expected results.

EUC Context

cont...

- Unless test points are systematically selected based on a use case's implicit constraints and relationships:
  - all fundamental relationships would not be exercised.
- **Example:** *A* Use *Case* "*Pay-Fees*" can be a generalization of "*Pay-by-Credit-Card*" and "*Pay-by-Cash*" Use Cases

EUC Fault Model

- What kind of bugs can EUCs help detect?
  - Domain bugs.
  - Logic bugs.
- Example: Customer registers with a Supermarket:
  - Domain error: Incorrect Pin code
  - Logic Error: Sales registered against an invalid customer code.

EUC Fault Model

cont...

- Generic system-scope faults:
  - Incorrect output.
  - Abnormal termination.
  - Inadequate response time.
  - Omitted capability.
  - Extra capability.

EUC Test Model

- An Extended Use Case consists of the following:
  - A complete inventory of **operational variables.**
  - A complete specification of domain constraints for each operational variable.
  - An operational relation for each use case
  - The relative frequency of each use case (optional).

EUC Test Procedure

- A test suite is developed in 4 Steps
  - 1. Identify the Operational Variables.
  - 2. Define the domains of the operational variables.
  - 3. Develop the Operational Relations.
  - 4. Develop Test cases.

1. Identify Operational Variables

- Operational variables:
  - Factors which vary from one scenario to the next.
  - Determine different system responses.
- Operational variables include:
  - Explicit inputs and outputs.
  - Environmental conditions which result in significantly different behavior.
  - Abstractions of the state of the system.

2. Define the Domains of the Operational Variables

- The domains are developed by defining:
  - The valid and invalid values for each variable.

3. Develop the Operational Relation

- Operational variables:
  - May determine distinct classes of system responses.
- Express them in the form of a decision table:
  - When all the conditions in a row are true, the expected action is to be produced.

4. Develop Test Cases

- Every variant is made true once and false once.
- Oracle:
  - Expected results are typically developed by inspection.

EUC Test Example

Use Case	Actor	Possible Input/Output Combinations
Establish Session	Bank Customer	(1) Wrong PIN entered once, corrected PIN entered. Display menu. (2) PIN ok, customer's bank not online. Display "Try later." (3) PIN ok, customer's accounts are closed. Display "Call your bank." (4) Stolen card inserted, valid PIN entered. Retain card.
Cash Withdrawal	Bank Customer	(1) Requests \$50, account open, balance 1,234.56, \$50 dispensed. (2) Requests \$100, account closed. (3) Requests \$155.39, account open. \$150 dispensed.
Cash Replenishment	ATM Operator with Armed Guard	(1) ATM opened, Cash dispenser is empty, \$15,000 is added. (2) ATM opened, Cash dispenser is full.

Some Use Cases and Scenarios for an Automatic Teller Machine

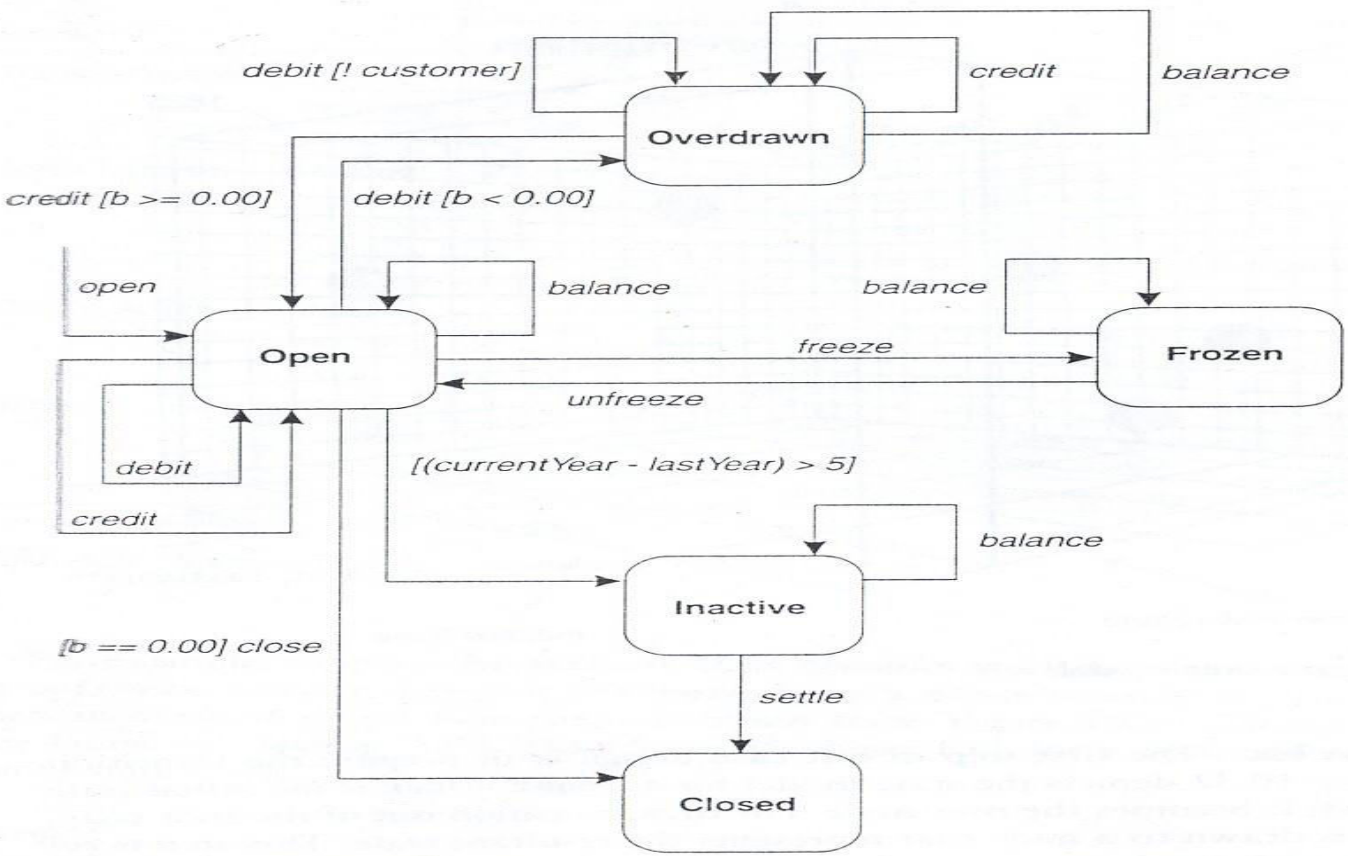
Variant	Operational Variables				Expected Result	
	Card PIN	Entered PIN	Customer Bank Response	Customer Account Status	Message	Card Action
1	Invalid	DC	DC	DC	Insert an ATM card	Eject
2	Valid	Matches Card PIN	Bank Acknowledges	Closed	Contact your bank.	Eject
3	Valid	Matches Card PIN	Bank Acknowledges	Open	Select a transaction	None
4	Valid	Matches Card PIN	Bank Does Not Acknowledge	DC	Please try later	Eject
5	Valid	Doesn't Match	DC	DC	Reenter PIN	None
6	Revoked	DC	Bank Acknowledges	DC	Card invalid	Retain
7	Revoked	DC	Bank Does Not Acknowledge	DC	Card invalid	Eject

ATM Operational Relation for the *Establish Session* Use Case

Modal Class Test (MCT) Pattern

- Some classes can accept any message in any state:
  - But others restrict based on past messages received or current values (domain).
- A modal class is a special class :
  - Places domain constraint as well as sequence constraint while accepting messages.
- **Domain Constraint**: An object of class Account will not accept a withdrawal message if the balance is less than 0.
- **Sequence Constraint**: In the frozen state, an Account object can accept a credit or debit message only after an unfreeze message.

Modal Class Test (MCT) cont...



State transition diagram for class Account.

MCT Fault Model

- A class may fail to implement its state model in five ways:

  1. **Missing transition**: A valid message is rejected in a valid state.
  2. **Incorrect action:** A wrong response for an accepting state and valid message.
  3. **Invalid resultant state**: A message results in transition to a wrong state.
  4. **Invalid state:** An invalid state is produced.
  5. **Sneak path:** A sneak path allows a wrong message to be accepted.

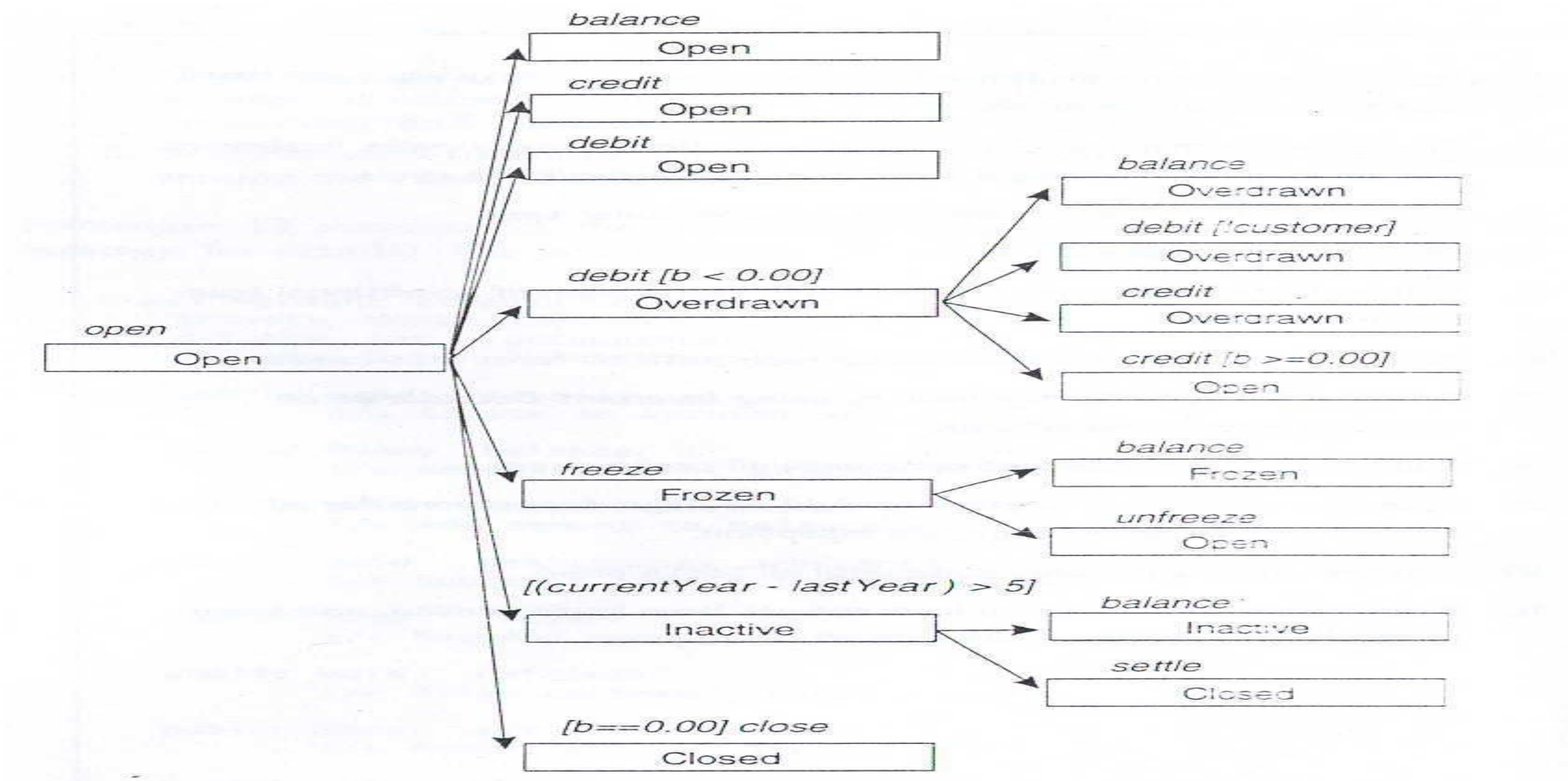
MCT Test Strategy

- The behavior of a class is represented using a state model.
- Generate the transition tree.
- Perform conditional transition test.
- Tabulate events and actions along each path to form test cases.
- Perform illegal transitions and sneak path test.

MCT Transition Tree

- For each transition out of the root state ,
  - A branch is drawn to a node that represents the resultant state.
  - This step is repeated for each resultant state node.
- A node is marked *terminal* :
  - if no transition to a new state (yet to be drawn) exists.
- Each branch in the tree (an event path) becomes a test case.

MCT Transition Tree Testing: Example



MCT Conditional Transition Test

- Each conditional transition:
  - Analyzed to identify additional test cases.
- For each conditional transition :
  - Develop a truth table for the variables in the condition.
- Develop an additional test case:
  - For each entry in the truth table that is not already exercised.

MCT Sneak Path Test

- Sneak Path is a bug that allows an illegal message to be accepted,
  - Resulting in an illegal transition.
- Illegal Transition is present:
  - When a valid state of CUT accepts a message not specified for that state.
- Illegal Message:
  - Results in an illegal transition.

MCT Sneak Path

cont...

- To test sneak paths send illegal messages:
  - **Example**: In the *overdrawn* state, messages: *open, freeze, close* etc can lead to possible sneak paths.
- To confirm that no sneak paths exist:
  - Attempt each type of illegal message.
- The expected response:
  - The message should be rejected.
  - Also, the state of the object should be unchanged after rejecting the illegal message.

Collaboration Integration Pattern(CIP)

- CIP chooses the order of integration:
  - according to collaborations and dependencies.
- Classes to be integrated are tested by testing one collaboration at a time.
- A system consists of many collaborations.
  - Their sequence is determined by dependencies and sequential activation constraints.
- Integration by collaboration:
  - Exercises interfaces between the participants of a collaboration.

CIP Test Procedure

- 1. Develop a dependency tree for the SUT.

  2. Map collaborations onto tree until all components and interfaces are covered.

  3. Choose a sequence in which to apply the collaborations.

  4. Several heuristics are available for choosing a sequence.

CIP Test Procedure cont..

Example Heuristics:

1. Begin with the simplest and finish with the most complex.
2. Begin with the collaboration that requires the fewest stubs .
3. Test in order of risk of disruption of system testing.

CIP Test Procedure

cont...

4. Develop test suite for each collaboration
5. Run the test suite and debug until first collaboration passes
6. Continue until all collaborations have been exercised

Integration Strategies

- Essentially two approaches exist
  - Thread-based
  - Use-based

Thread-based integration Strategies

- A thread consists of:
  - All the classes needed to respond to a single external input.
- Each class is unit tested,
  - then each thread is exercised.

Use Case-based Integration Strategies

- Use case-based integration:
  - Begins by testing classes that use services of none (or a few) of other classes.
  - Next, classes that use the first group of classes are tested.
  - Followed by classes that use the second group, and so on.

Research Challenge 1: Testing Based on Precode Artifacts

- Precode artifacts:
  - Design,
  - Requirements,
  - Architecture specifications.
- Software architecture involves description of elements from which systems are built,
  - Interactions among those elements,
  - Patterns that guide their composition,
  - Constraints on these patterns.

Research Challenge 2:Testing Evolving Software

- Regression testing is used to test software that undergoes evolution due to:
  - Technology changes,
  - New or modified components.
- Regression testing remains one of the most expensive activities performed during a software lifecycle.

Research Challenge 3: Measuring Effectiveness Of Testing Techniques

- Quantitative values of effectiveness of a test-set design in revealing faults:
  - Analytical, statistical, or empirical.
- We also need to understand the classes of faults for which the different criteria are useful.

Research Challenge 4: Test Data Generation

- Generating test data (inputs for test cases) is often a labour-intensive process.
- To date, available techniques generating test data automatically work for toy systems:
  - Do not scale to large systems.
- Also, it should be possible to automatically execute test cases:
  - The test output should also be automatically analyzed.

Research Challenges 5: Methods and Tools

- Along with fundamental research and empirical methods for testing:
  - tool development is also needed.
- An important requirement:
  - Methods and tools be scalable to large systems.
  - Many of the tools and methods developed so far work only on toy systems.

Summary

- Discussed introduction to O-O S/W testing
- Challenges in testing OO programs
- Test suite design using UML models
- Test design patterns
- Current research challenges

References

1. R. Mall, Fundamentals of Software Engineering, Fifth Edition, (Chapter – 10), PHI, 2018.
2. R. S. Pressman, Software Engineering, McGraw-Hill, 2018.
3. N. Chauhan, Software Testing; Principles and Practices, Second Edition, (Chapter – 14), Oxford University Press, 2018.

Thank You