

# Chapter 14

# Testing Tactics

Moonzoo Kim

CS Division of EECS Dept.  
KAIST

[moonzoo@cs.kaist.ac.kr](mailto:moonzoo@cs.kaist.ac.kr)

<http://pswlab.kaist.ac.kr/courses/cs550-07>

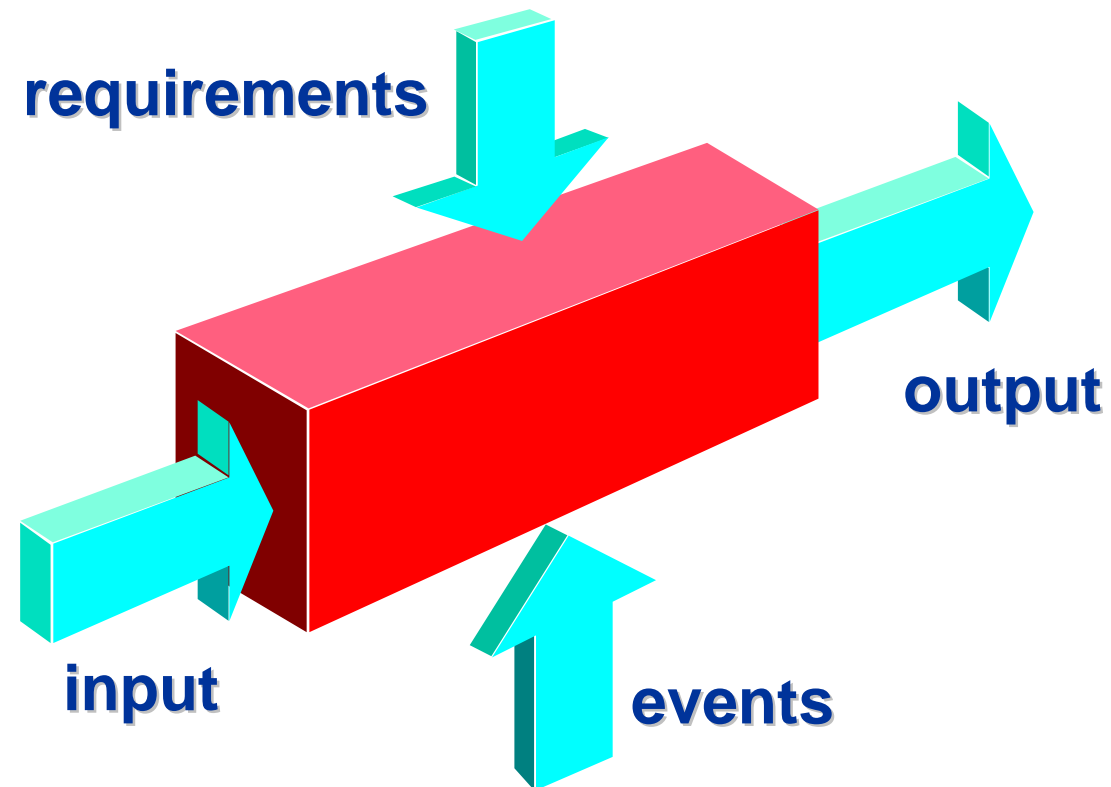
# Overview of Ch14. Testing Tactics

- 14.1 Software Testing Fundamentals
- 14.2 Blackbox and White-Box Testing
- 14.3 White-Box Testing
- 14.4 Basis Path Testing
  - Glow Graph Notation
  - Independent Program Paths
  - Deriving Test Cases
  - Graph Matrices
- 14.5 Control Structure Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing

# Overview of Ch14. Testing Tactics

- 14.6 Blackbox Testing
  - Graph-based testing methods
  - Equivalence partitioning
  - Boundary value analysis
  - Orthogonal array testing
- 14.7 OO Testing Methods
  - The test case design implications of OO concepts
  - Applicability of conventional test case design methods
  - Fault-based testing
  - Test cases and class hierarchy
  - Scenario-based testing
  - Testing surface structure and deep structure
- 14.8 Testing methods applicable at the class level
- 14.9 InterClass Test Case Design
- 14.10 Testing for Specialized Environments, Architectures, and Applications
- 14.11 Testing Patterns

# Black-Box Testing



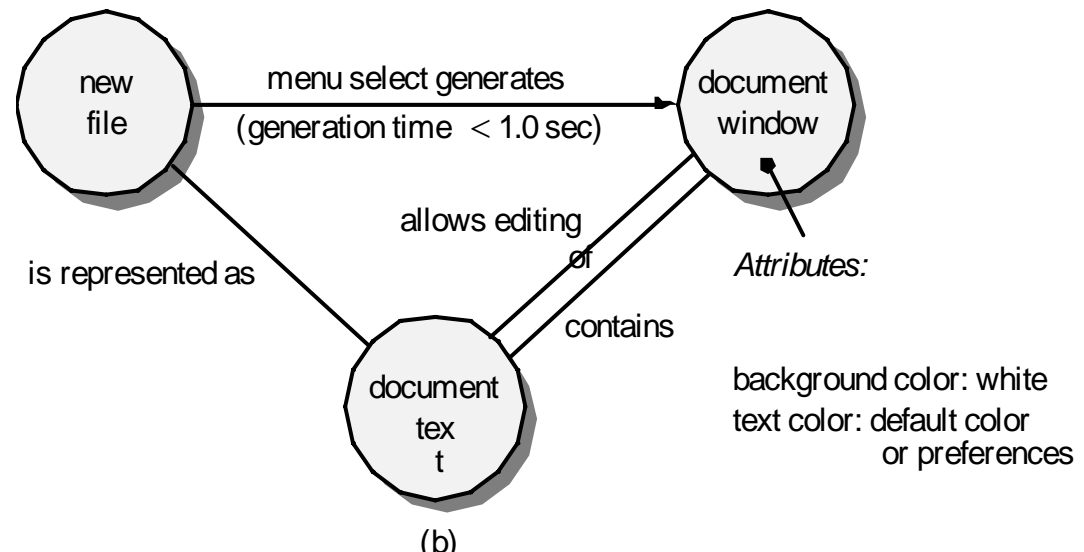
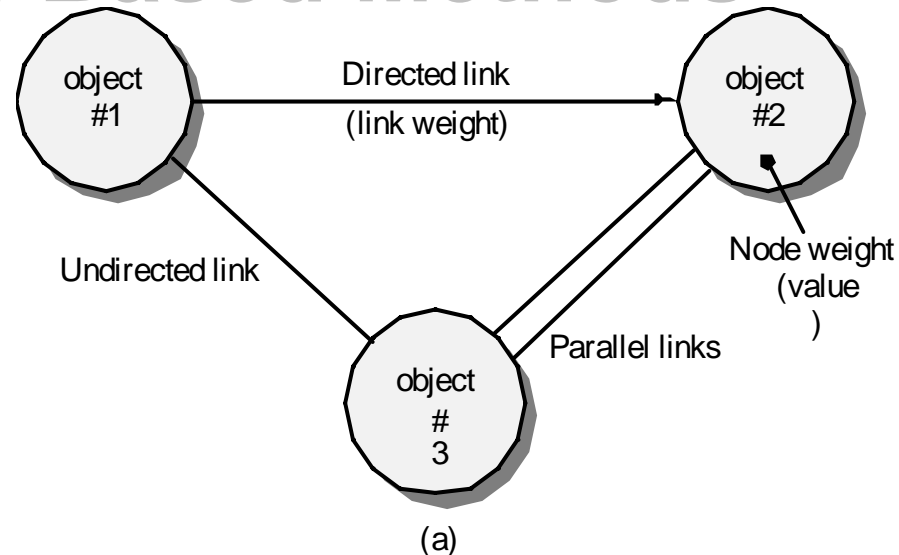
# Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

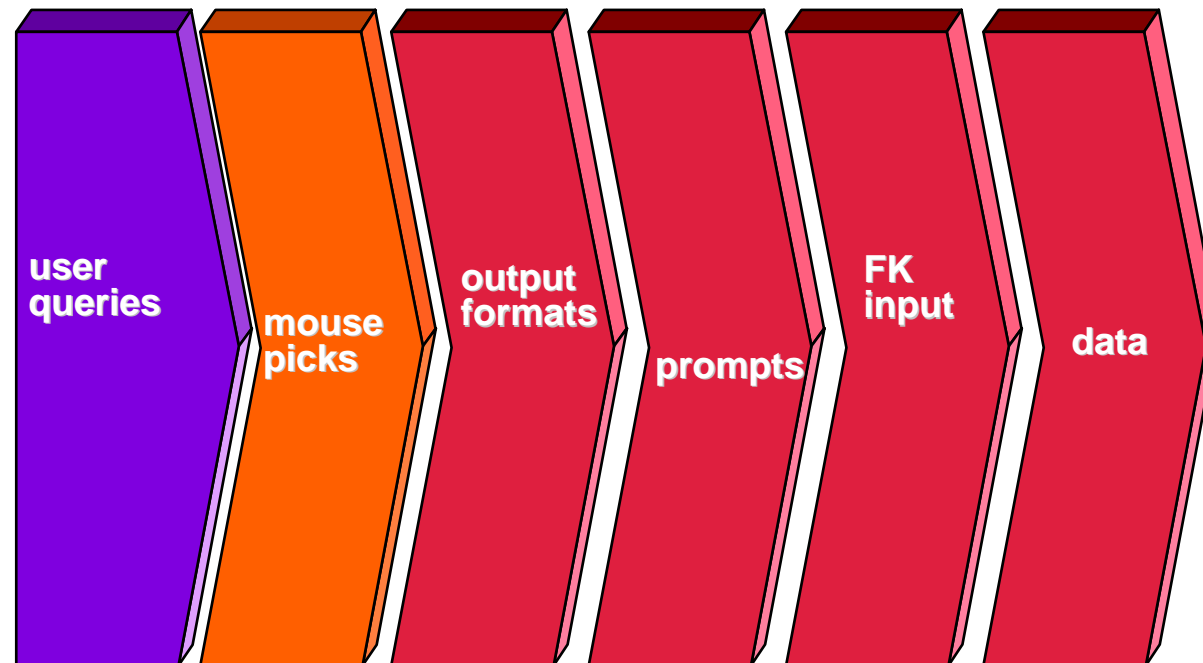
# Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

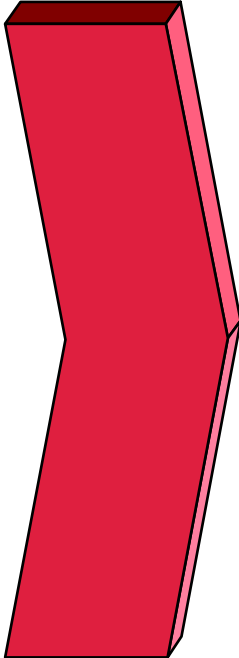


# Equivalence Partitioning



# Sample Equivalence Classes

## Valid data



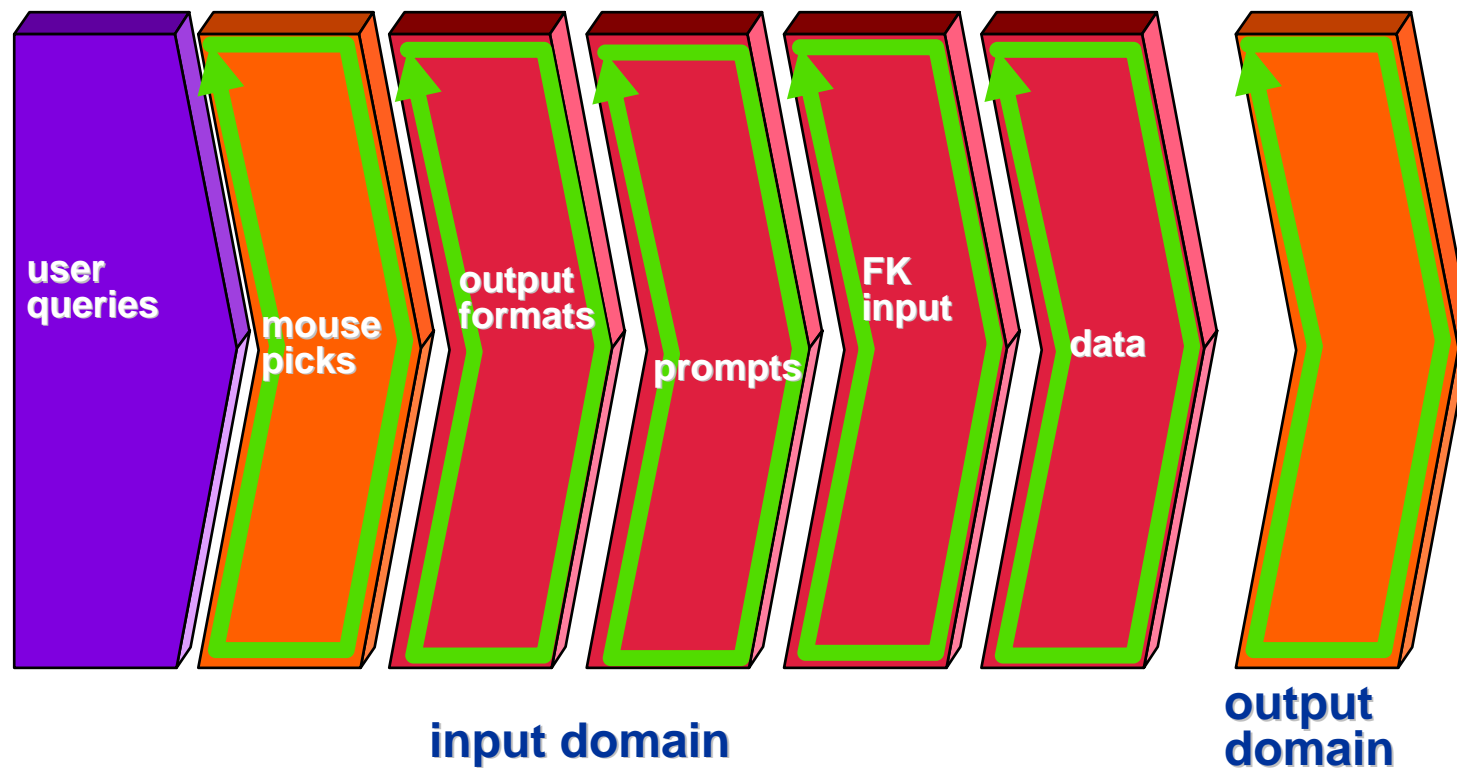
- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

## Invalid data

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place



# Boundary Value Analysis

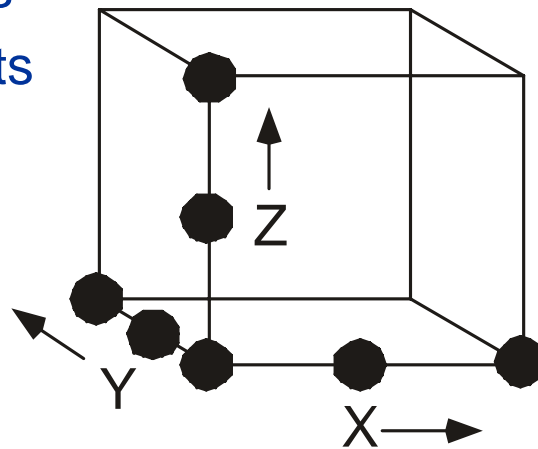


# Comparison Testing

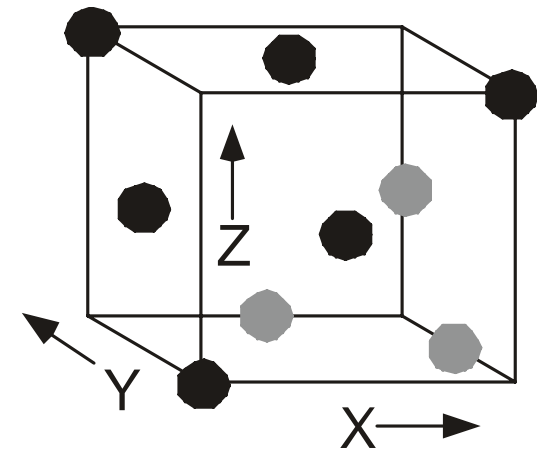
- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

# Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded
  - Single mode faults
  - Double mode faults
  - Multimode faults



One input item at a time



L9 orthogonal array

# Testing Methods

## ■ Fault-based testing

- The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

## ■ Class Testing and the Class Hierarchy

- Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

## ■ Scenario-Based Test Design

- Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

# OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a minimum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

# OOT Methods: Partition Testing

## ■ Partition Testing

- reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
- state-based partitioning
  - categorize and test operations based on their ability to change the state of a class
- attribute-based partitioning
  - categorize and test operations based on the attributes that they use
- category-based partitioning
  - categorize and test operations based on the generic function each performs

# OOT Methods: Inter-Class Testing

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

# OOT Methods: Behavior Testing

The tests to be designed should achieve **all state coverage** [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states

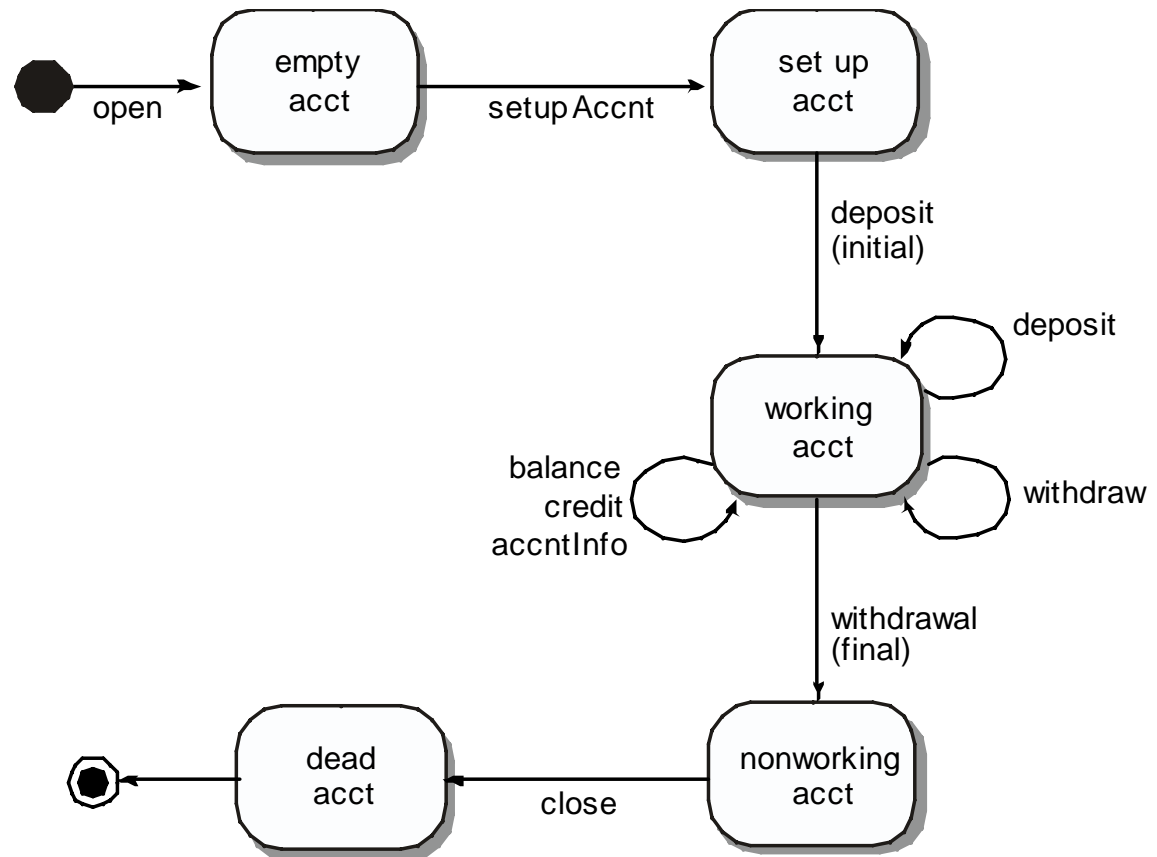


Figure 14.3 State diagram for Account class (adapted from [ KIR94])



# Testing Patterns

## **Pattern name: pair testing**

**Abstract:** A process-oriented pattern, pair testing describes a technique that is analogous to pair programming (Chapter 4) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

## **Pattern name: separate test interface**

**Abstract:** There is a need to test every class in an object-oriented system, including “internal classes” (i.e., classes that do not expose any interface outside of the component that used them). The separate test interface pattern describes how to create “a test interface that can be used to describe specific tests on classes that are visible only internally to a component.” [LAN01]

## **Pattern name: scenario testing**

**Abstract:** Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The scenario testing pattern describes a technique for exercising the software from the user’s point of view. A failure at this level indicates that the software has failed to meet a user visible requirement.

[KAN01]