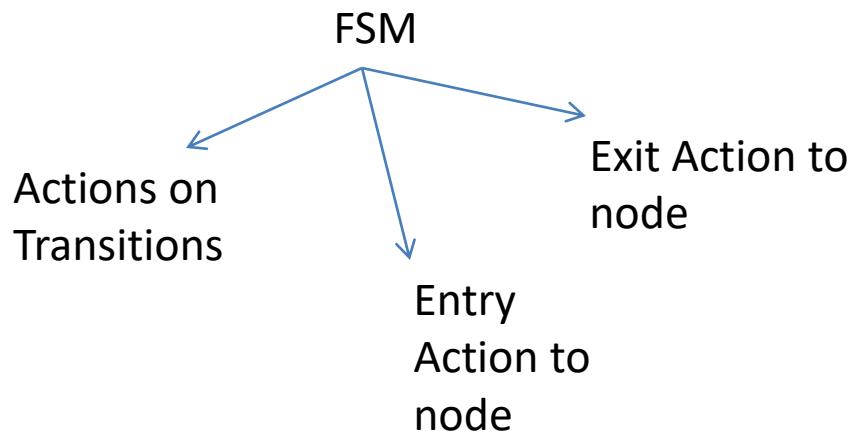


# Finite State Machine

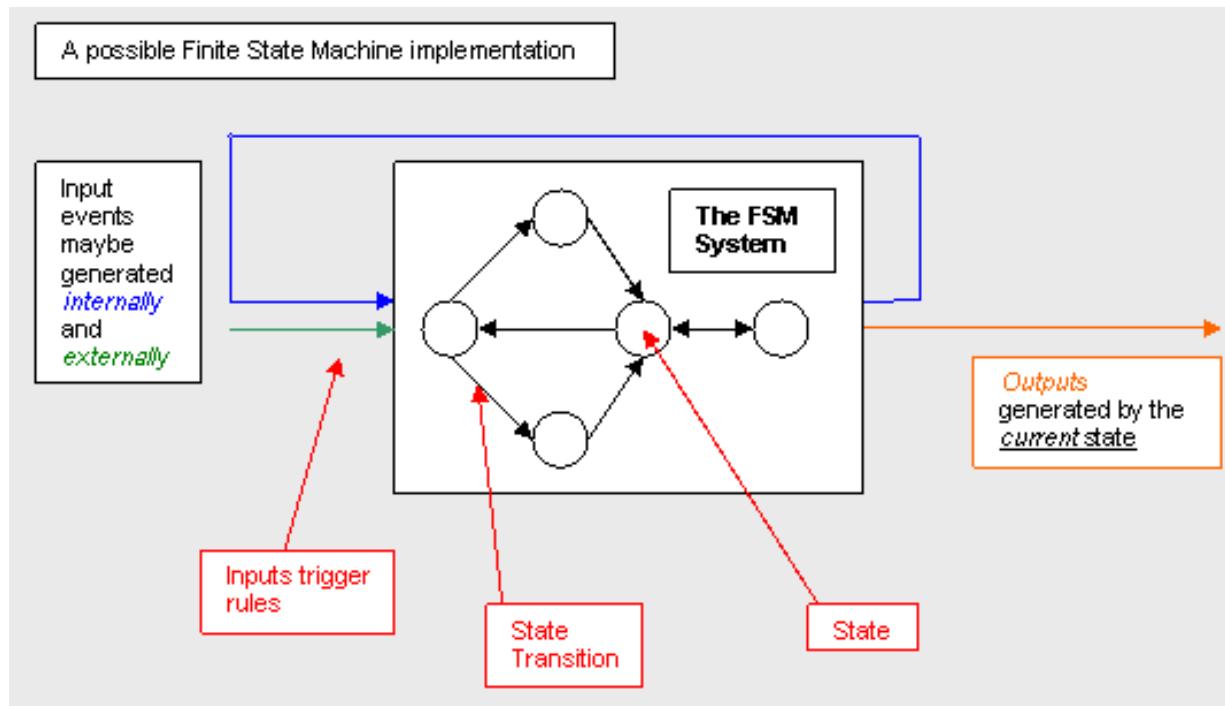
- A **finite-state machine (FSM)** or **finite-state automaton (FSA,** plural: *automata*), **finite automaton**, or simply a **state machine**, is a mathematical model of computation.
  - It is an abstract machine that can be in exactly one of a finite number of states at any given time.
  - The FSM can change from one **state** to another in response to some external inputs; the change from one state to another is called a **transition**. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.
  - Finite state machines are of two types - **deterministic finite state machine** and **non deterministic finite state machine**
  - Examples: Vending machine, Elevator, Traffic lights etc
- FSM is a graph that describes how software variables are modified during execution.

- FSM helps to analyze early detection of errors through analysis of the model such as UML, State tables, Boolean logic etc. Since, these are modeled even before the code is generated.
- FSM is used for testing since 30 years
- It is best suited for control intensive applications such as elevators and not best suited for data control applications like web apps



- Nodes – represent states which in turn indicates the set of values for the key variables
- Edges – represent transitions which will have guards (conditions) and or actions.
- Preconditions (guards) – Conditions to be present for transition to occur
- Triggering events – changes to variables that cause the transitions
- Testers thus have to create test cases where every state has to be visited and hence every path will be checked for i.e for every transition there should be test cases generated
- Thus, testers have to know about FSM since they should be able to draw FSM models with complete state coverage. Also they have to define the data associated with each state

- Initial state - provides a starting point.
- Current state - remembers the product of the last state transition.



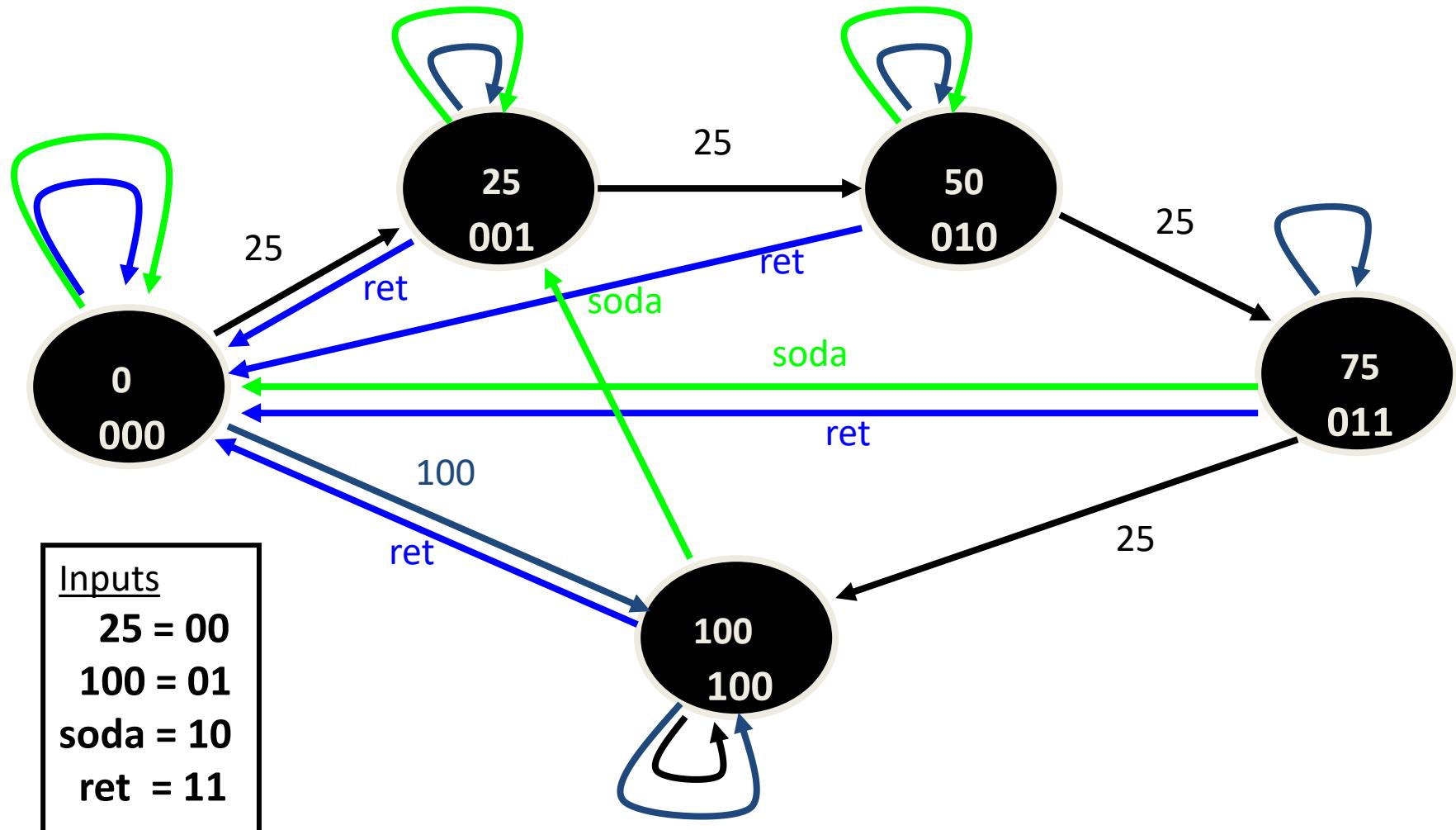
# Example: Vending Machine

- Takes only quarters and dollar bills
- Won't hold more than \$1.00
- Sodas cost \$.75
- Possible actions (inputs)
  - deposit \$.25 (25)
  - deposit \$1.00 (\$)
  - push button to get soda (soda)
  - push button to get money returned (ret)

# Classic Example: Vending Machine

- State: description of the internal settings of the machine, e.g. how much money has been deposited and not spent
- Finite states: 0, 25, 50, 75, 100,
- Rules: determine how inputs can change state

# Example: Vending Machine



# Petri Net Overview

- Petri nets were invented by Carl Petri in 1966 to explore cause and effect relationships
- Expanded to include deterministic time
- Then stochastic time
- Then logic

# Definition

- A Petri Nets (PN) comprises places, transitions, and arcs
  - Places are system states
  - Transitions describe events that may modify the system state
  - Arcs specify the relationship between places
- Tokens reside in places and are used to specify the state of a PN

# Definition of Petri Net

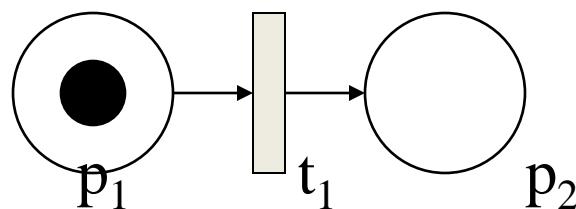
- $C = (P, T, I, O)$ 
  - **Places**  
 $P = \{p_1, p_2, p_3, \dots, p_n\}$
  - Represents an entity or a pathway component
  - **Transitions**  
 $T = \{t_1, t_2, t_3, \dots, t_n\}$
  - Represent a process between places and / or used to model dependencies between places
  - **Input**  
 $I : T \rightarrow P^r$  ( $r = \text{number of places}$ )
  - **Output**  
 $O : T \rightarrow P^q$  ( $q = \text{number of places}$ )
  - Tokens
  - Tokens reside in places, and are used to specify the state of a Petri Net
- marking  $\mu$  : assignment of tokens to the places of Petri net  
 $\mu = \mu_1, \mu_2, \mu_3, \dots, \mu_n$

# Applications of Petri Net

- Petri net is primarily used for studying the dynamic concurrent behavior of network-based systems where there is a discrete flow.
- Petri Nets are applied in practice by industry, academia, and other places.

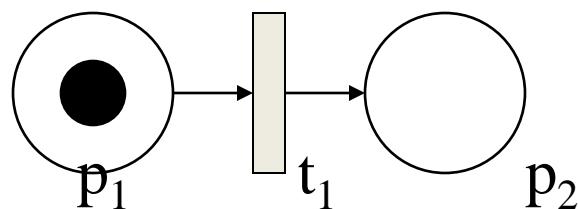
## Basics of Petri Nets

- Petri net consist two types of nodes: *places* and *transitions*. An arc exists only from a place to a transition or from a transition to a place.
- A place may have zero or more *tokens*.
- Graphically, places, transitions, arcs, and tokens are represented respectively by: circles, bars, arrows, and dots.



## Basics of Petri Nets -continued

- Below is an example Petri net with two places and one transaction.
- Transition node is ready to **fire** if and only if there is at least one token at each of its input places



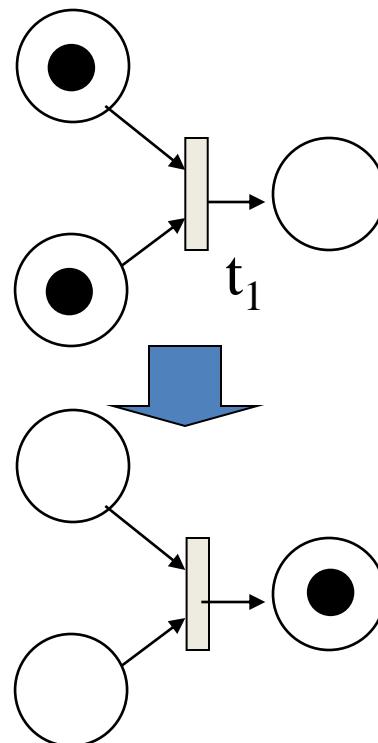
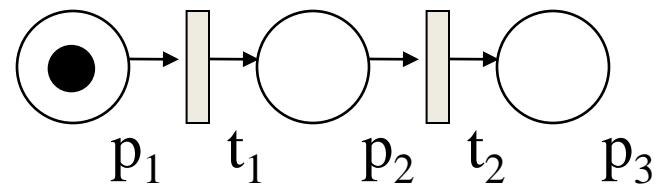
state transition of form  $(1, 0) \rightarrow (0, 1)$

$p_1$  : input place

$p_2$ : output place

# Properties of Petri Nets

- Sequential Execution  
Transition  $t_2$  can fire only after the firing of  $t_1$ . This impose the precedence of constraints " $t_2$  after  $t_1$ ."
- Synchronization  
Transition  $t_1$  will be enabled only when a token there are at least one token at each of its input places.
- Merging  
Happens when tokens from several places arrive for service at the same transition.

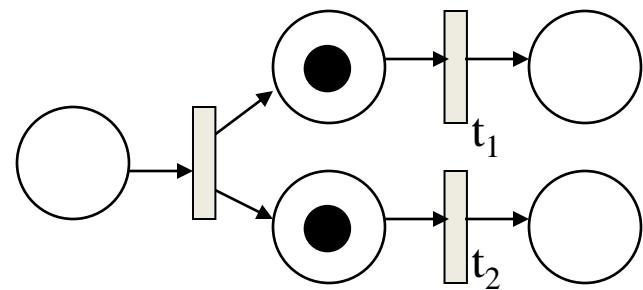


# Properties of Petri Nets -continued

- **Concurrency**

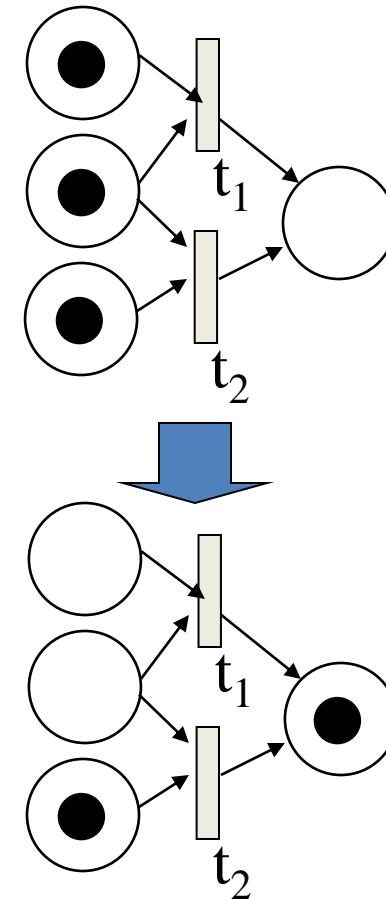
$t_1$  and  $t_2$  are concurrent.

- with this property, Petri net is able to model systems of distributed control with multiple processes executing concurrently in time.



# Properties of Petri Nets -continued

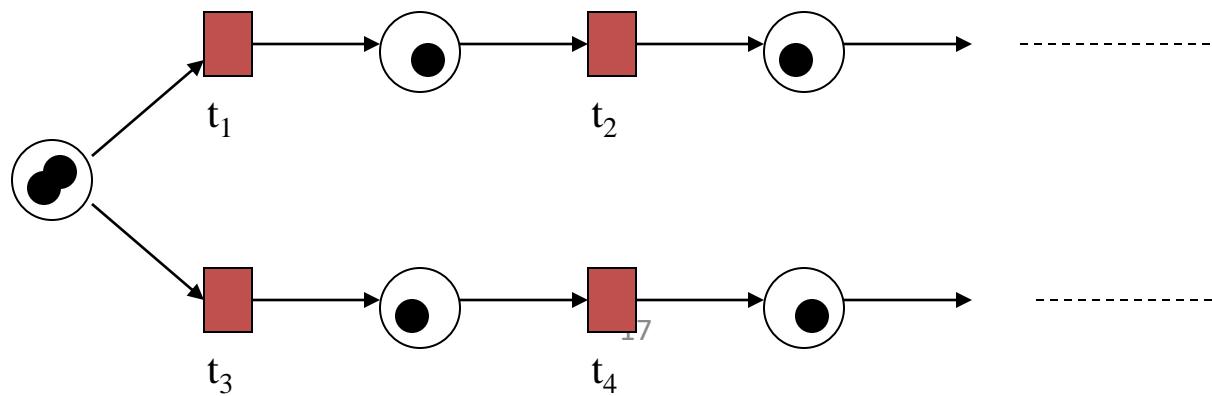
- Conflict  
 $t_1$  and  $t_2$  are both ready to fire but the firing of any leads to the disabling of the other transitions.



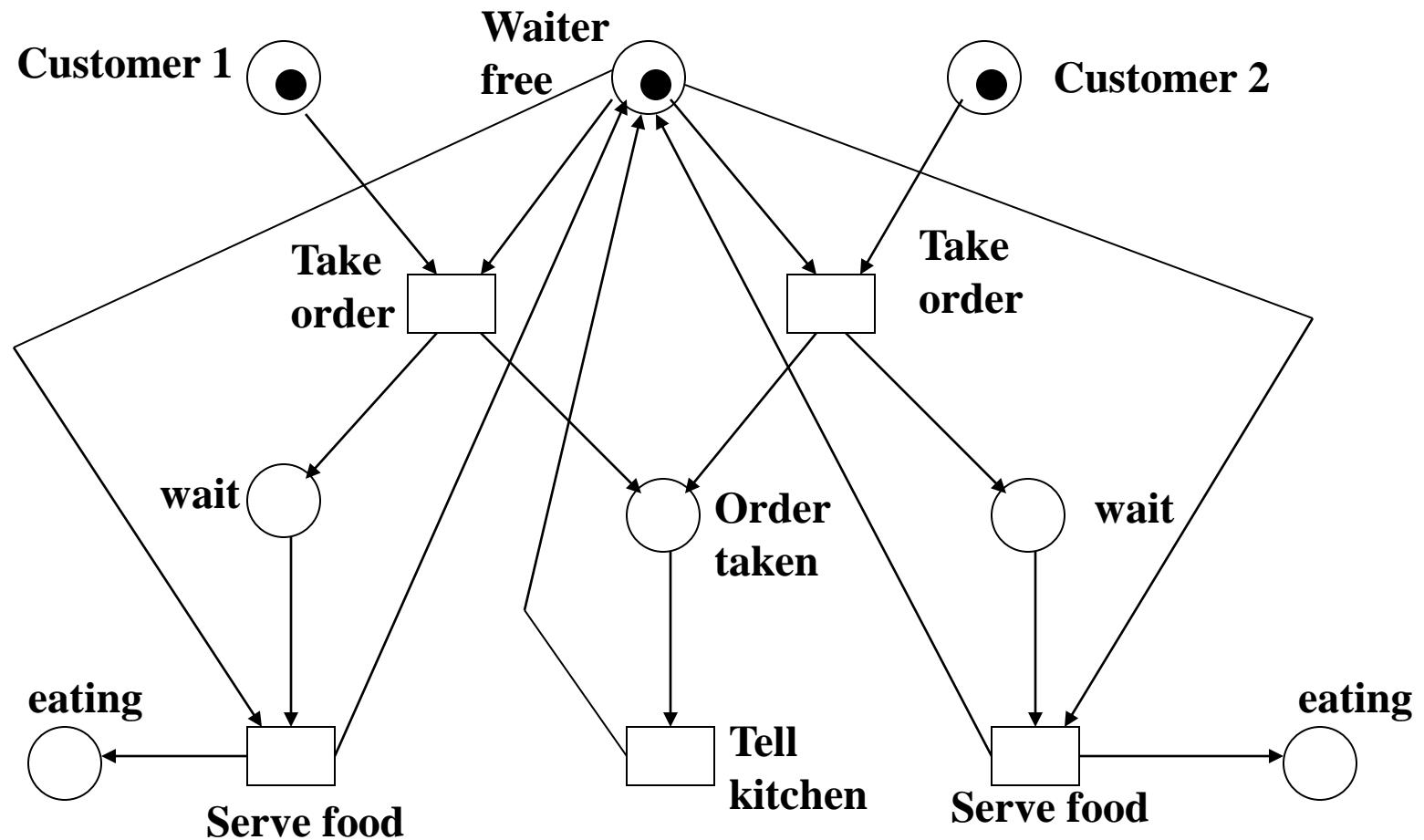
# Properties of Petri Nets -continued

- Conflict - continued
  - the resulting conflict may be resolved in a purely non-deterministic way or in a probabilistic way, by assigning appropriate probabilities to the conflicting transitions.

there is a choice of either  $t_1$  and  $t_2$ , or  $t_3$  and  $t_4$



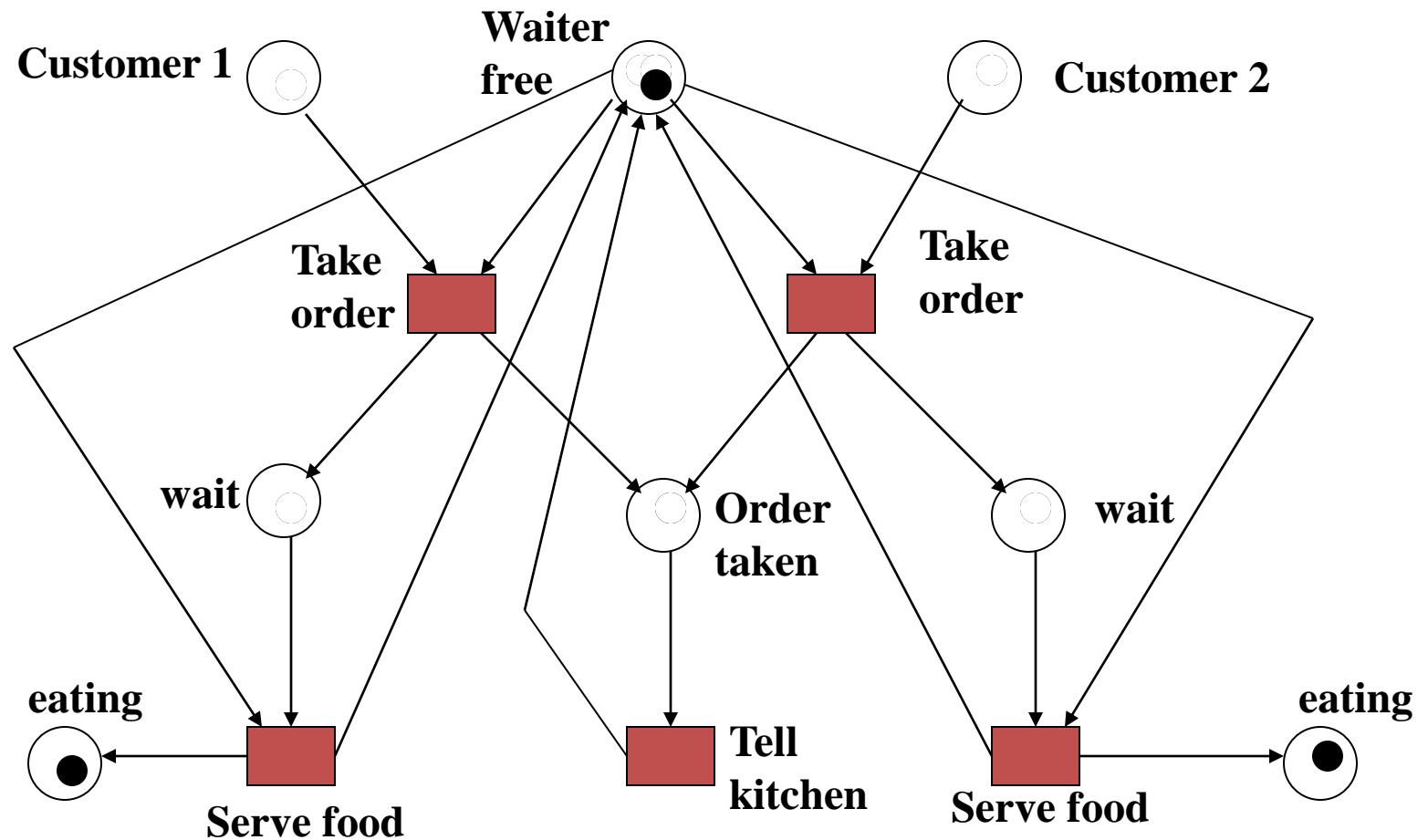
# Example: In a Restaurant (A Petri Net)



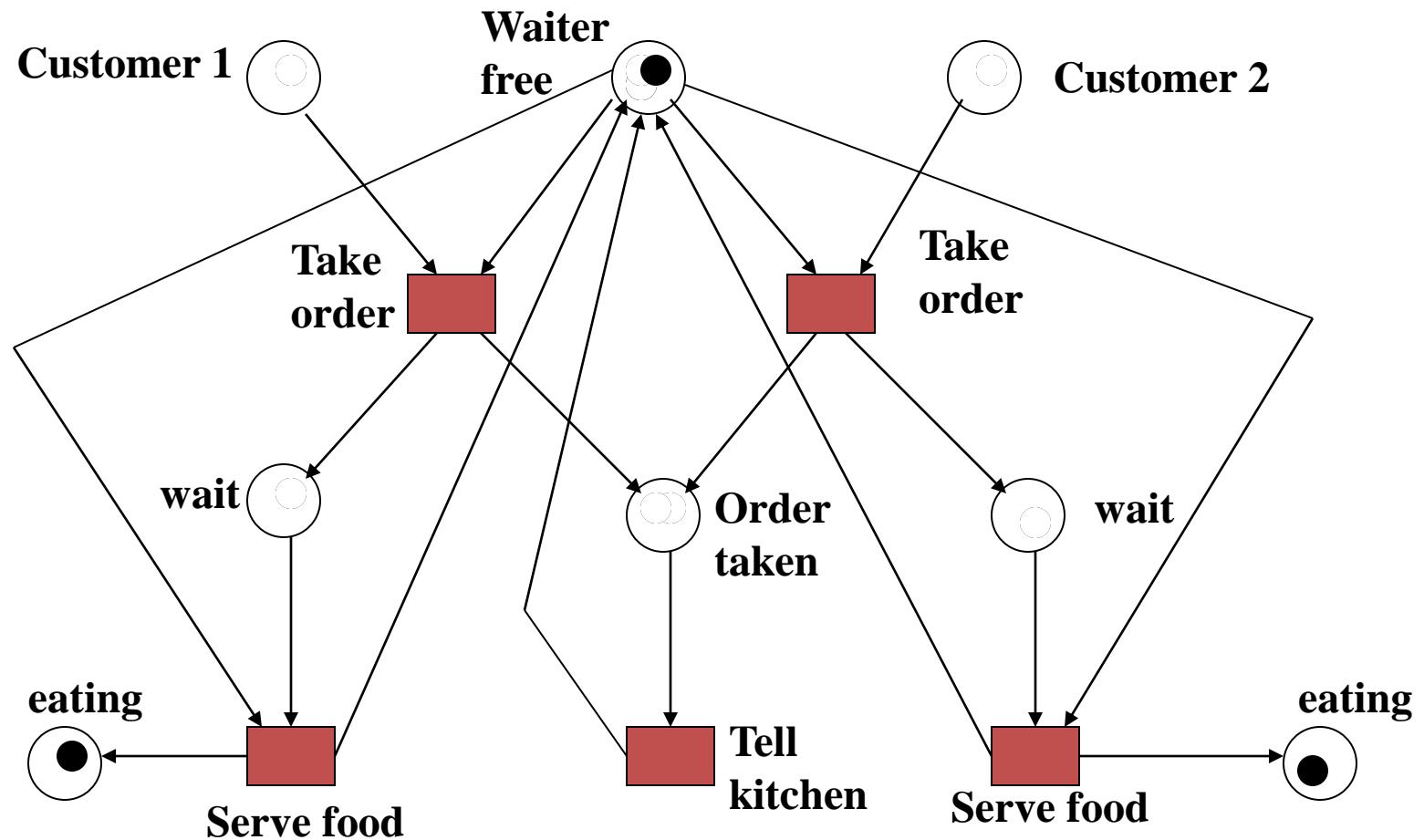
# Example: In a Restaurant (Two Scenarios)

- Scenario 1:
  - Waiter takes order from customer 1; serves customer 1; takes order from customer 2; serves customer 2.
- Scenario 2:
  - Waiter takes order from customer 1; takes order from customer 2; serves customer 2; serves customer 1.

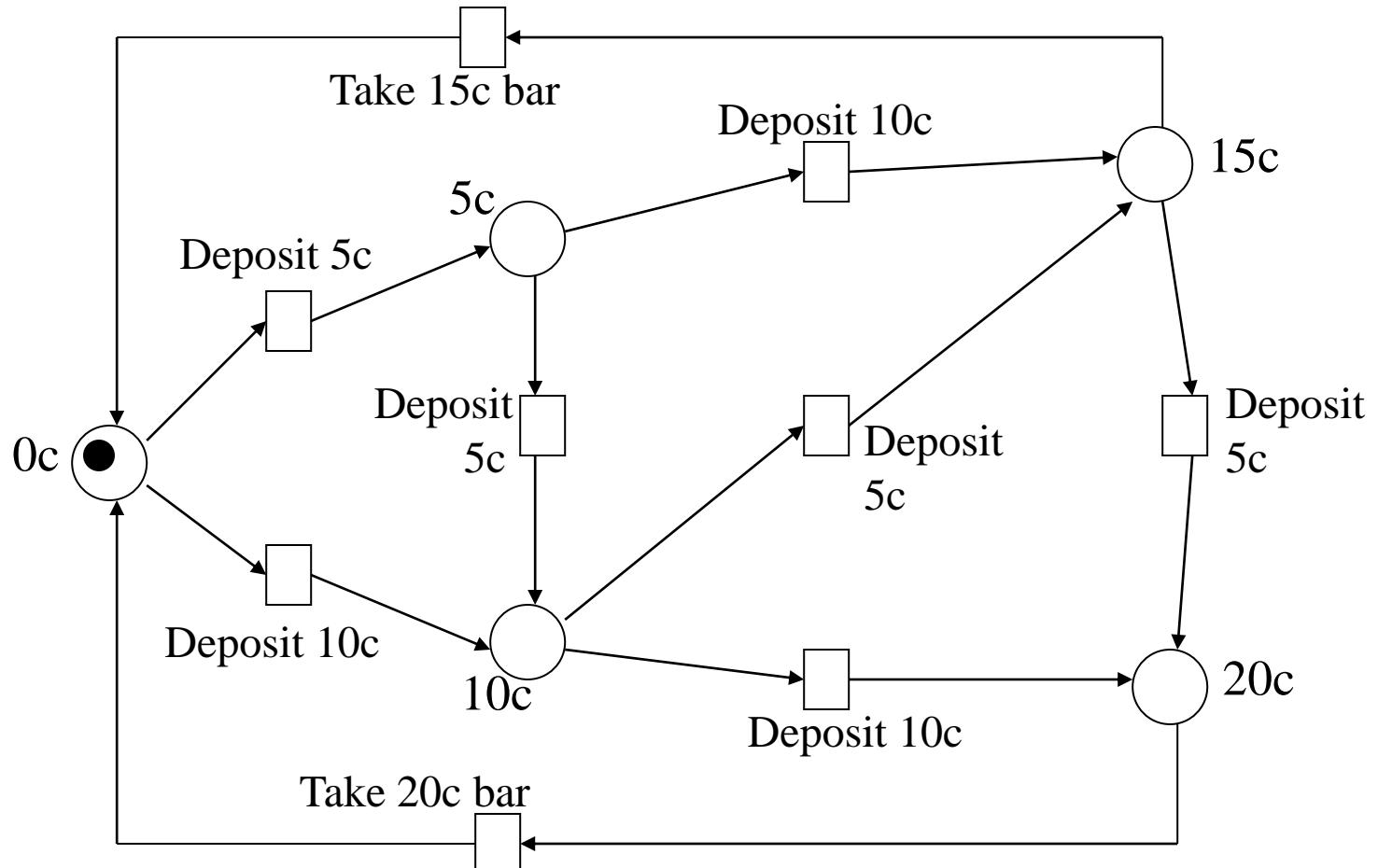
# Example: In a Restaurant (Scenario 1)



# Example: In a Restaurant (Scenario 2)



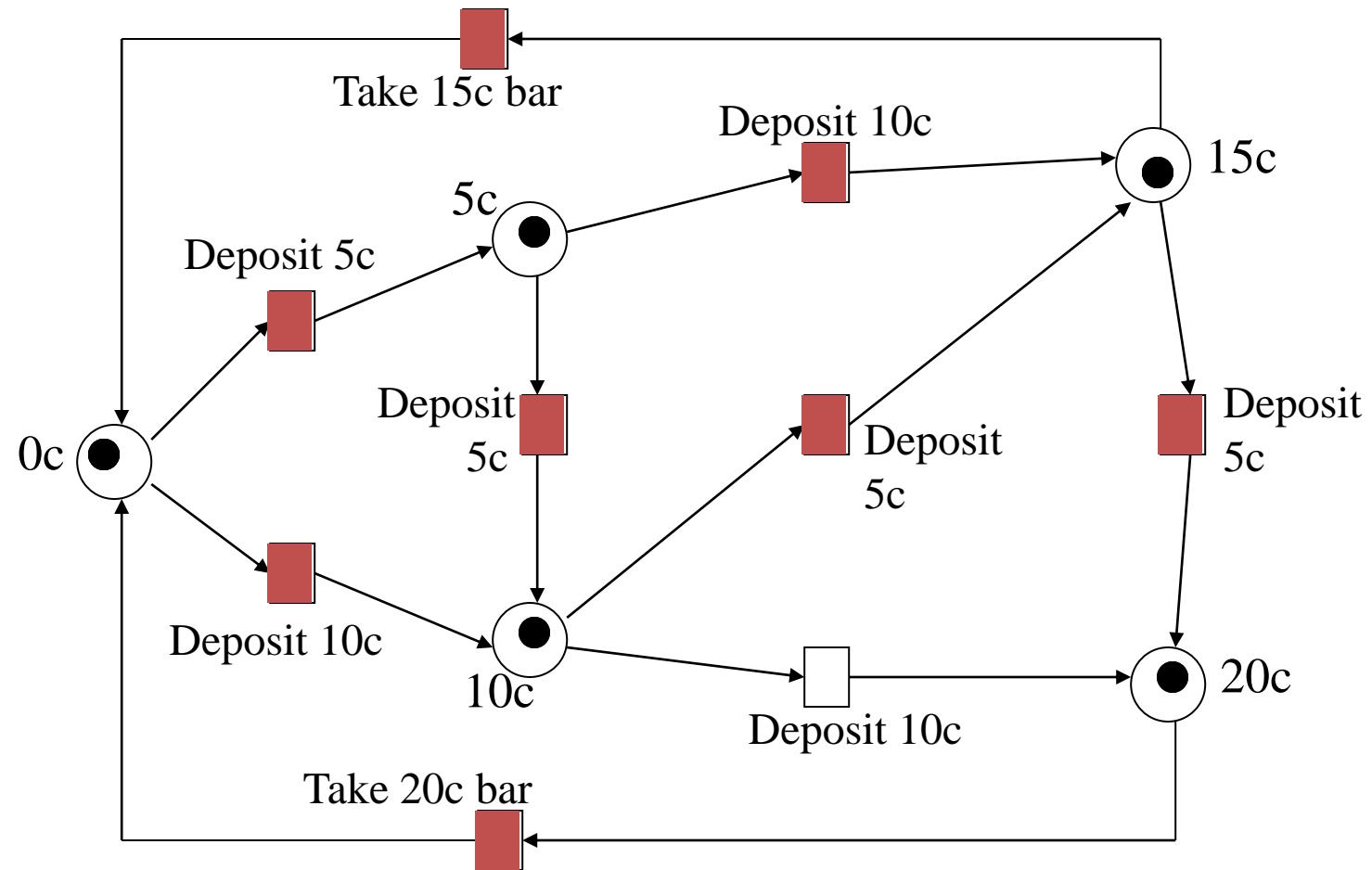
# Example: Vending Machine (A Petri net)

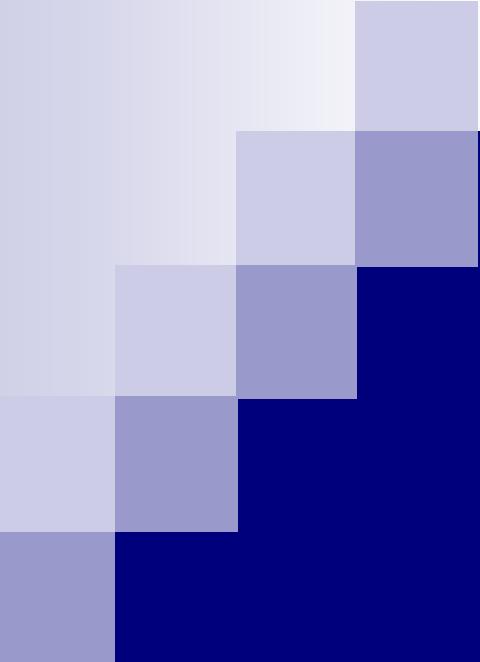


# Example: Vending Machine (3 Scenarios)

- Scenario 1:
  - Deposit 5c, deposit 5c, deposit 5c, deposit 5c, take 20c snack bar.
- Scenario 2:
  - Deposit 10c, deposit 5c, take 15c snack bar.
- Scenario 3:
  - Deposit 5c, deposit 10c, deposit 5c, take 20c snack bar.

# Example: Vending Machine (Token Games)

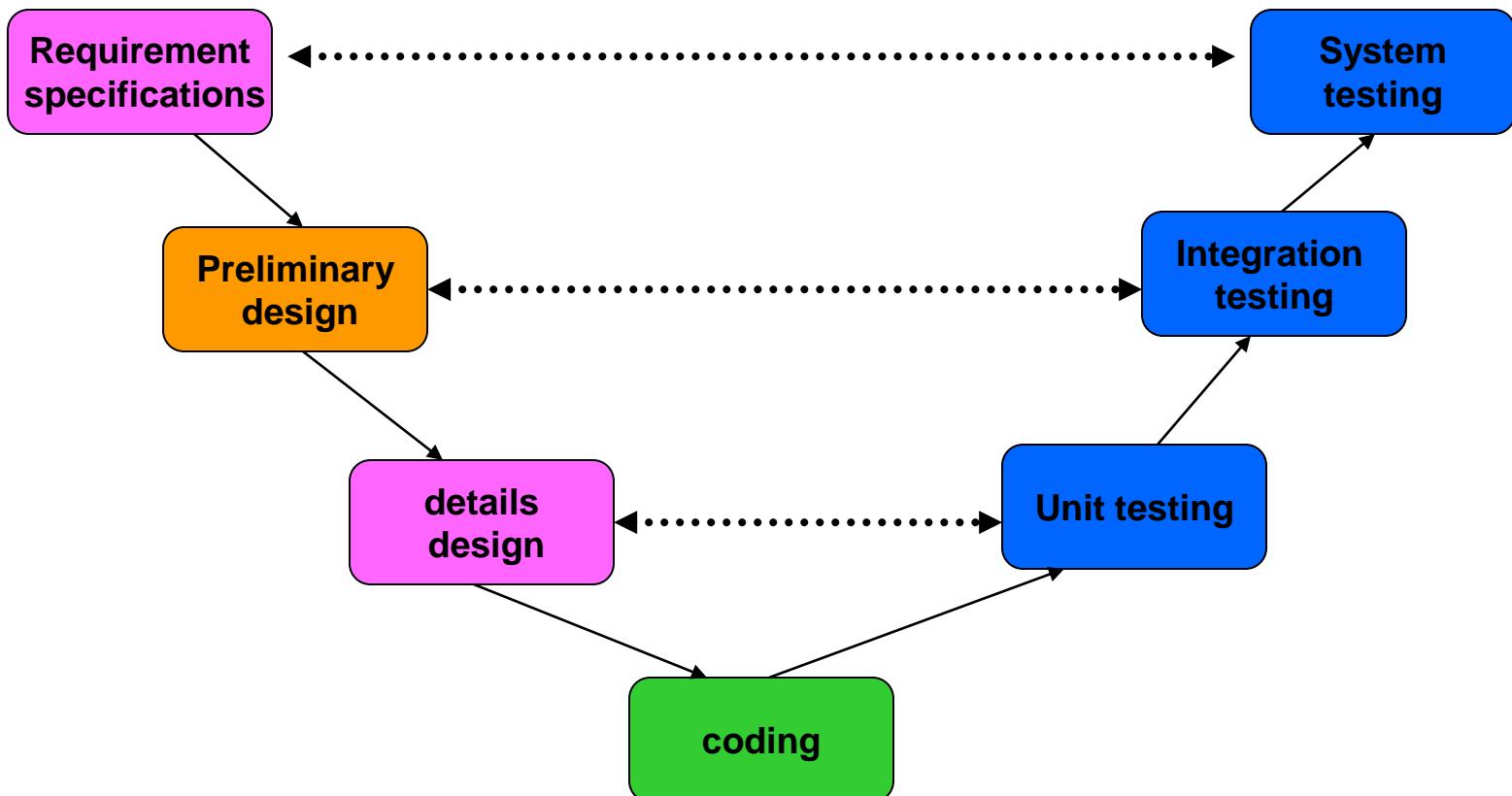


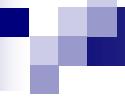


# Integration testing

**Testing Types:** Integration Testing

# The waterfall life cycle





## Goals/Purpose of Integration Testing

- Presumes previously tested units
- Not system testing  
(at level of system inputs and outputs)
- tests functionality "between" unit and system levels
- basis for test case identification?
- Emphasis shifts from “how to test” to “what to test”

# Approaches to Integration Testing

## 1. Based on Functional Decomposition

**Top-Down**  
**Bottom-Up**  
**Sandwich**  
**Big Bang**  
Incremental

## 2. Based on Call Graph

**Pair-wise**  
**Neighborhood**

## 3. Based on Paths

**MM-Paths**  
**Atomic System Functions**

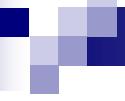
# Steps in Integration-Testing

- 1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
- 2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
- 3. Do *functional testing*: Define test cases that exercise all uses cases with the selected component
- 4. Do *structural testing*: Define test cases that exercise the selected component
- 5. Execute *performance tests*
- 6. Keep records of the test cases and testing activities.
- 7. Repeat steps 1 to 7 until the full system is tested.

The primary goal of *integration testing* is to identify errors in the (current) component configuration.

# Which Integration Strategy should you use?

- Factors to consider
  - Amount of test harness (stubs & drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - ...
- ...Top-level components are usually important and cannot be neglected up to the end of testing
- Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult



# Running example: Simple ATM system

- An ATM:
  - Provides 15 screens for interactions
  - includes 3 function buttons

### Screen 1

Welcome.

Please Insert your  
ATM card for service

### Screen 2

Enter your Personal  
Identification Number

— — — —  
Press Cancel if Error

### Screen 3

Your Personal  
Identification Number  
is incorrect. Please  
try again.

### Screen 4

Invalid identification.  
Your card will be  
retained. Please call  
the bank.

### Screen 5

Select transaction type:  
balance  
deposit  
withdrawal

Press Cancel if Error

### Screen 6

Select account type:

checking  
savings

Press Cancel if Error

### Screen 7

Enter amount.  
Withdrawals must be  
in increments of \$10

Press Cancel if Error

### Screen 8

Insufficient funds.  
Please enter a new  
amount.

Press Cancel if Error

### Screen 9

Machine cannot  
dispense that amount.

Please try again.

### Screen 10

Temporarily unable to  
process withdrawals.  
Another transaction?

yes  
no

### Screen 11

Your balance is being  
updated. Please take  
cash from dispenser.

### Screen 12

Temporarily unable to  
process deposits.  
Another transaction?

yes  
no

### Screen 13

Please put envelope into  
deposit slot. Your  
balance will be updated

Press Cancel if Error.

### Screen 14

Your new balance is  
printed on your receipt.  
Another transaction?

yes  
no

### Screen 15

Please take your  
receipt and ATM  
card. Thank you.

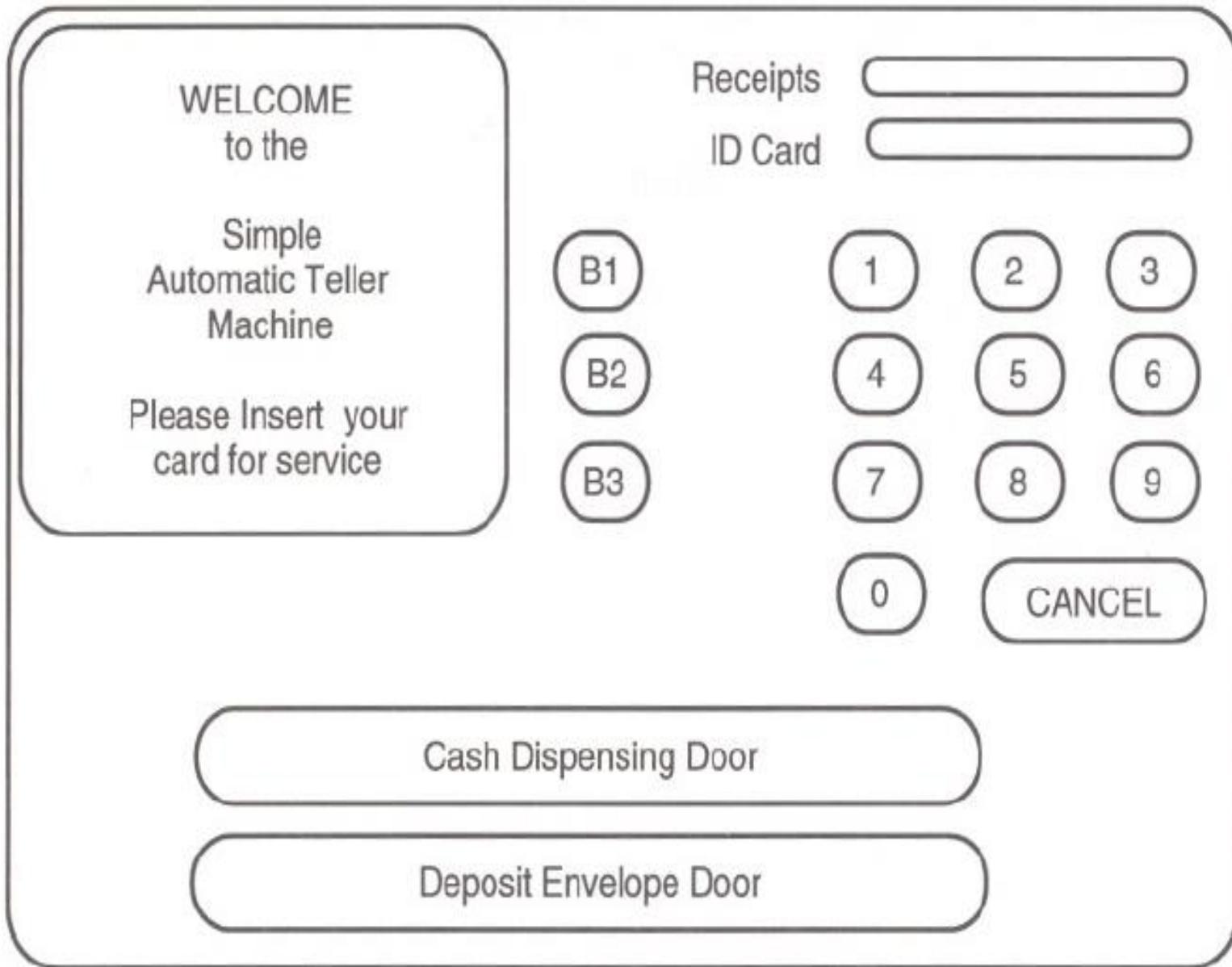


Figure 12.8 The SATM Terminal

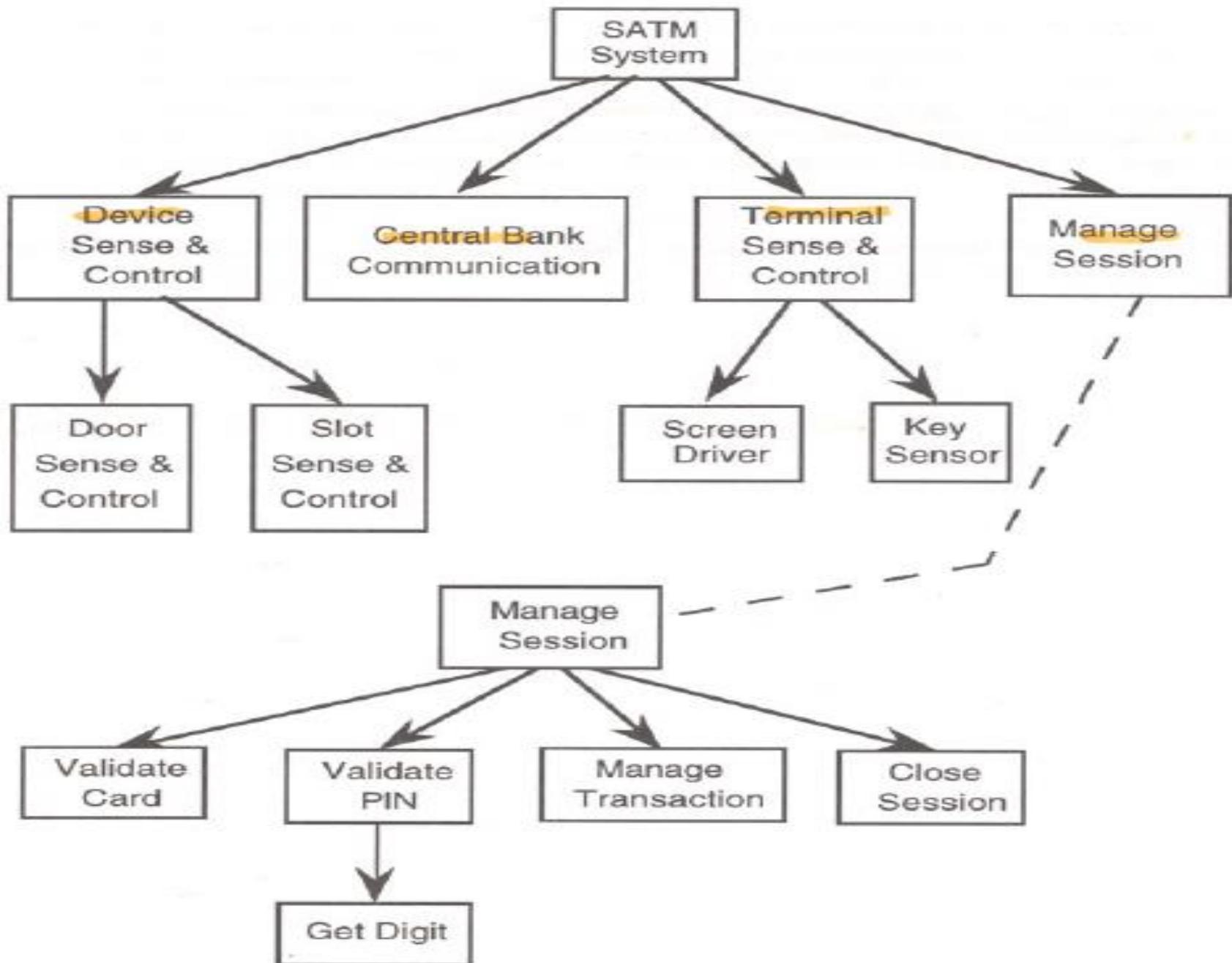


Figure 12.14 A Decomposition Tree for the SATM System

# Functional Decomposition of the SATM System

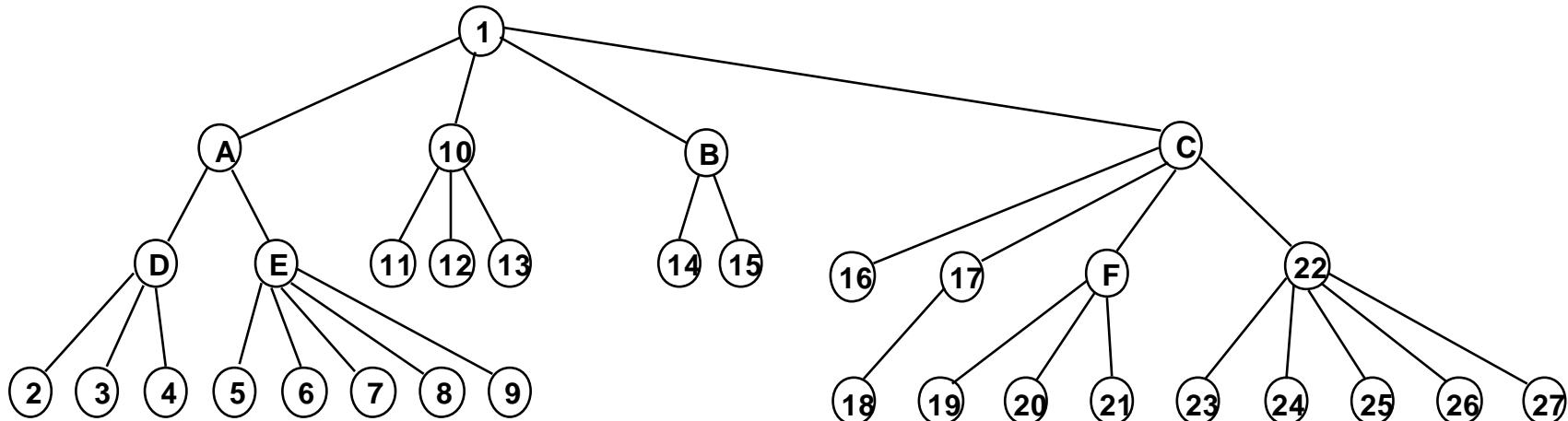
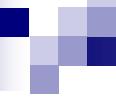


Table 1: SATM Units and Abbreviated Names

Unit	Level	Unit Name
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.3	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions

Unit	Level	Unit Nam
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal



# **Decomposition based strategies**

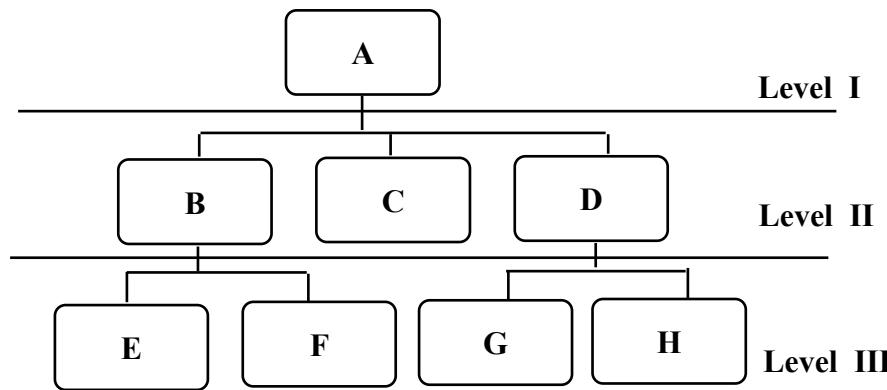
- Decomposition based:

- Top/down
  - Bottom up
  - Sandwich
  - Big bang

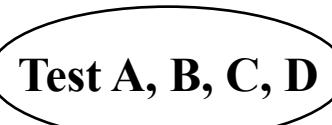
# Top-Down Integration Testing Strategy

- Top-down integration strategy focuses on testing the top layer or the controlling subsystem first (i.e. the *main*, or the root of the call tree)
- The general process in top-down integration strategy is to gradually add more subsystems that are referenced/required by the already tested subsystems when testing the application
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub*:
  - A program or a method that simulates the input-output functionality of a missing subsystem by answering to the decomposition sequence of the calling subsystem and returning back simulated or “canned” data.

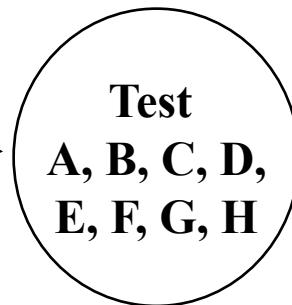
# Top-down Integration Testing Strategy



**Level I Unit(s)**



**Level I and II Units**

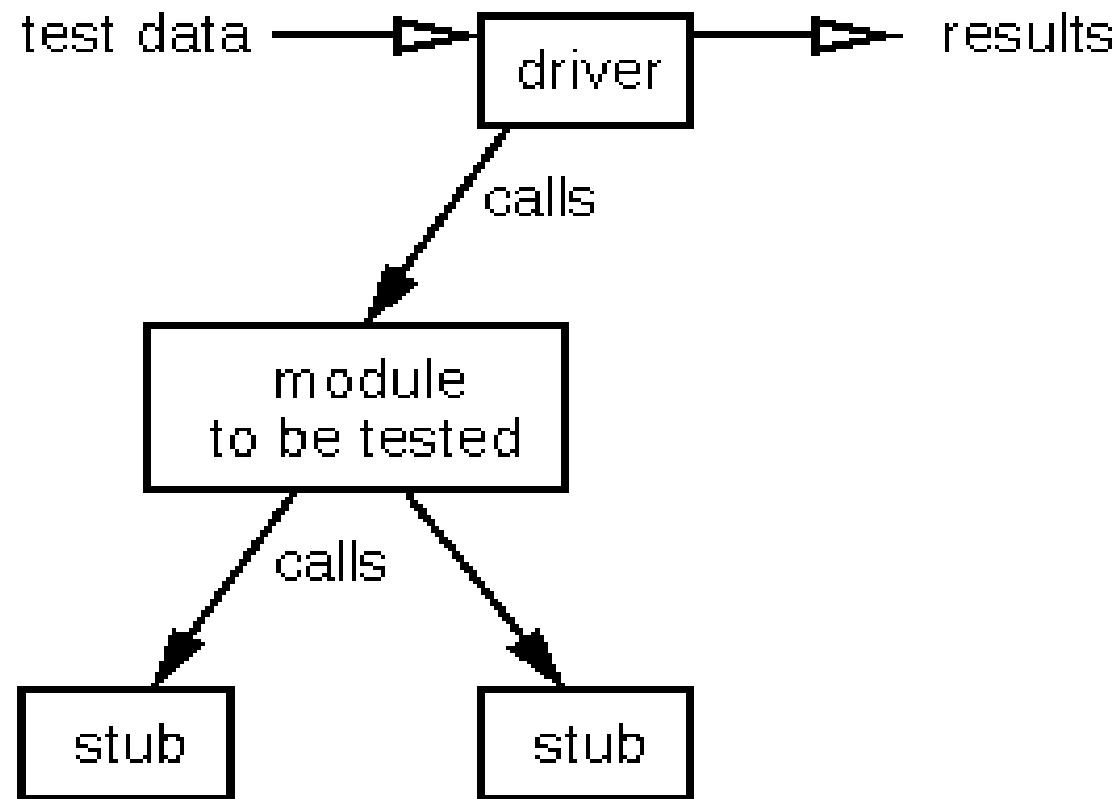


**All Units – all levels**

# Advantages and Disadvantages of Top-Down Integration Testing

- Test cases can be defined in terms of the functionality of the system (functional requirements). Structural techniques can also be used for the units in the top levels
- Writing stubs can be difficult especially when parameter passing is complex. Stubs must allow all possible conditions to be tested
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many functional units
- One solution to avoid too many stubs: *Modified top-down testing strategy (Bruege)*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, stubs and drivers are needed

# Deviation: Stubs and Drivers

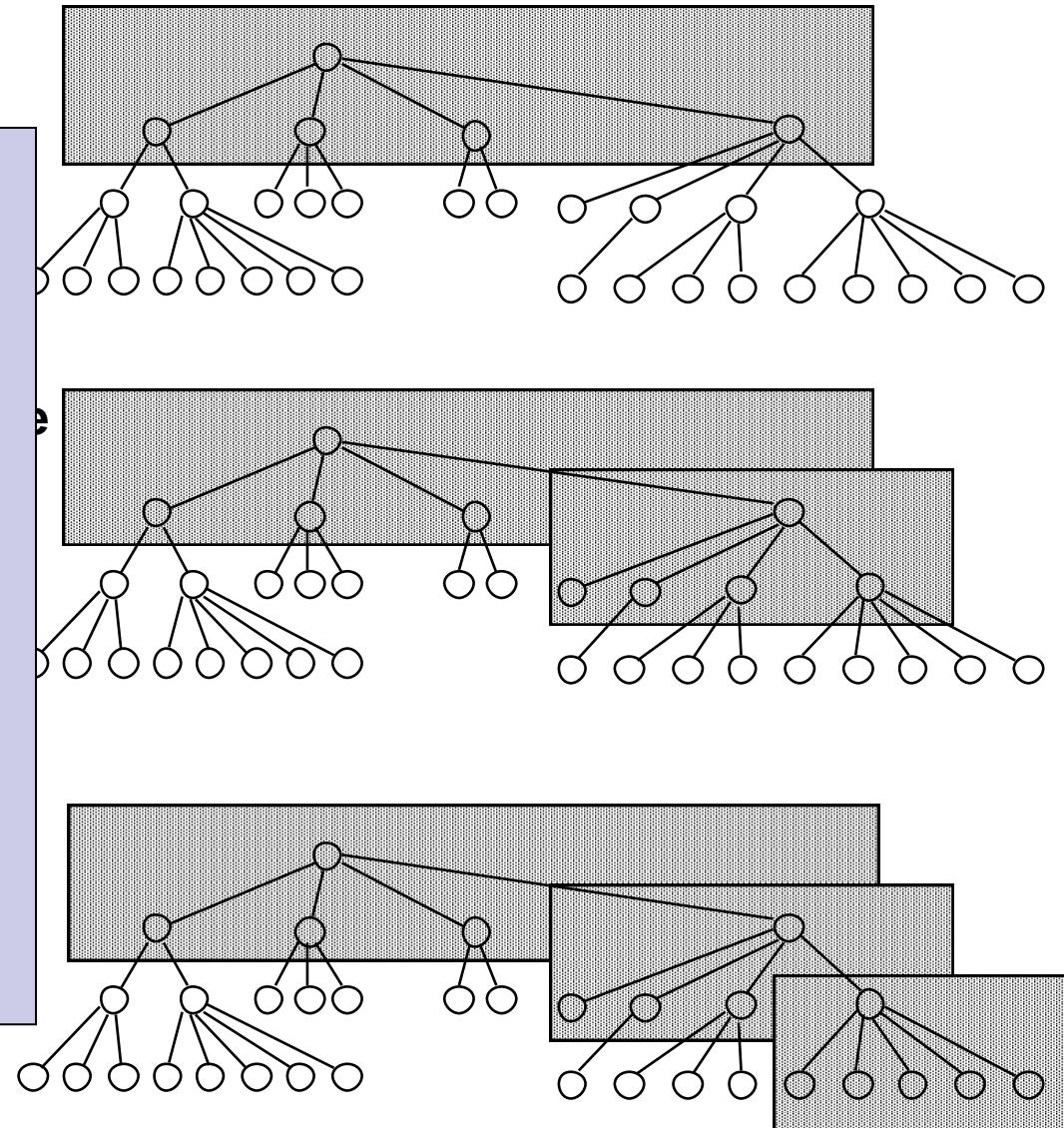


# Top-Down Integration

## Top Subtree (Sessions 1-4)

```
Procedure GetPINforPAN
(PAN, ExpectedPIN) STUB
IF PAN = '1123' THEN PIN := 
'8876';
IF PAN = '1234' THEN PIN := 
'8765';
IF PAN = '8746' THEN PIN := 
'1253';
End,
```

```
Procedure KeySensor
(KeyHit) STUB
data: KeyStrokes STACK OF '
8 '.' 8 ', ' 7 ', ' cancel '
KeyHit = POP (KeyStrokes)
End,
```



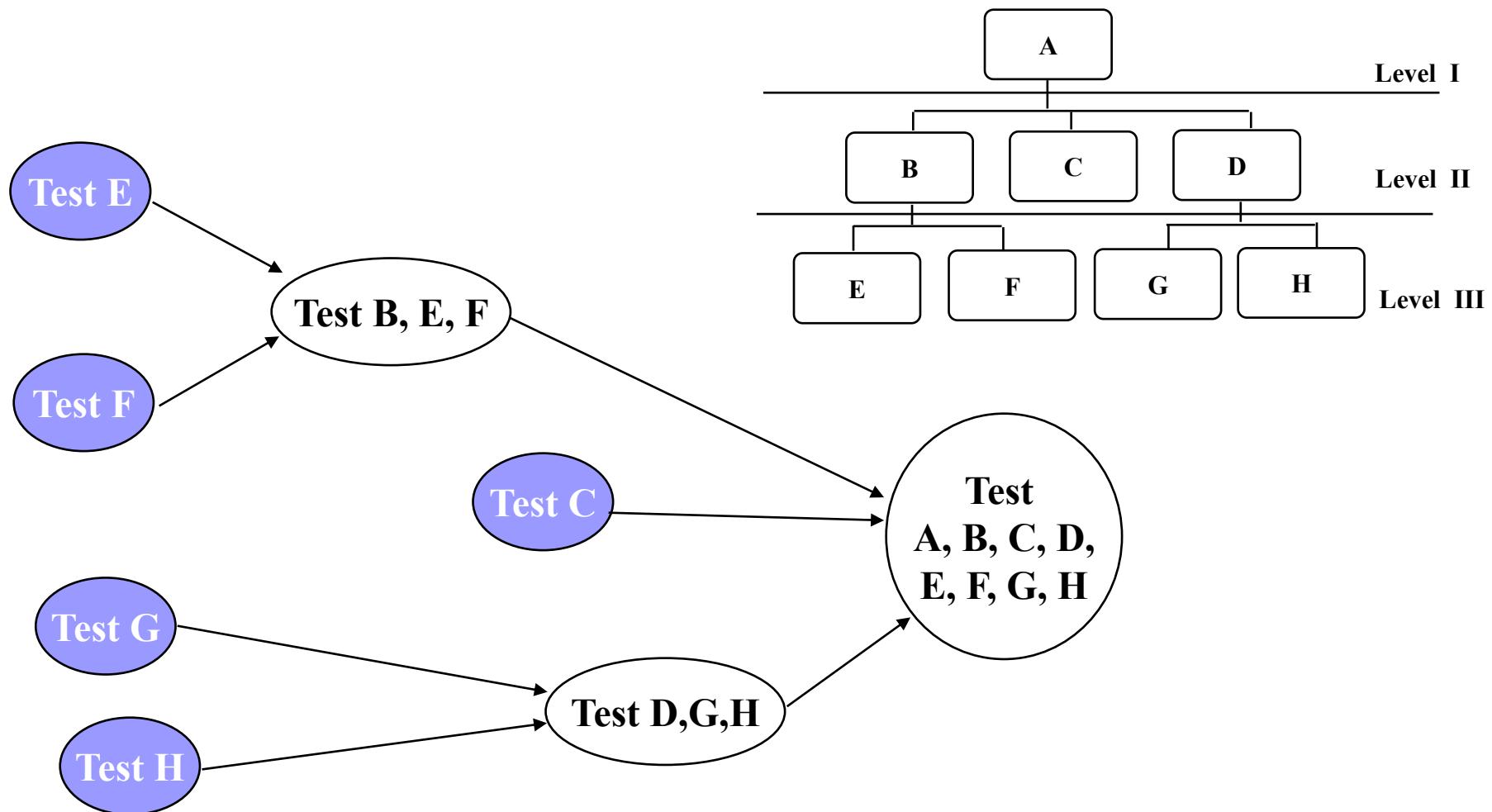
# Top-down: Complications

- The most common complication occurs when processing at low level hierarchy demands adequate testing of upper level
- To overcome:
  - Either, delay many tests until stubs are replaced with actual modules (**BAD**)
  - Or, develop stubs that perform limited functions that simulate the actual module (**GOOD**)
  - Or, Integrate the software using bottom up approach

# Bottom-Up Integration Testing Strategy

- Bottom-Up integration strategy focuses on testing the units at the lowest levels first (i.e. the units at the leafs of the decomposition tree)
- Integrate individual components in levels until the complete system is created
- The general process in bottom-up integration strategy is to gradually include the subsystems that reference/require the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special program called *Test Driver* is needed to do the testing,
  - The Test Driver is a “fake” routine that requires a subsystem and passes a test case to it

# Example Bottom-Up Strategy



# Advantages and Disadvantages of Bottom-Up Integration Testing

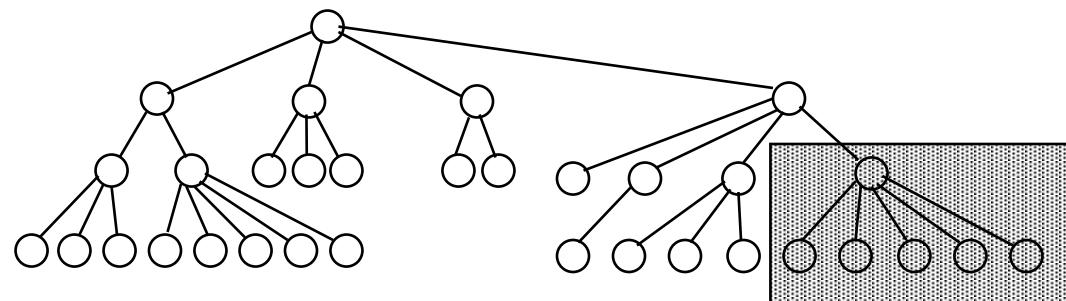
- Not optimal strategy for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - Real-time systems
  - Systems with strict performance requirements

# Bottom-up approach

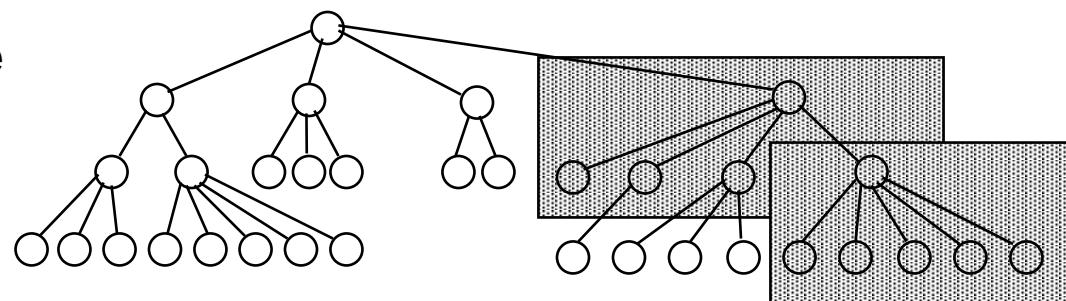
- Starts with construction and testing with atomic modules
  - No need for stub
  - Low level components are combined into cluster (or **builds**) to perform a specific sub-function
  - A driver (a control program for testing) is written to coordinate test cases input/output
  - Cluster is tested
  - Drivers are removed and clusters are combined moving upward in the program structure

# Bottom-Up Integration

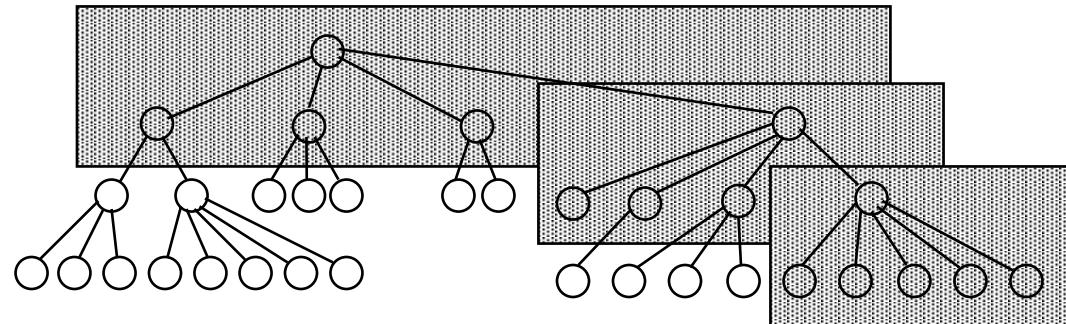
**Bottom Level Subtree  
(Sessions 13-17)**



**Second Level Subtree  
(Sessions 25-28)**



**Top Subtree  
(Sessions 29-32)**



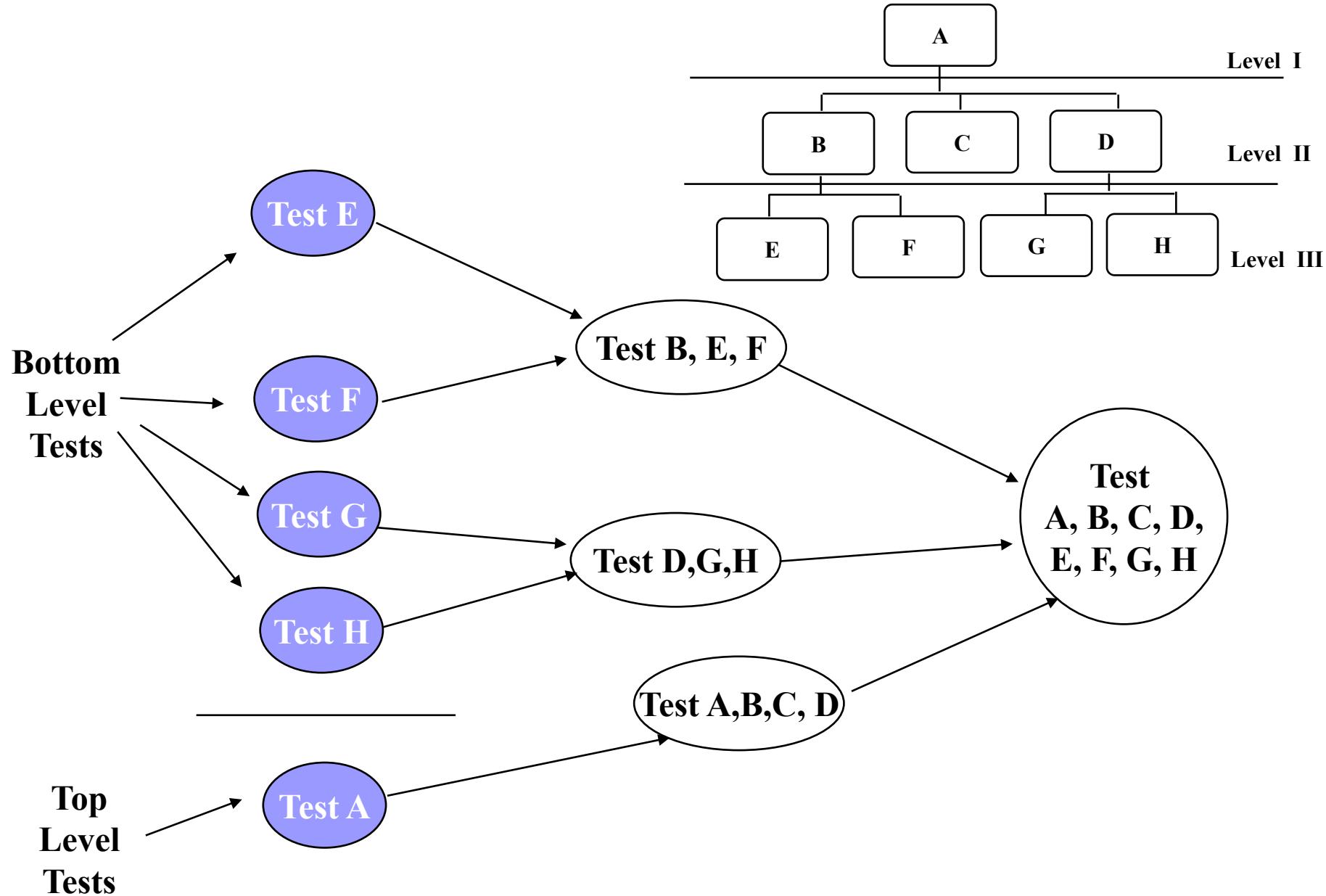
# Top-down Vs. Bottom-up

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing

# Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is viewed as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

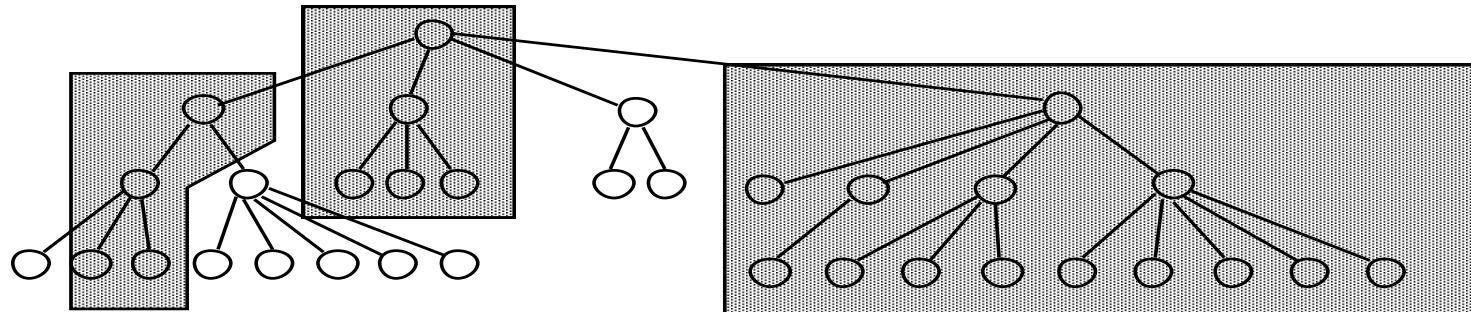
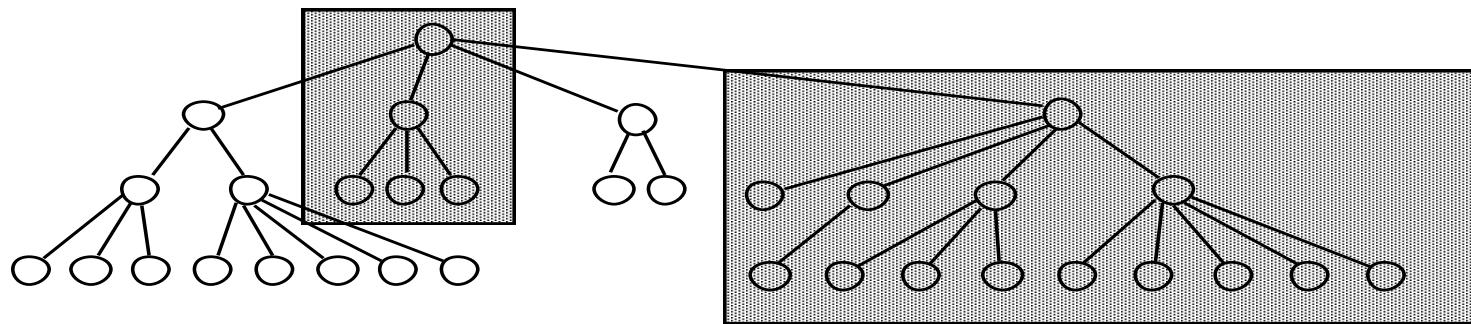
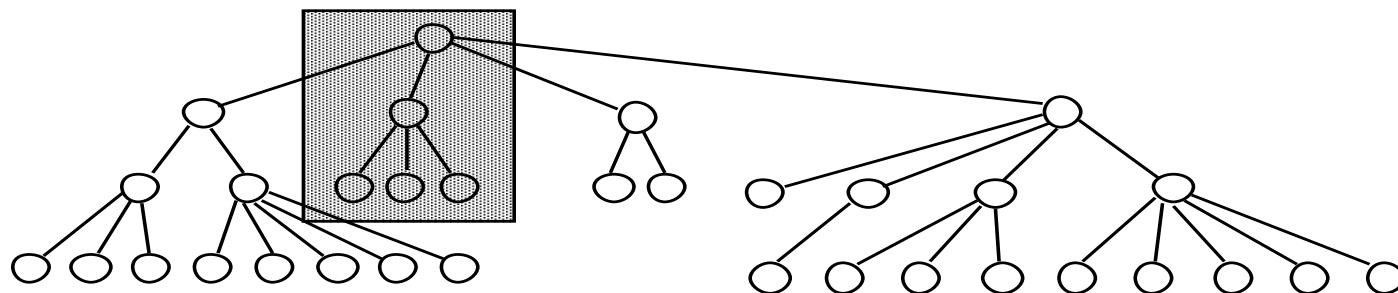
# Sandwich Testing Strategy



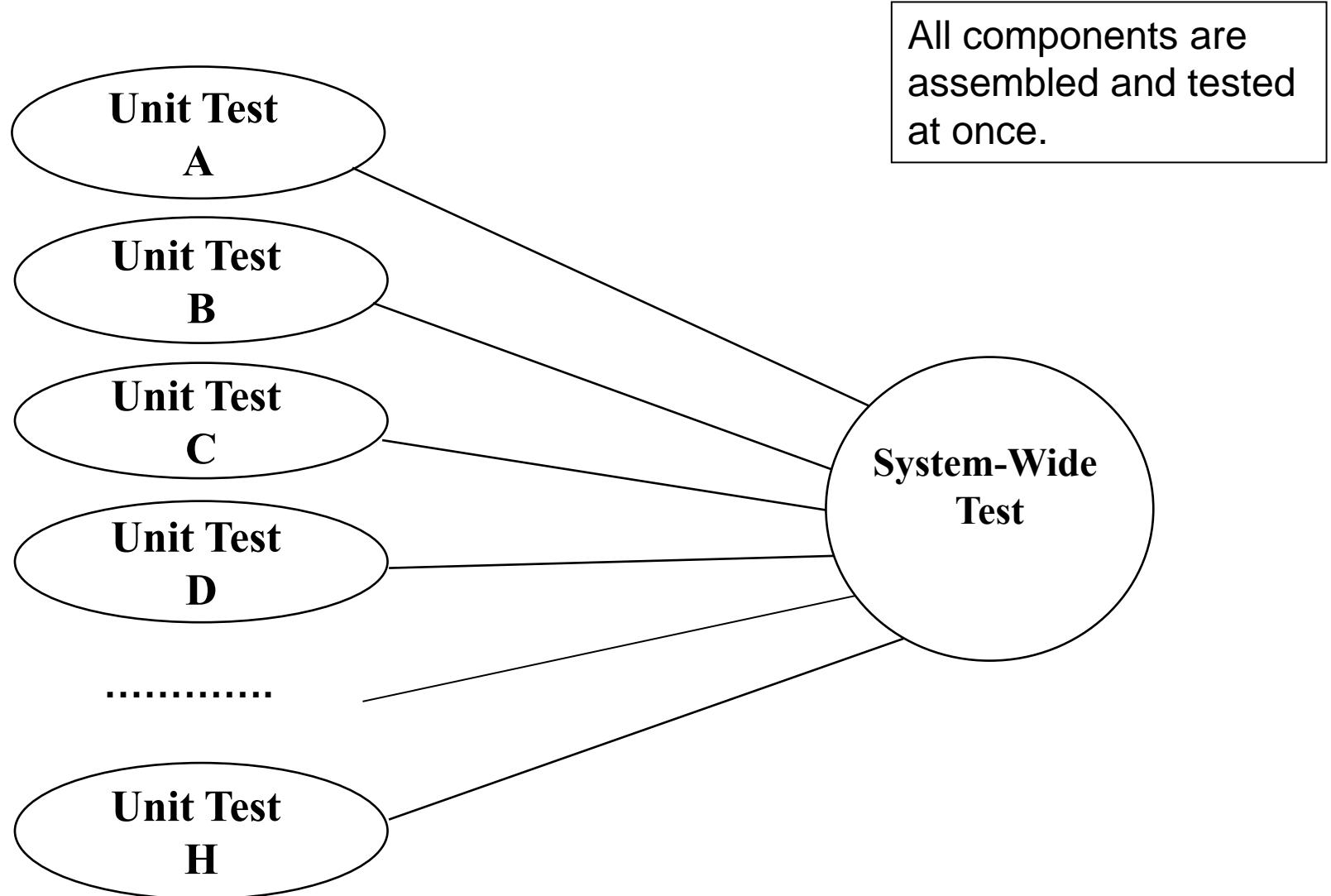
# **Advantages and Disadvantages of Sandwich Integration Testing**

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy (Bruege)

# Sandwich Integration



# Big-Bang Integration Testing



# Test Sessions

- A test session refers to **one set of tests** for a specific configuration of actual code and stubs
- The number of integration test sessions using a decomposition tree can be computed
  - Sessions=nodes – leaves + edges

# Decomposition based testing: 2

- For SATM system
  - 42 integration testing session (i.e., 42 separate sets of integration test cases)
    - top/down
      - (Nodes-1) stubs are needed
      - 32 stub in SATM
    - bottom/up
      - (Nodes-leaves) of drivers are needed
      - 10 drivers in SATM

# Decomposition based strategies: Pros and Cons

- Intuitively clear and understandable
  - In case of faults, most recently added units are suspected ones
- Can be tracked against decomposition tree
- Suggests breadth-first or depth-first traversals
- Units are merged using the decomposition tree
  - Correct behavior follows from individually correct units and interfaces
- Stubs/Drives are major development Overhead

# Call Graph Based Integration

- The basic idea is to use the call graph instead of the decomposition tree
- The call graph is a directed, labeled graph
- Two types of call graph based integration testing
  - Pair-wise Integration Testing
  - Neighborhood Integration Testing

# Pair-Wise Integration Testing

- The idea behind Pair-Wise integration testing is to eliminate the need for developing stubs/drivers
- The objective is to use actual code instead of stubs/drivers
- In order no to deteriorate the process to a big-bang strategy, we restrict a testing session to just a pair of units in the call graph
- The result is that we have one integration test session for each edge in the call graph

# Neighborhood Integration Testing

- We define the neighborhood of a node in a graph to be the set of nodes that are one edge away from the given node
- In a directed graph means all the immediate predecessor nodes and all the immediate successor nodes of a given node
- The number of neighborhoods for a given graph can be computed as:

$$\text{InteriorNodes} = \text{nodes} - (\text{SourceNodes} + \text{SinkNodes})$$

$$\text{Neighborhoods} = \text{InteriorNodes} + \text{SourceNodes}$$

Or

$$\text{Neighborhoods} = \text{nodes} - \text{SinkNodes}$$

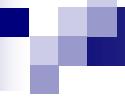
- Neighborhood Integration Testing reduces the number of test sessions

# Advantages and Disadvantages of Call-Graph Integration Testing

- Call graph based integration techniques move towards a behavioral basis
- Aim to eliminate / reduce the need for drivers/stubs
- Closer to a build sequence
- Neighborhoods can be combined to create “villages”
- Suffer from the fault isolation problem especially for large neighborhoods
- Nodes can appear in several neighborhoods

# Call graph based integration testing

- Call graph
  - A directed graph
  - Nodes corresponds to unit
  - Edges corresponds to the call
  - E.g.
    - $A \rightarrow B$  (i.e., A is calling B)
- Attempts to overcome the decomposition problem (structural)
- Moves toward behavioral testing

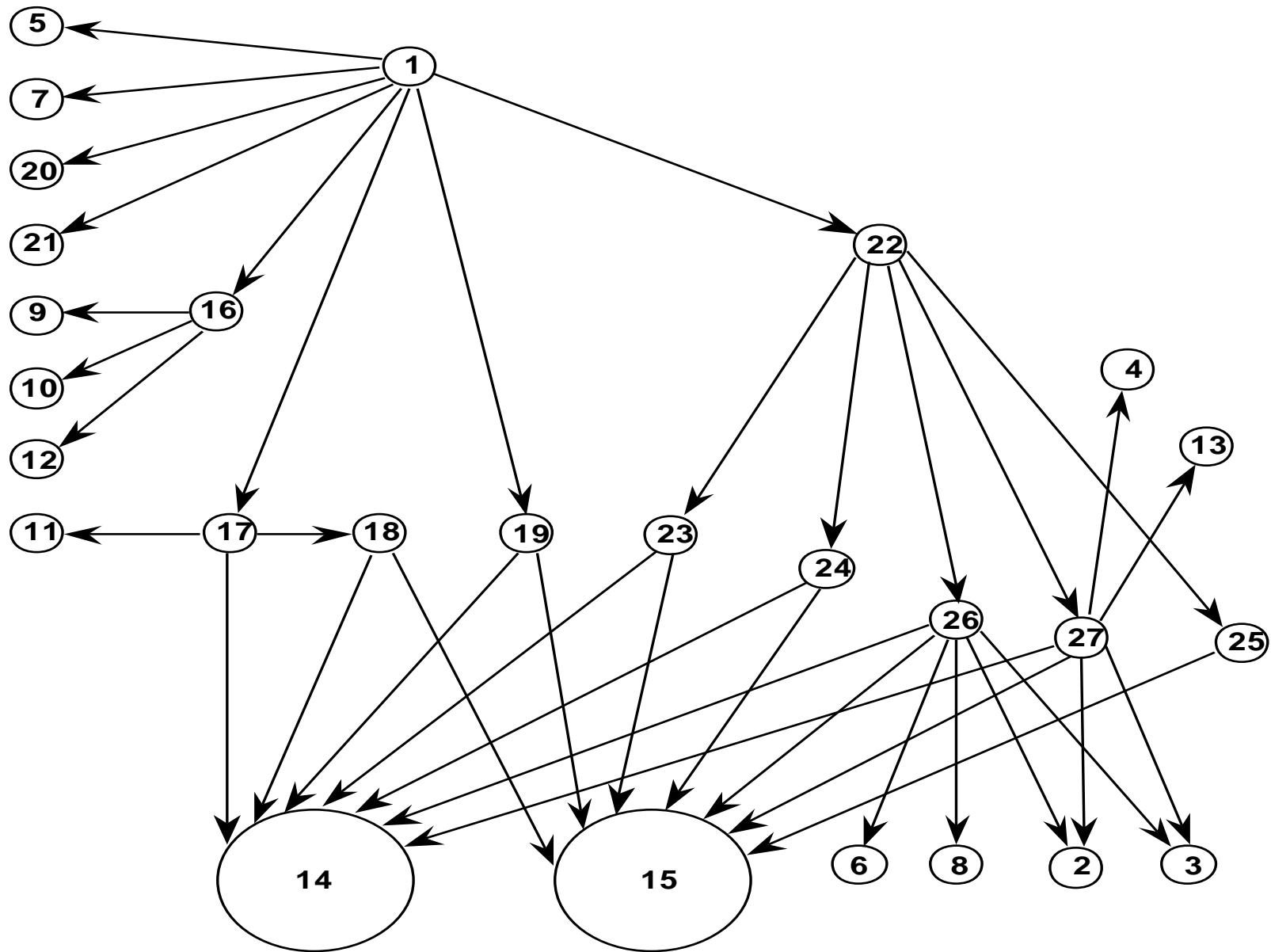


# Call graph (CG): approaches

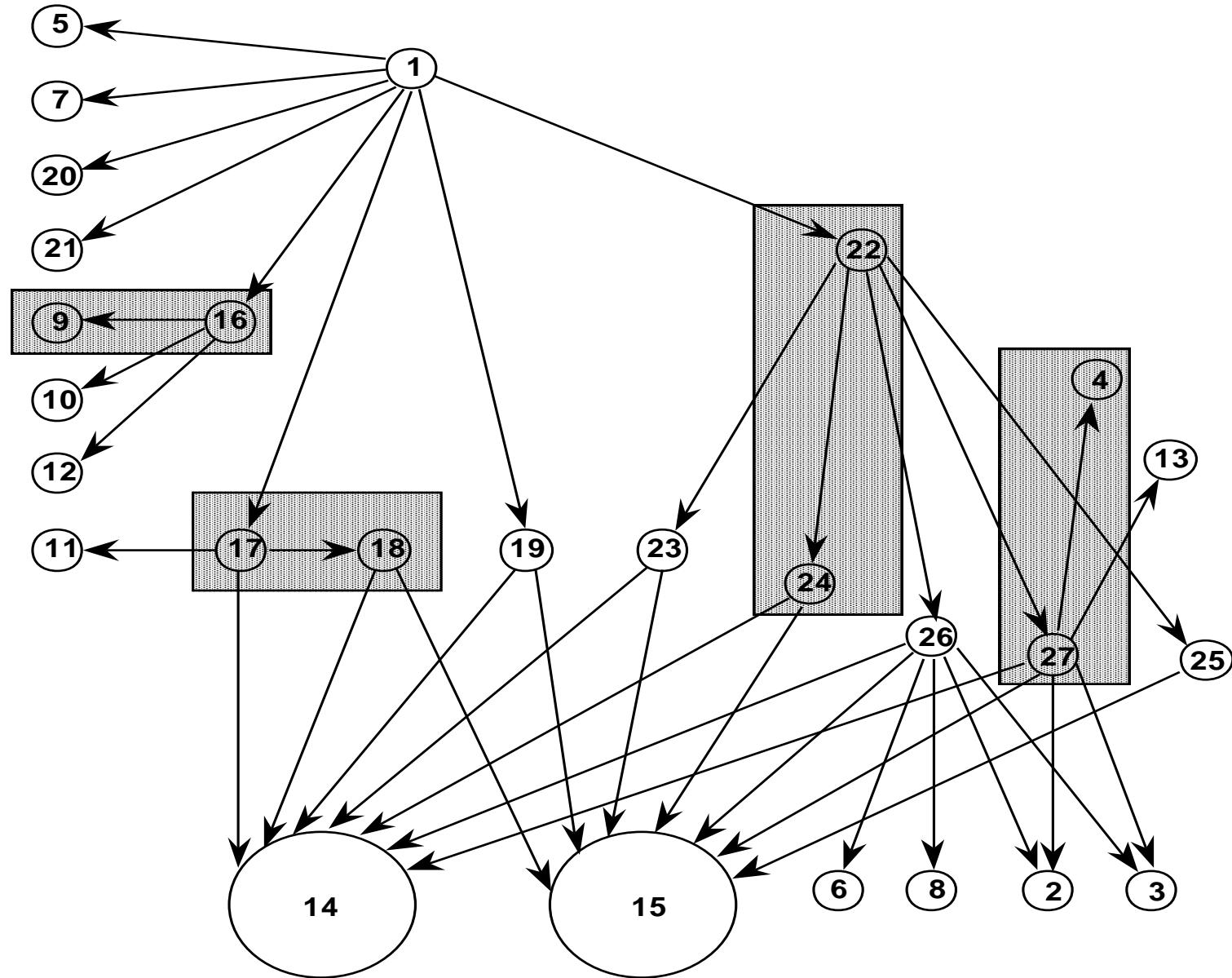
- Two main approaches based on Call Graph
  - Pair-wise integration
  - Neighborhood integration

Table 2: Adjacency Matrix for the SATM Call Graph

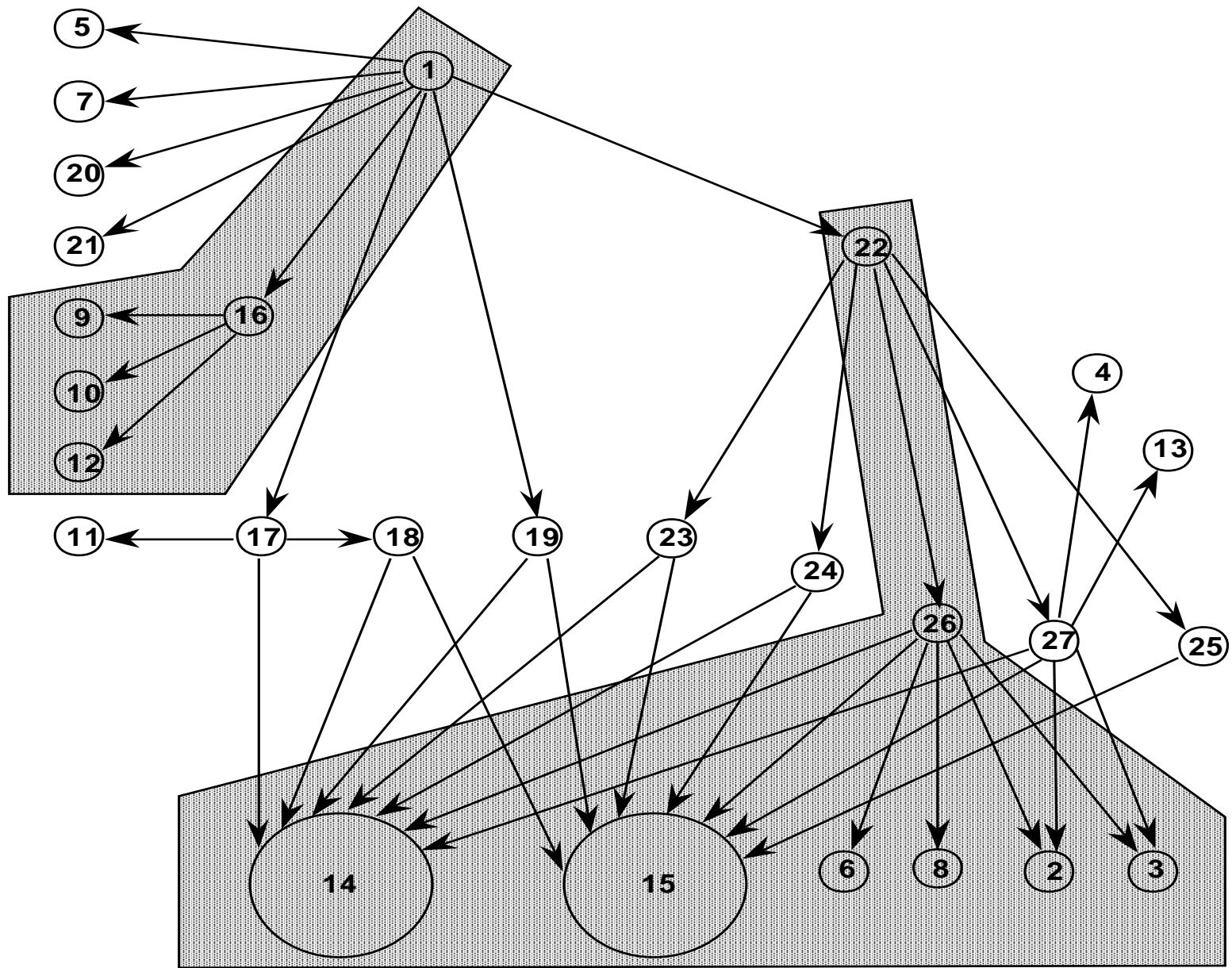
# Call Graph of the SATM System



# Some Pair-wise Integration Sessions



# Two Neighborhood Integration Sessions



# SATM Neighborhoods

Node	Predecessors	Successors
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15
23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	n/a	5, 7, 2, 21, 16, 17, 19, 22

# Pros and cons

- Benefits (GOOD)
  - Mostly behavioral than structural
  - Eliminates sub/drive overhead
  - Works well with incremental development method such as Build and composition
- Liabilities (BAD)
  - The fault isolation
  - E.g.,
    - Fault in one node appearing in several neighborhood

# Path-based Integration Testing

- The hybrid approach (i.e., structural and behavioral) is an ideal one integration testing
- The focus is on the interactions among the units
  - Interfaces are structural
  - Interactions are behavioral
- With unit testing, some path of source statements is traversed
  - What happens when there is a call to another unit?
    - Ignore the single-entry/single exit
    - Use exist follows by an entry
    - Suppress the call statement

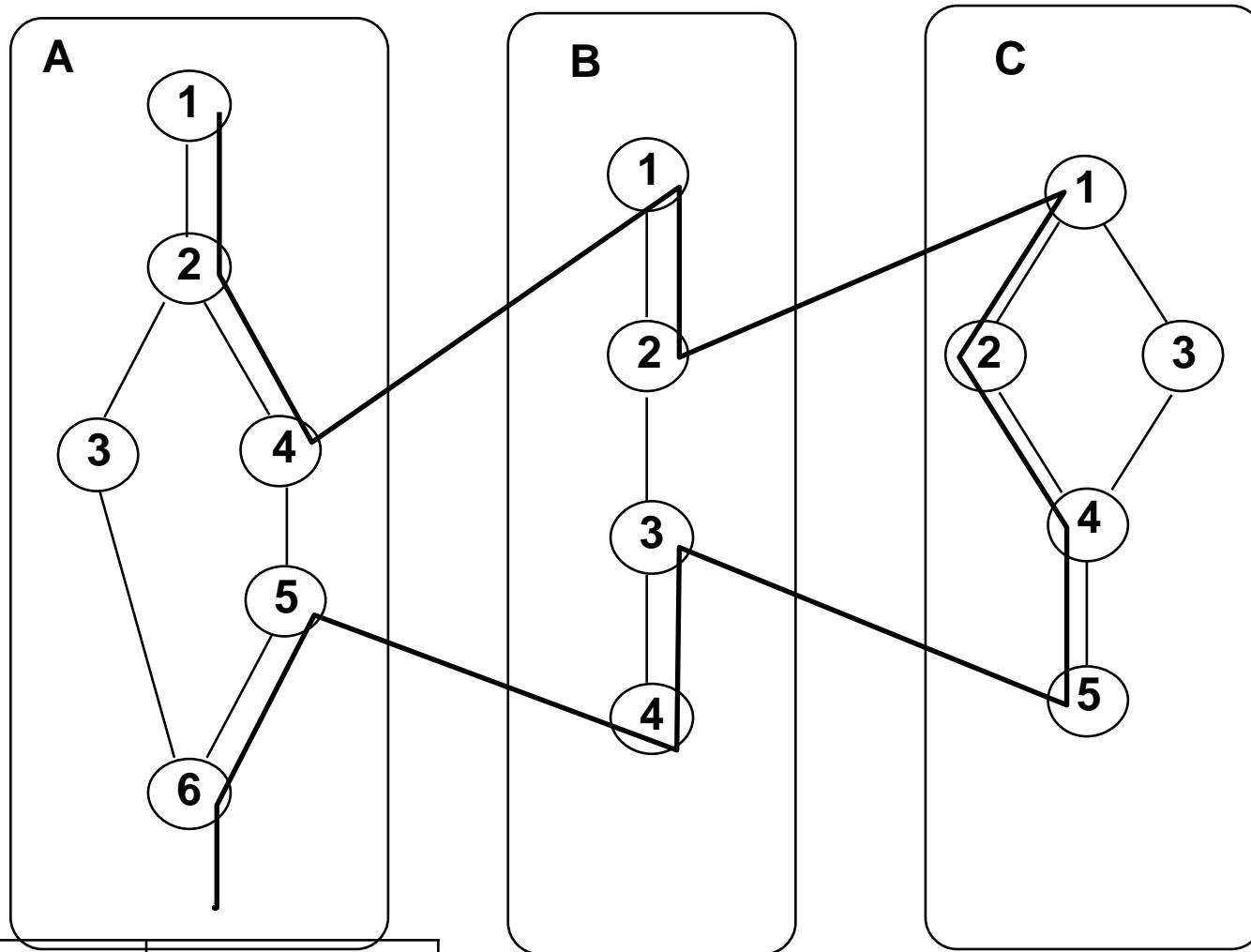
# The path based integration testing (definition)

- MM-Path (definition)
  - An interleaved sequence of module execution paths (**MEP**) and **messages**
- MM-Path can be used
  - to describe **sequences of module execution paths** including transfers of control among units using messages
  - To represents feasible execution paths that cross unit boundaries
  - To extent program graph
    - Where
      - nodes = execution paths
      - edges = messages
- Atomic system function (ASF)
  - An action that is observable at the system level in terms of port input and output events

# **MM-Path Graph (definition)**

- Given a set of units, their MM-Path graph is the directed graph in which nodes are module execution paths (MM-PATHS) and edges represents messages/returns from one unit to another
- Supports composition of units

# An MM-Path



Module	Source node	Sink node
A	1,5	4,6
B	1,3	2,4
C	1	4

# Figure 13.4

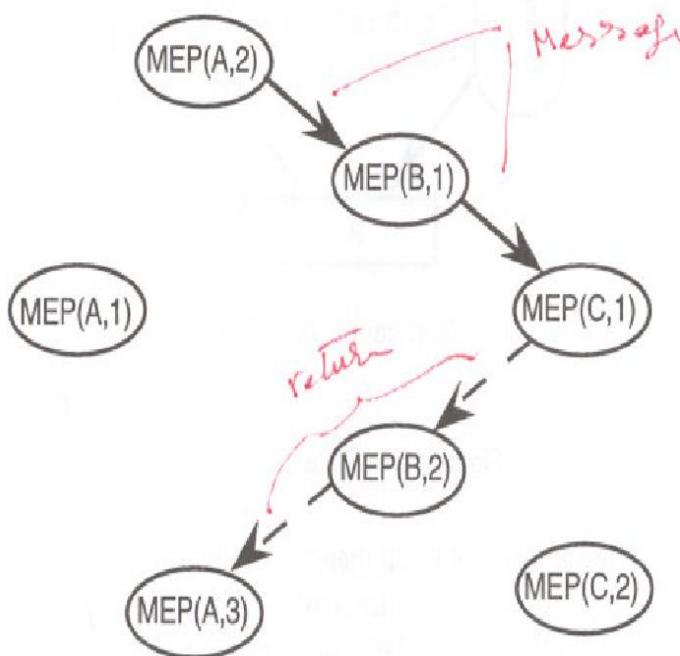


Figure 13.4 MM-Path Graph Derived from Figure 13.3

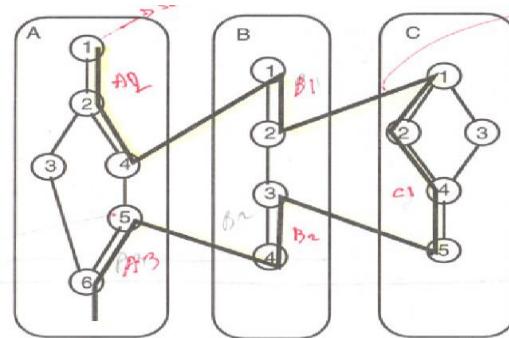


Figure 13.3 MM-Path Across Three Units

- MEP(A,1) =  $\langle 1, 2, 3, 6 \rangle$
- MEP(A,2) =  $\langle 1, 2, 4 \rangle$
- MEP(A,3) =  $\langle 5, 6 \rangle$
- MEP(B,1) =  $\langle 1, 2 \rangle$
- MEP(B,2) =  $\langle 3, 4 \rangle$
- MEP(C,1) =  $\langle 1, 2, 4, 5 \rangle$
- MEP(C,2) =  $\langle 1, 3, 4, 5 \rangle$

action testing analysis of the TDD Path graph 11

# PDL Description of SATM Main Program

1. Main Program
2. State = AwaitCard
3. CASE State OF
4. AwaitCard: ScreenDriver(1, null)
5.     WatchCardSlot(CardSlotStatus)
6.     WHILE CardSlotStatus is Idle DO
7.         WatchCardSlot(CardSlotStatus)
8.         ControlCardRoller(accept)
9.         ValidateCard(CardOK, PAN)
10.        IF CardOK THEN State = AwaitPIN
11.           ELSE ControlCardRoller(eject)
12.           State = AwaitCard
13. AwaitPIN: ValidatePIN(PINok, PAN)
14.        IF PINok THEN ScreenDriver(5, null)
15.           State = AwaitTrans
16.        ELSE ScreenDriver(4, null)
17.           State = AwaitCard
18. AwaitTrans: ManageTransaction
19.        State = CloseSession
20. CloseSession: IF NewTransactionRequest
21.        THEN State = AwaitTrans
22.        ELSE PrintReceipt
23.        PostTransactionLocal
24.        CloseSession
25.        ControlCardRoller(eject)
26.        State = AwaitCard
27. End, (CASE State)
28. END. (Main program SATM)

# PDL Description of ValidatePIN Procedure

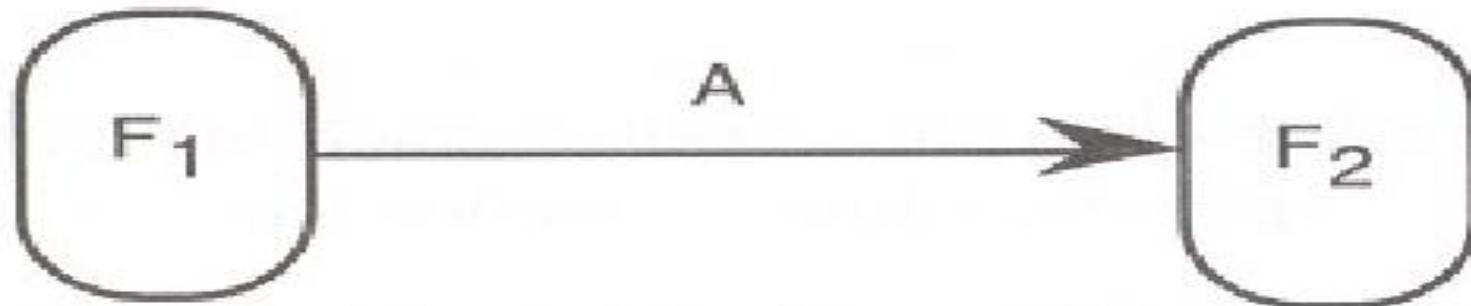
```
29. Procedure ValidatePIN(PINok, PAN)
30. GetPINforPAN(PAN, ExpectedPIN)
31. Try = First
32. CASE Try OF
33.   First: ScreenDriver(2, null)
34.     GetPIN(EnteredPIN)
35.     IF EnteredPIN = ExpectedPIN
36.       THEN PINok = TRUE
37.       RETURN
38.     ELSE ScreenDriver(3, null)
39.     Try = Second
40.   Second: ScreenDriver(2, null)
41.     GetPIN(EnteredPIN)
42.     IF EnteredPIN = ExpectedPIN
43.       THEN PINok = TRUE
44.       RETURN
45.     ELSE ScreenDriver(3, null)
46.     Try = Third
47.   Third: ScreenDriver(2, null)
48.     GetPIN(EnteredPIN)
49.     IF EnteredPIN = ExpectedPIN
50.       THEN PINok = TRUE
51.       RETURN
52.     ELSE ScreenDriver(4, null)
53.       PINok = FALSE
54. END, (CASE Try)
55. END. (Procedure ValidatePIN)
```

# PDL Description of GetPIN Procedure

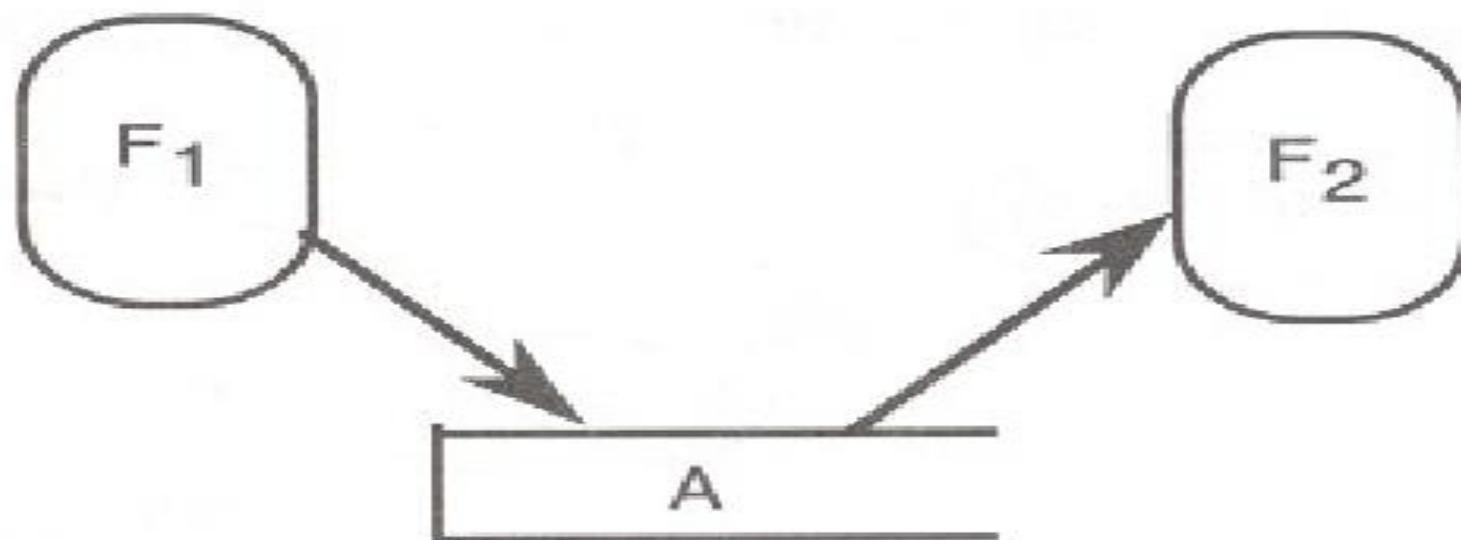
```
56. Procedure GetPIN(EnteredPIN, CancelHit)
57. Local Data: DigitKeys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
58. BEGIN
59.   CancelHit = FALSE
60.   EnteredPIN = null string
61.   DigitsRcvd=0
62.   WHILE NOT(DigitsRcvd=4 OR CancelHit) DO
63.     BEGIN
64.       KeySensor(KeyHit)
65.       IF KeyHit IN DigitKeys
66.         THEN BEGIN
67.           EnteredPIN = EnteredPIN + KeyHit
68.           INCREMENT(DigitsRcvd)
69.           IF DigitsRcvd=1 THEN ScreenDriver(2,'X--')
70.           IF DigitsRcvd=2 THEN ScreenDriver(2,'XX--')
71.           IF DigitsRcvd=3 THEN ScreenDriver(2,'XXX-')
72.           IF DigitsRcvd=4 THEN ScreenDriver(2,'XXXX')
73.         END
74.     END {WHILE}
75. END. (Procedure GetPIN)
```

# More on MM-Path

- MM-Path issues
  - How long is an MM-Path?
  - What is the endpoint?
- The following observable behavior that can be used as endpoints
  - Event quiescence ( event inactivity)
    - System level event
    - Happens when system is ideal/waiting
  - Message quiescence (msg inactivity)
    - Unit that sends no messages is reached
  - Data quiescence (data inactivity)
    - Happens when a sequences of processing generate a stored data that is not immediately used
    - E.g. account balance that is not used immediately

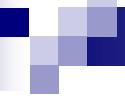


Causal data flow



Non-causal data flow

Figure 13.5 Data Quiescence



# MM-Path Guidelines

- MM-Path Guiltiness
  - Points of quiescence are natural endpoints for an MM-path
  - atomic system functions (system behavior) are considered as an upper limit for MM-Paths
    - MM-Paths should not cross ASF boundaries

# Pros and cons

- (GOOD) hybrid approach
- (GOOD) The approach works equally well for software testing developed by waterfall model
- (GOOD) Testing closely coupled with actual system behavior
- (BAD) Identification of MM-Paths which can be offset by elimination of sub/drivers

# Number of Integration Testing Sessions

## 1. Based on Functional Decomposition

Top-Down  
Bottom-Up  
Sandwich  
Big Bang

nodes-leaves+edges  
nodes-leaves+edge  
(Max is number of subtrees)  
1

## 2. Based on Call Graph

Pair-wise  
Neighborhood

number of edges  
nodes - sink nodes

## 3. Based on Paths

MM-Paths  
Atomic System Functions

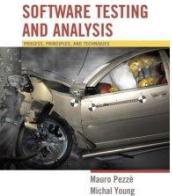
(application dependent)  
(application dependent)

# Effort and Throw-away Code

Method	Sessions	Stubs/Drivers
Top-Down	42	32 stubs
Bottom-Up	42	10 drivers
Sandwich	?	none
Big Bang	1	none
Pair-wise Neighborhood	39 11	pair dependent neighborhood dependent
MM-Path ASF	1 per MM-Path 1 per ASF	1 driver per MM-Path none

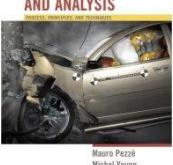


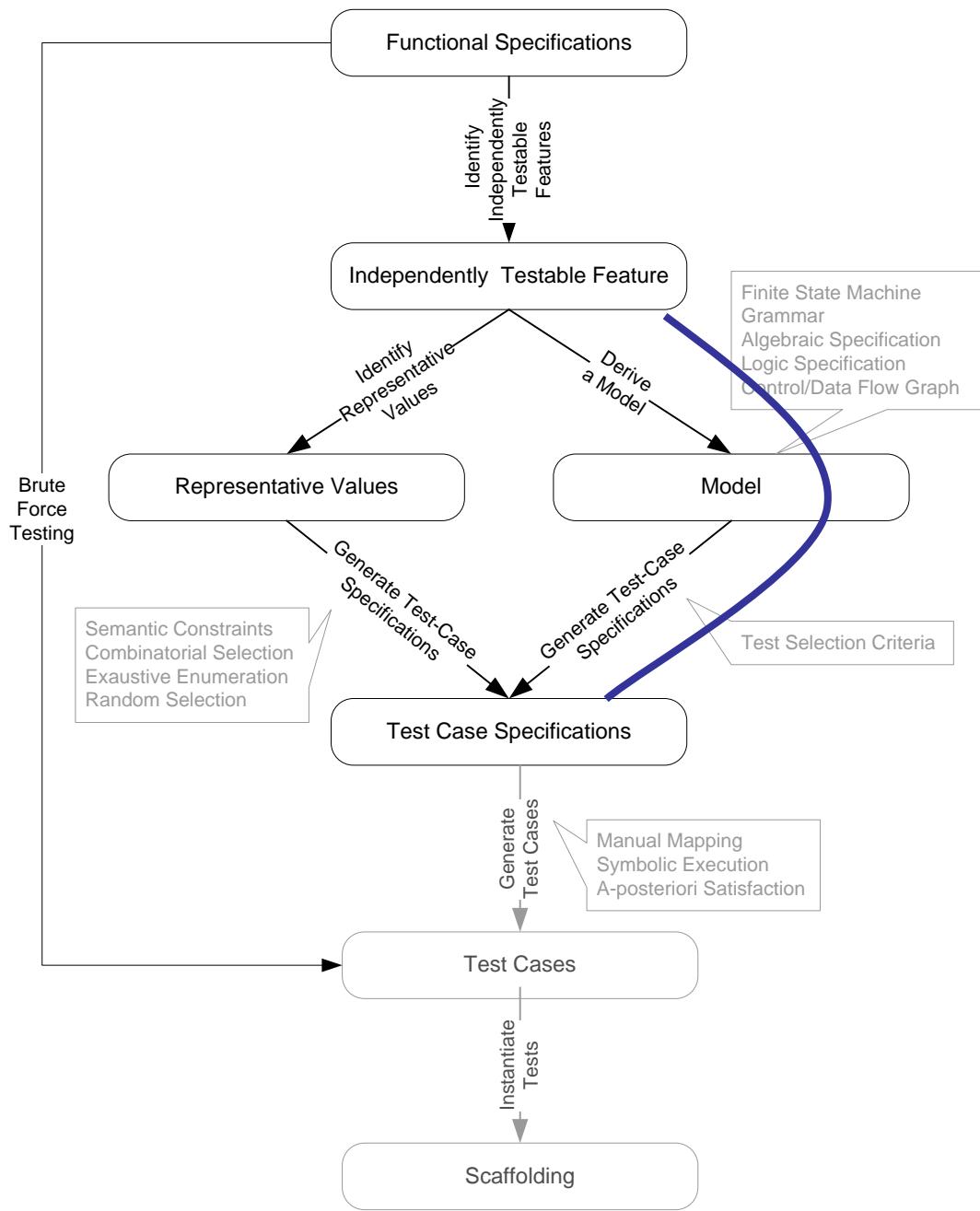
# Model based testing



# Learning Objectives

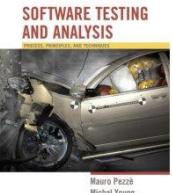
- Understand the role of models in devising test cases
  - Principles underlying functional and structural test adequacy criteria, as well as model-based testing
- Understand some examples of model-based testing techniques
  - A few of the most common model-based techniques, representative of many others
- Be able to understand, devise and refine other model-based testing techniques
  - Grasp the basic approach and rationale well enough to apply it in other contexts





# Why model-based testing?

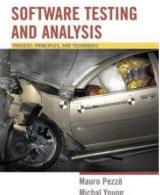
- Models used in specification or design have structure
  - Useful information for selecting representative classes of behavior; behaviors that are treated differently with respect to the model should be tried by a thorough test suite
  - In combinatorial testing, it is difficult to capture that structure clearly and correctly in constraints
- We can devise test cases to check actual behavior against behavior specified by the model
  - “Coverage” similar to structural testing, but applied to specification and design models



# Deriving test cases from finite state machines

A common kind of model for describing behavior that depends on sequences of events or stimuli

Example: UML state diagrams



# From an informal specification...

**Maintenance:** The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, the customer can request either by calling the maintenance department or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web, and the customer is a resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provides for it, the customer can determine the possibilities for the next step ...

If the product is not covered by warranty or maintenance contract, the customer can request only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.

If the customer does not accept the estimate, Small problems can be repaired directly at the maintenance station. If the station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

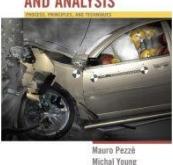
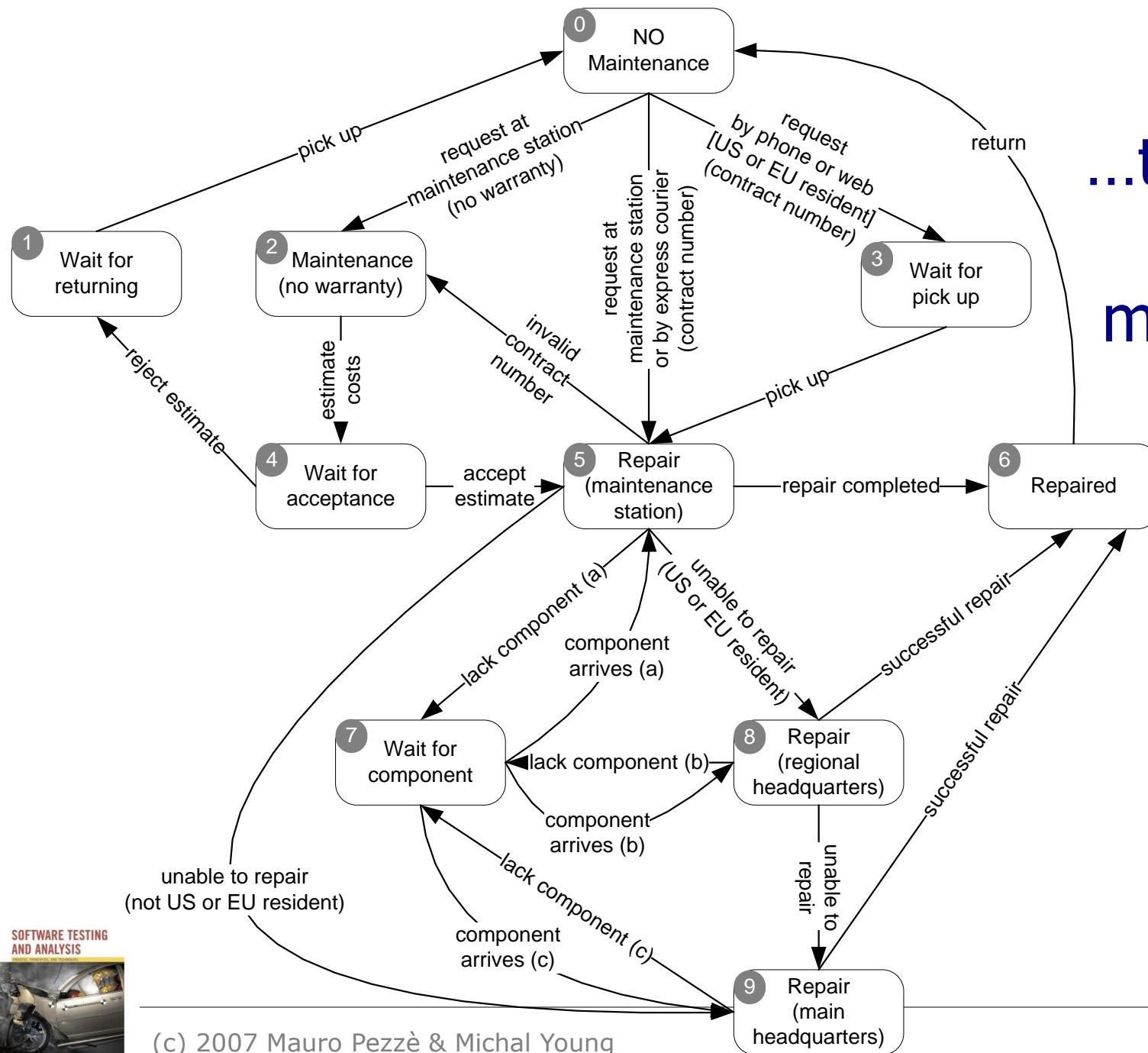
Once repaired, the product is returned to the customer.

Multiple choices in the first step ...

... determine the possibilities for the next step ...

... and so on ...

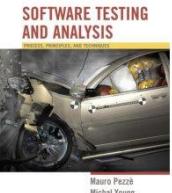
...to a finite state machine...



# ...to a test suite

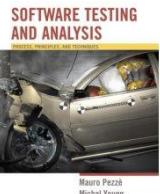
<i>TC1</i>	0	2	4	1	0	Meaning: From state 0 to state 2 to state 4 to state 1 to state 0						
<i>TC2</i>	0	5	2	4	5	6	0					
<i>TC3</i>	0	3	5	9	6	0						
<i>TC4</i>	0	3	5	7	5	8	7	8	9	6	0	

*Is this a thorough test suite?  
How can we judge?*



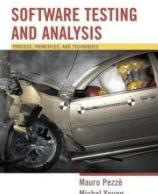
# “Covering” finite state machines

- State coverage:
  - Every state in the model should be visited by at least one test case
- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - *This is the most commonly used criterion*
    - A transition can be thought of as a (precondition, postcondition) pair



# Path sensitive criteria?

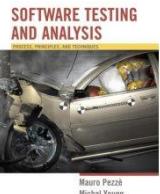
- Basic assumption: States fully summarize history
  - No distinction based on how we reached a state; this should be true of well-designed state machine models
- If the assumption is violated, we may distinguish paths and devise criteria to cover them
  - Single state path coverage:
    - traverse each subpath that reaches each state at most once
  - Single transition path coverage:
    - traverse each transition at most once
  - Boundary interior loop coverage:
    - each distinct loop of the state machine must be exercised the minimum, an intermediate, and the maximum or a large number of times
    - *Of the path sensitive criteria, only boundary-interior is common*



# Testing decision structures

Some specifications are structured as decision tables, decision trees or flow charts.

We can exercise these as if they were program source code.



# From an informal specification..

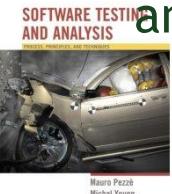
Pricing: The pricing function determines the adjusted price of a configuration for a particular customer.

The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.

There are three price schedules and three corresponding discount schedules, Business, Educational, and Individual.

....

- Educational prices: The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.
- Special-price non-discountable offers: Sometimes a complete configuration is offered at a special, non-discountable price. When a special, non-discountable price is available for a configuration, the adjusted price is the non-discountable price or the regular price after any applicable discounts, whichever is less



# ...to a decision table ...

	edu		individual						
EduAc	T	T	F	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-	-
YP > YT1	-	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T	
YP > YT2	-	-	-	-	-	-	-	-	-
SP < Sc	F	T	F	T	-	-	-	-	-
SP < T1	-	-	-	-	F	T	-	-	-
SP < T2	-	-	-	-	-	-	F	T	
out	Edu	SP	ND	SP	T1	SP	T2	SP	

# ...with constraints...

at-most-one (EduAc, BusAc)

at-most-one ( $YP < YT1$ ,  $YP > YT2$ )

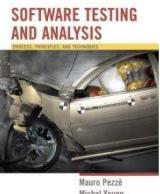
$YP > YT2 \rightarrow YP > YT1$

at-most-one ( $CP < CT1$ ,  $CP > CT2$ )

$CP > CT2 \rightarrow CP > CT1$

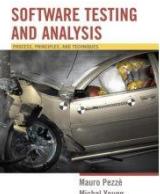
at-most-one ( $SP < T1$ ,  $SP > T2$ )

$SP > T2 \rightarrow SP > T1$



# ...to test cases

- Basic condition coverage
  - a test case specification for each column in the table
- Compound condition adequacy criterion
  - a test case specification for each combination of truth values of basic conditions
- Modified condition/decision adequacy criterion (MC/DC)
  - each column in the table represents a test case specification.
  - we add columns that differ in one input row and in outcome, then merge compatible columns



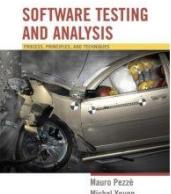
# Example MC/DC

	C.1	C.1a	C.1b	C.10
<b>EduAc</b>	T	F	T	-
<b>BusAc</b>	-	-	-	T
<b>CP &gt; CT1</b>	-	-	-	F
<b>YP &gt; YT1</b>	-	-	-	F
<b>CP &gt; CT2</b>	-	-	-	-
<b>YP &gt; YT2</b>	-	-	-	-
<b>SP &gt; Sc</b>	F	F	T	T
<b>SP &gt; T1</b>	-	-	-	-
<b>SP &gt; T2</b>	-	-	-	-
<b>out</b>	Edu	*	*	SP

Generate C.1a and C.1b by flipping one element of C.1

C.1b can be merged with an existing column (C.10) in the spec

Outcome of generated columns must differ from source column

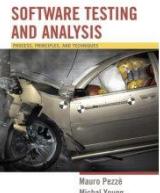


SOFTWARE TESTING  
AND ANALYSIS  
Practices, Methodology, and Tools

Mauro Pezzè  
Michal Young

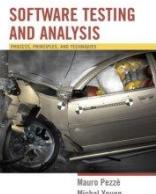
# Flowgraph based testing

If the specification or model has both decisions and sequential logic, we can cover it like program source code.



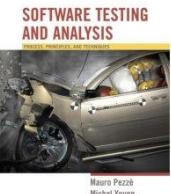
# from an informal spec (i/iii)...

- Process shipping order: The Process shipping order function checks the validity of orders and prepares the receipt  
A valid order contains the following data:
  - cost of goods: If the cost of goods is less than the minimum processable order (MinOrder) then the order is invalid.
  - shipping address: The address includes name, address, city, postal code, and country.
  - preferred shipping method: If the address is domestic, the shipping method must be either land freight, expedited land freight, or overnight air; If the address is international, the shipping method must be either air freight, or expedited air freight.



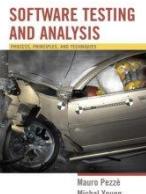
## ... (ii/iii) ...

- a shipping cost is computed based on
  - address and shipping method.
  - type of customer which can be individual, business, educational
- preferred method of payment. Individual customers can use only credit cards, business and educational customers can choose between credit card and invoice
- card information: if the method of payment is credit card, fields credit card number, name on card, expiration date, and billing address, if different than shipping address, must be provided. If credit card information is not valid the user can either provide new data or abort the order

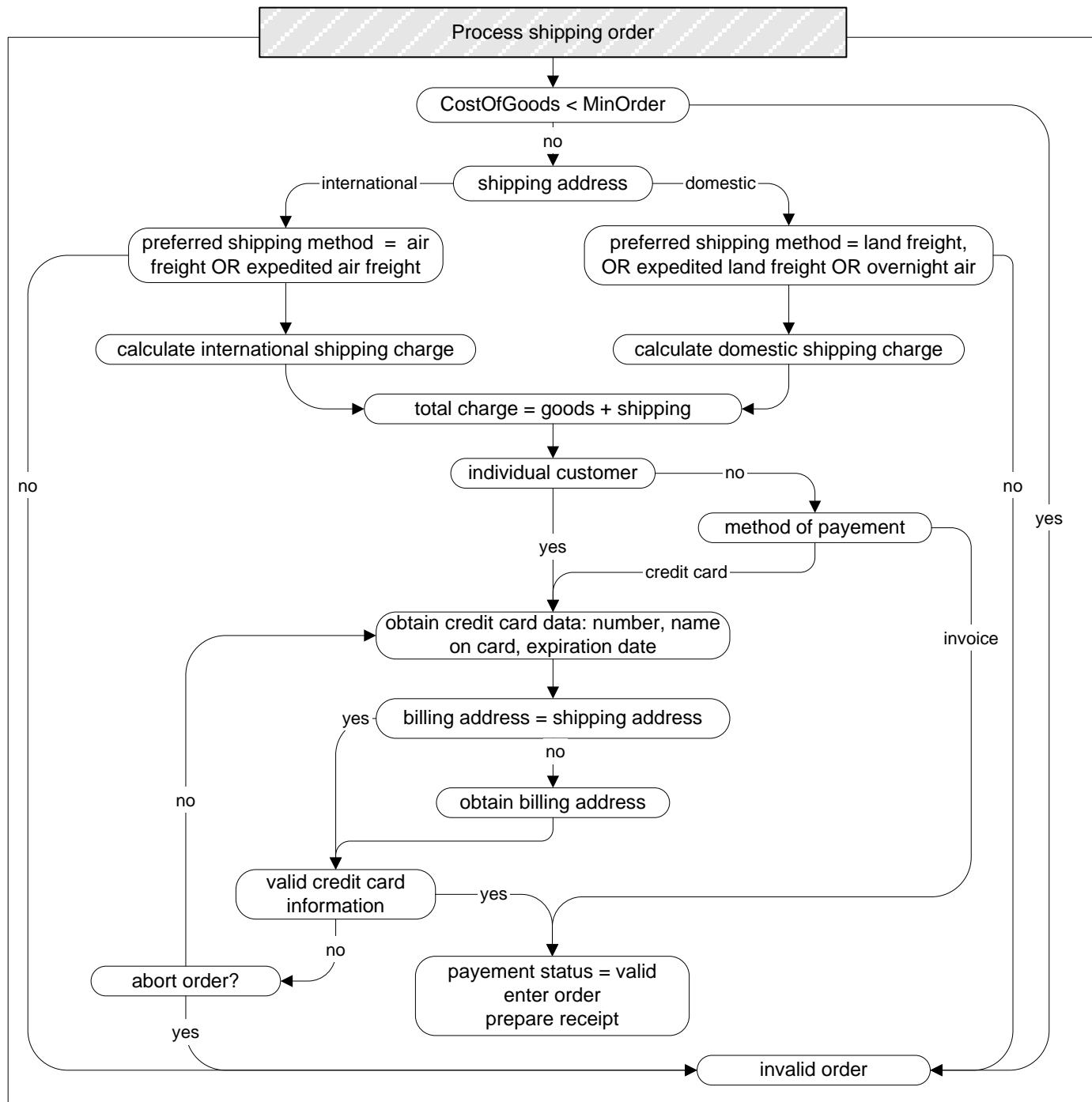


## ... (iii/iii)

- The outputs of Process shipping order are
- validity: Validity is a boolean output which indicates whether the order can be processed.
- total charge: The total charge is the sum of the value of goods and the computed shipping costs (only if validity = true).
- payment status: if all data are processed correctly and the credit card information is valid or the payment is invoice, payment status is set to valid, the order is entered and a receipt is prepared; otherwise validity = false.



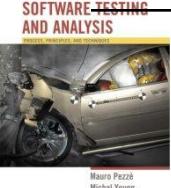
# ...to a flowgraph



# ...from the flow graph to test cases

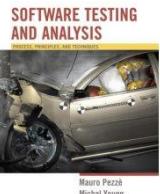
Branch testing: cover all branches

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	-	-	-	-
TC-3	Yes	-	-	-	-	-	-
TC-4	No	Dom	Air	-	-	-	-
TC-5	No	Int	Land	-	-	-	-
TC-6	No	-	-	Edu	Inv	-	-
TC-7	No	-	-	-	CC	Yes	-
TC-8	No	-	-	-	CC	-	No (abort)
TC-9	No	-	-	-	CC	-	No (no abort)



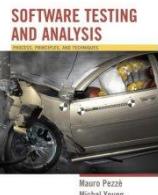
# Grammar-based testing

Complex input is (or can) often be described  
by a context-free grammar



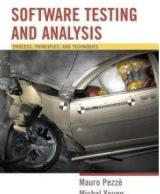
# Grammars in specifications

- Grammars are good at:
  - Representing inputs of varying and unbounded size
  - With recursive structure
  - And boundary conditions
- Examples:
  - Complex textual inputs
  - Trees (search trees, parse trees, ... )
    - Note XML and HTML are trees in textual form
  - Program structures
    - Which are also tree structures in textual format!



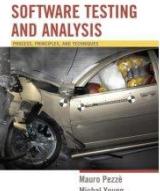
# Grammar-based testing

- Test cases are strings *generated* from the grammar
- Coverage criteria:
  - Production coverage: each production must be used to generate at least one (section of) test case
  - Boundary condition: annotate each recursive production with minimum and maximum number of application, then generate:
    - Minimum
    - Minimum + 1
    - Maximum - 1
    - Maximum



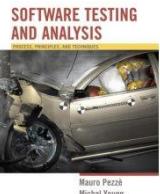
# from an informal specification (i/iii)...

- The Check-configuration function checks the validity of a computer configuration.
- The parameters of check-configuration are:
  - Model
  - Set of components



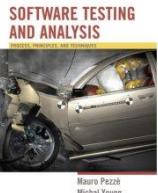
## ... (ii/iii)...

- Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs
  - Example: The required ``slots'' of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.



## ... (iii/iii)

- Set of Components: A set of [slot, component] pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value empty is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

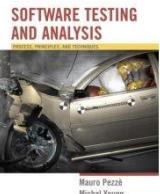


# ...to a grammar

<Model>	::= <modelNumber> <compSequence> <optCompSequence>
<compSequence>	::= <Component> <compSequence>   empty
<optCompSequence>	::= <OptionalComponent> <optCompSequence>   empty
<Component>	::= <ComponentType> <ComponentValue>
<OptionalComponent>	::= <ComponentType>
<modelNumber>	::= string
<ComponentType>	::= string
<ComponentValue>	::= string

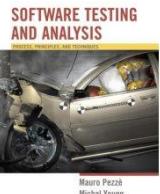
# ...to a grammar with limits

Model	<Model>	::= <modelNumber> <compSequence> <optCompSequence>
compSeq1 [0, 16]	<compSequence>	::= <Component> <compSequence>
compSeq2	<compSequence>	::= empty
optCompSeq1 [0, 16]	<optCompSequence>	::= <OptionalComponent> <optCompSequence>
optCompSeq2	<optCompSequence>	::= empty
Comp	<Component>	::= <ComponentType> <ComponentValue>
OptComp	<OptionalComponent>	::= <ComponentType>
modNum	<modelNumber>	::= string
CompTyp	<ComponentType>	::= string
CompVal	<ComponentValue>	::= string



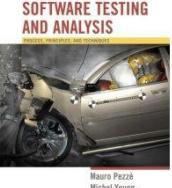
# ...to test cases

- “Mod000”
  - Covers Model, compSeq1[0], compSeq2, optCompSeq1[0], optCompSeq2, modNum
- “Mod000 (Comp000, Val000) (OptComp000)”
  - Covers Model, compSeq1[1], compSeq2, optCompSeq2[0], optCompSeq2, Comp, OptComp, modNum, CompTyp, CompVal
- Etc...
- Comments:
  - By first applying productions with nonterminals on the right side, we obtain few, large test cases
  - By first applying productions with terminals on the right side, we obtain many, small test cases



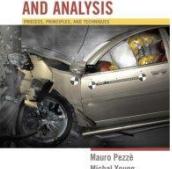
# Grammar vs. Combinatorial Testing

- Combinatorial specification-based testing is good for “mostly independent” parameters
  - We can incorporate a few constraints, but complex constraints are hard to represent and use
  - We must often “factor and flatten”
    - E.g., separate “set of slots” into characteristics “number of slots” and predicates about what is in the slots (all together)
- Grammar describes sequences and nested structure naturally
  - But some relations among different parts may be difficult to describe and exercise systematically, e.g., compatibility of components with slots



# Summary: The big picture

- Models are useful abstractions
  - In specification and design, they help us think and communicate about complex artifacts by emphasizing key features and suppressing details
  - Models convey structure and help us focus on one thing at a time
- We can use them in systematic testing
  - If a model divides behavior into classes, we probably want to exercise each of those classes!
  - Common model-based testing techniques are based on state machines, decision structures, and grammars
    - but we can apply the same approach to other models





# Object Oriented testing

# What is Object Oriented Programming?

- Map the real world as “objects”
- Objects have “state”
- Objects have “behavior”
- Original plan: Objects could be reused without modification or additional testing

# What is Object Oriented Programming?

- What is object oriented programming?
- "Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects."
- What is a class?
- "A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries."

# Class Example

Class: Person( ... )

States: Age, height, weight, phone number, gender (these are *instance variables*)

Behaviors: getHeight( ), getWeight( ), computeBMI( ),  
computeLifeExpectancy( )  
(these are *methods*)

# OO Testing

- OO Testing takes mostly in lower abstraction levels
- Nature of OO systems influence both testing strategy and methods
- Will re-use mean less need for testing? NO
- In Object Oriented systems the view of testing is broadened to encompass Analysis and Design
- *“It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.”*
- Allows early circumvention of later problems

# Differences between procedural testing and OO testing

- Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these approaches.

According to Davis the dependencies occurring in conventional systems are:

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variable and its types.

But in Object-Oriented systems there are following additional dependencies:

- Class to class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

## Issues in OO Testing

### Issues in Testing Classes:

Additional testing techniques are, therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e, objects can be tested. Similarly, the concept of inheritance opens various issues e.g., if changes are made to a parent class or superclass, in a larger system of a class it will be difficult to test subclasses individually and isolate the error to one class.

In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

## **Object Oriented Testing methods:**

- Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

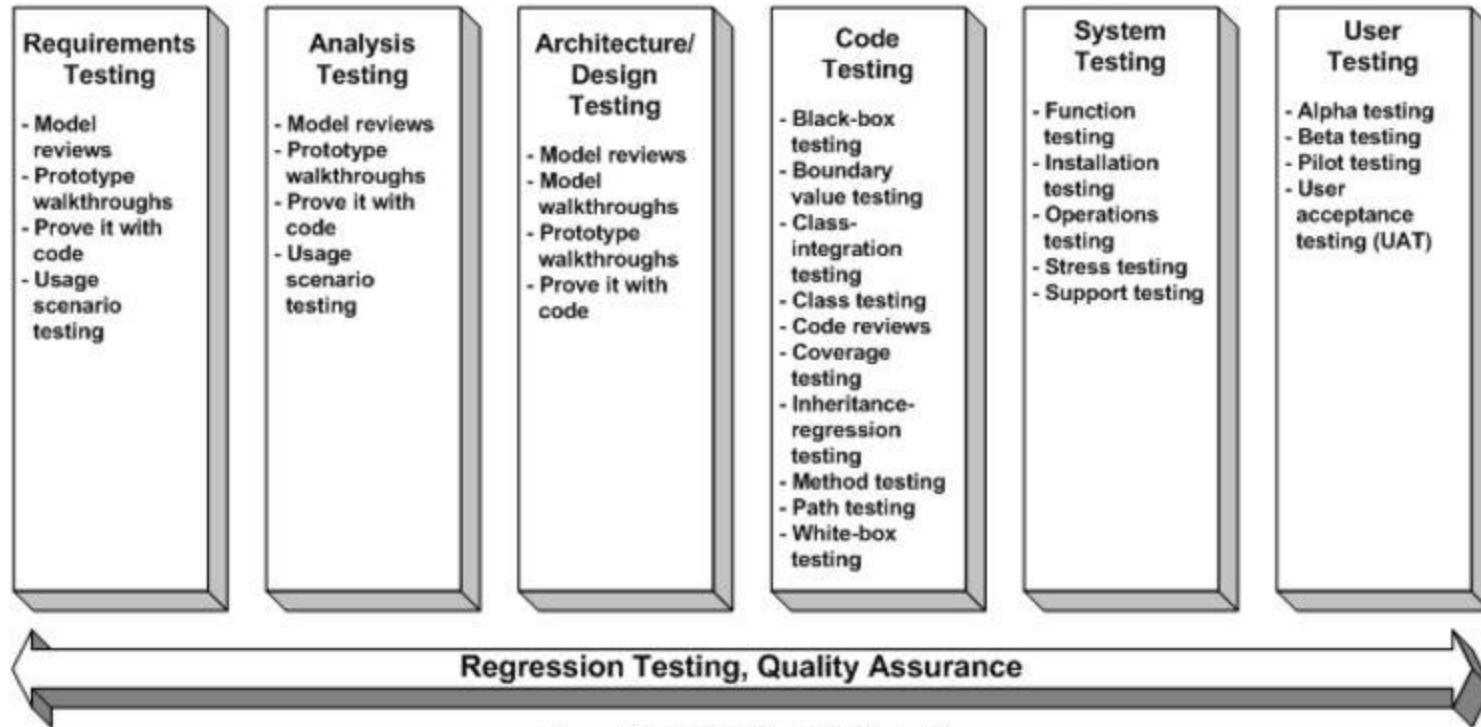
### **Unit Testing:**

- In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free.
- Unit testing is the responsibility of the application engineer who implements the structure.

### **Subsystem Testing:**

- This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside.
- Subsystem tests can be used as regression tests for each newly released version of the subsystem.

- **System Testing:**
- System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.



Copyright 2004 Scott W. Ambler

## Object-Oriented Testing Techniques:

### Grey Box Testing:

- The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:
- **State model based testing:** This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing:** Each scenario in each use case is tested.
- **Class diagram based testing:** Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing:** The methods in the messages in the sequence diagrams are tested.

### Techniques for Subsystem Testing:

- The two main approaches of subsystem testing are:
- **Thread based testing:** All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing:** The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

## Categories of System Testing:

- **Alpha testing:** This is carried out by the testing team within the organization that develops software.
- **Beta testing:** This is carried out by select group of co-operating customers.
- **Acceptance testing:** This is carried out by the customer before accepting the deliverables.
- **Black-box testing:**  
Testing that verifies the item being tested when given the appropriate input provides the expected results.
- **Boundary-value testing:**  
Testing of unusual or extreme situations that an item should be able to handle.
- **Class testing:**  
The act of ensuring that a class and its instances (objects) perform as defined.
- **Component testing:**  
The act of validating that a component works as defined.
- **Inheritance-regression testing:**  
The act of running the test cases of the super classes, both direct and indirect, on a given subclass.
- **Integration testing:**  
Testing to verify several portions of software work together.

- **Model review:**  
An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.
- **Path testing:**  
The act of ensuring that all logic paths within your code are exercised at least once.
- **Regression testing:**
- The acts of ensuring that previously tested behaviors still work as expected after changes have been made to an application.
  
- **Stress testing:**
- The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.
- **Technical review:**
- A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review.
- **User interface testing:**
- The testing of the user interface (UI) to ensure that it follows accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing.
- **White-box testing:**  
Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing.

# OBJECT ORIENTED TESTING

SYSTEM TESTING

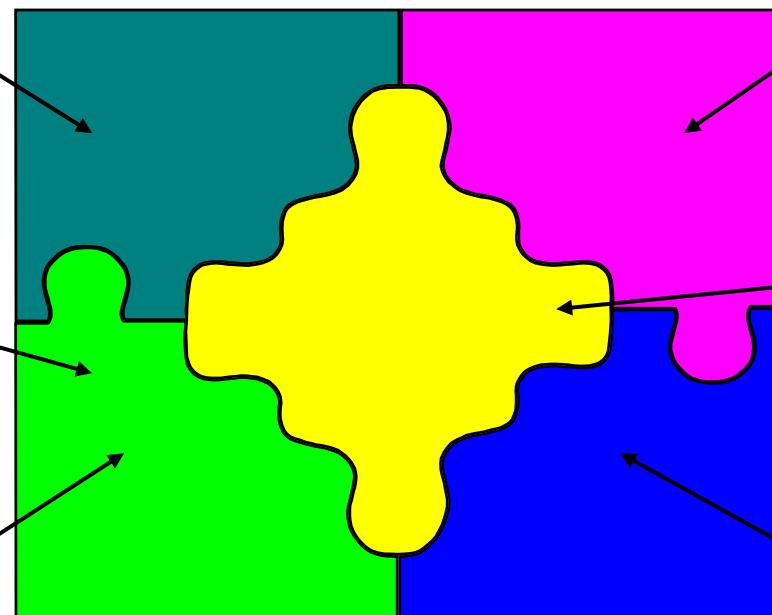
INHERITANCE

INTEGRATION  
TESTING

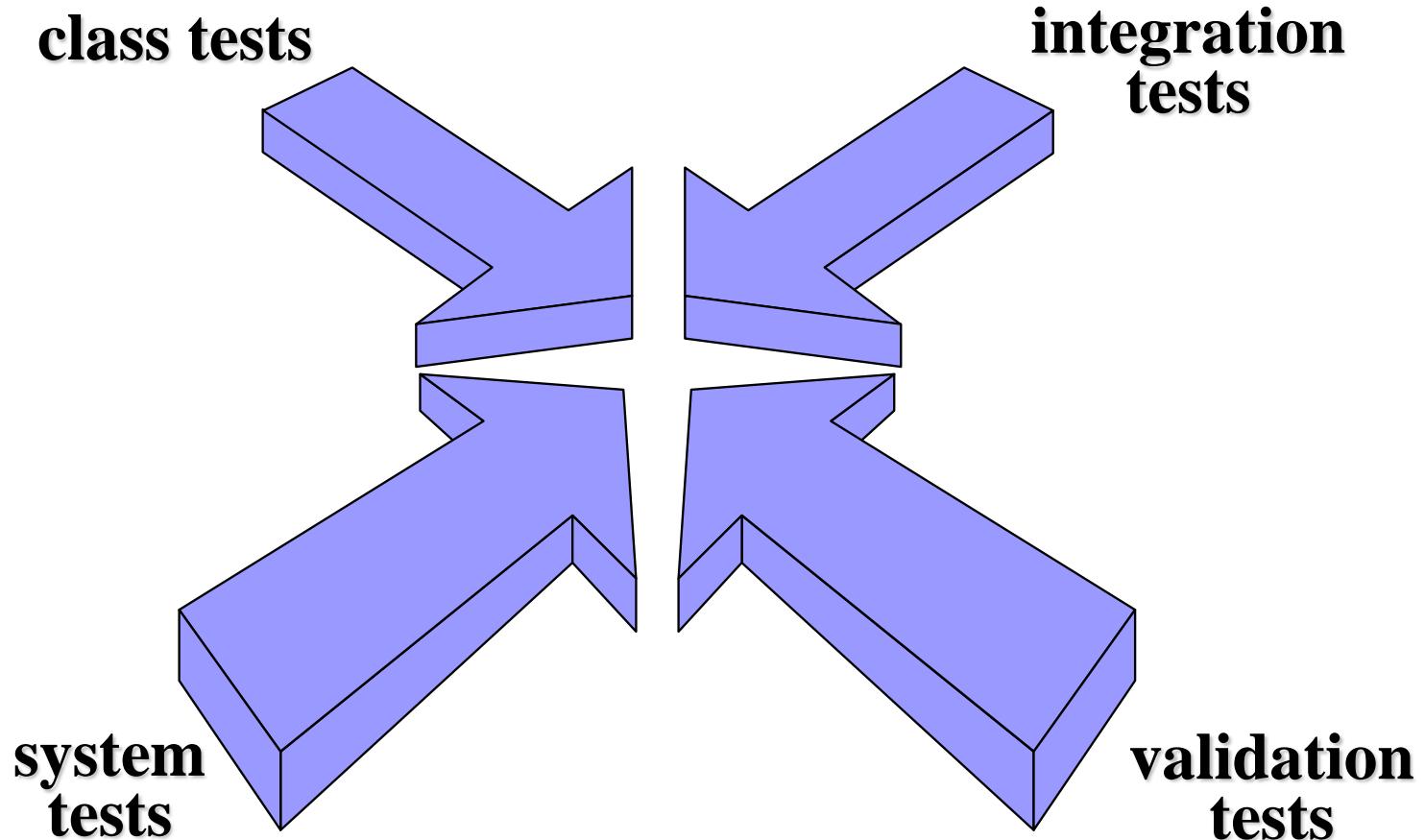
POLYMORPHISM

UNIT TESTING

ENCAPSULATION



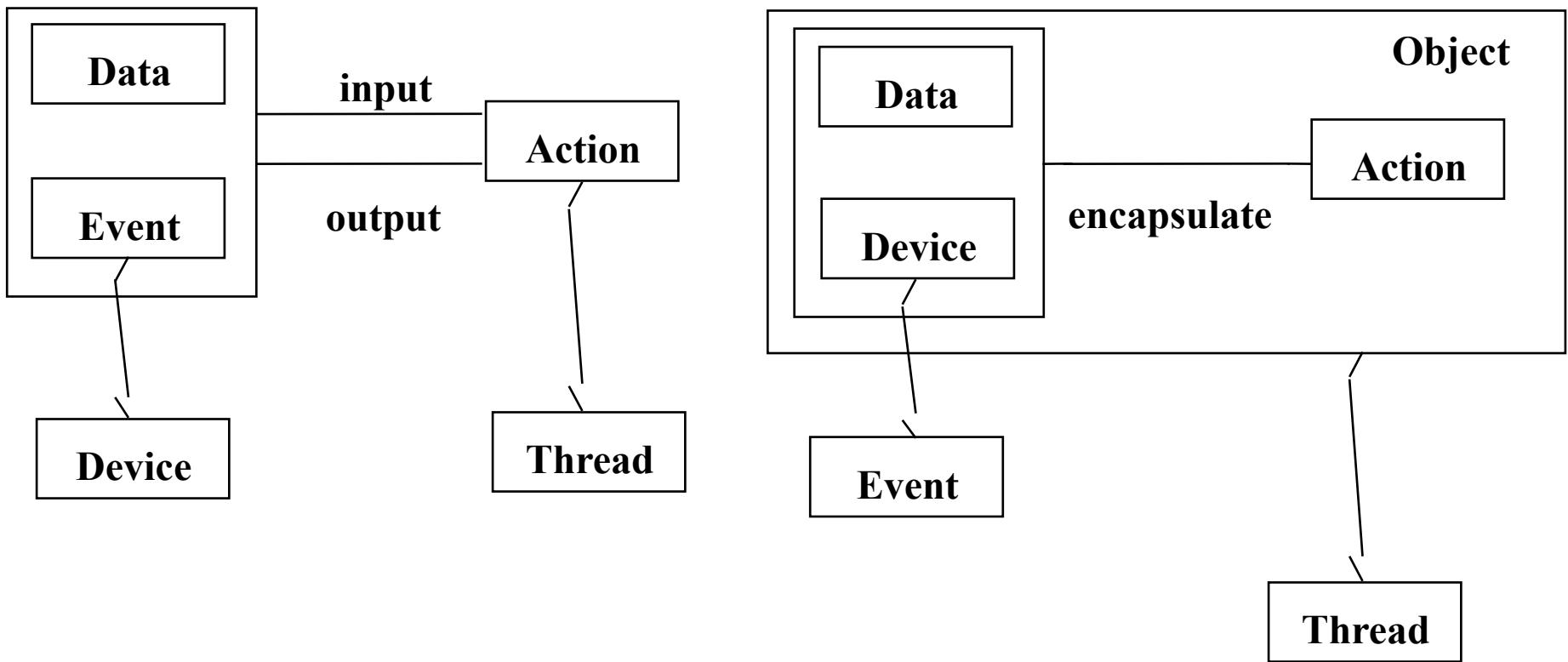
# Testing OO Code



# Effects on Testing

- OO software has potentially more severe testing problems than traditional software
- Objects don't really exist until they're instantiated
- Input/process/output paradigm is gone
- White Box testing can be more difficult
  - Path and line coverage
  - Code instrumentation

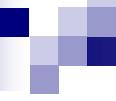
# Traditional vs. Object-Oriented Testing



**Object-orientation repackages the basis concepts  
(all relations are 0..n)**

# Object oriented testing

- System
  - Same as traditional
  - Still based on reqrs spec
- Unit
  - Two common structures used
    - Method\*
    - Class
  - Same as traditional(drivers & stubs)



# What causes the problems for testing?

- Encapsulation/Composition
- Inheritance
- Polymorphism

# Encapsulation: Definition

- The binding together of data members and member functions in *classes* (structures)
- Design decisions are hidden in the implementation
- “The world” sees only the interfaces
- Subclasses may inherit methods
- Real burden of testing is at the Integration level, no matter how good the Unit Testing is

# Encapsulation: Effects on testing (I)

- Changing the implementation has ripple effects
- Classes cannot be tested -- only instantiations of classes
- If a superclass changes, the subclasses must be retested

# Encapsulation: Effects on testing (II)

- Methods inherited from a superclass must be retested in their new context
- Two messages could be sent independently but concurrently, causing timing problems that can't be tested very well
- Classes may not get instantiated in the same order each time

# Inheritance: Definition

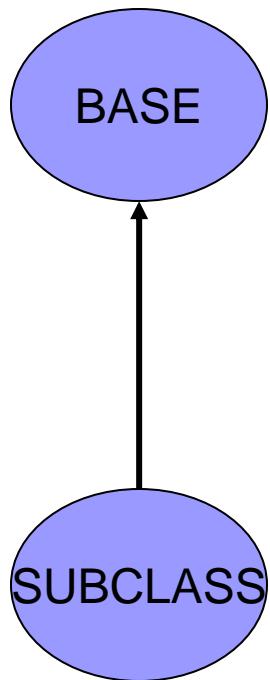
- New classes may be derived (inherited) from existing ones
- Notation: A *child class* of a *parent class*; a *base class* and its *derived class*
- Methods and variables may be inherited, even if not used
- Strict vs non-strict inheritance (Inheritance is called **strict** if descendants do **not** delete **or** modify (override) any **inherited** features, **non-strict** otherwise. Basically, if A and B are classes and A is the **strict** father of B then object of type B has the same behavior as A with more features.)
- Multiple and repeated inheritance (Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class D have a common parent A. D is then called a repeated descendant of A, and A a repeated ancestor of D)

# Inheritance: Effects on testing

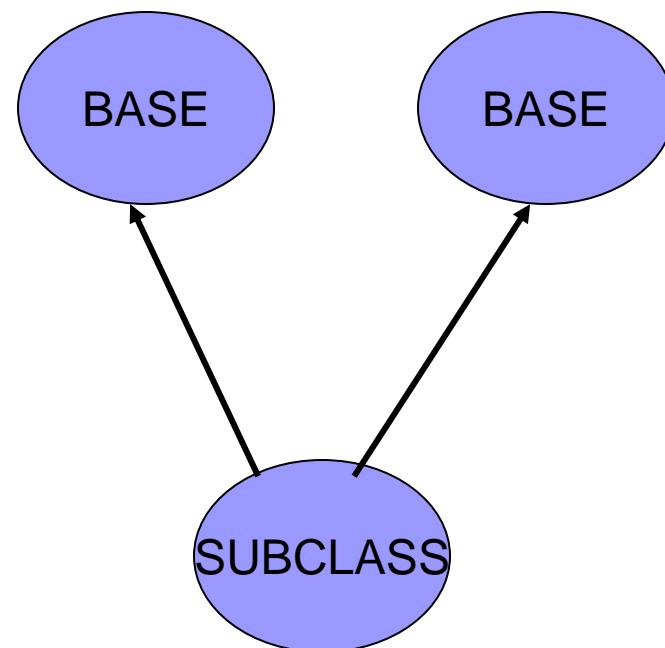
- Strict: parent-class tests should work on the subclass
- Non-strict: new tests are needed (methods may be renamed or not even present)
- Multiple and repeated: Much worse! The order of inheritance matters too
- Cannot test the validity of the inheritance itself

# INHERITANCE STRUCTURES

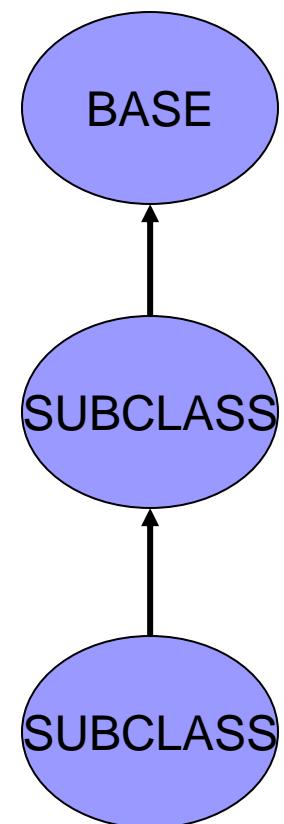
SINGLE

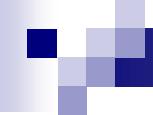


MULTIPLE



MULTIPLE LEVELS





# PERSON



# EMPLOYEE



MGR      GRUNT

If a method P takes Type Person as a parameter, then calling it with an Employee or a Grunt is allowed and must be tested

# Example

Class C has methods M and N, and method M calls method N. D inherits from C, overriding M; E inherits from D, overriding N. M for sure must be tested in all 3 contexts.

C M( ), N( )

↑

D overrides M( ), inherits N( )

↑

E inherits M( ), overrides N( )

# Polymorphism: Definition

- “Many forms”
- An object can be many types at once
- From before: A class has state and behavior, plus all the states and behaviors of every class it inherits from
- Methods and operations can be overridden

# Polymorphism: Effects on testing

- An inherited method that is overridden must be retested in the subclass
- If the implementation is changed, new white box tests are needed
- The reason for overriding almost certainly means new or different external behavior, so new black box tests are needed also

# Levels of Object-Oriented Testing

<u>Level</u>	<u>Item</u>	<u>Boundary</u>
Unit	<b>Method of an object?</b> <b>Class?</b>	<b>Program graph</b>
Integration	<b>MM-Path</b>	<b>Message quiescence</b>
	<b>Atomic System Function</b>	<b>Event quiescence</b>
System	<b>Thread</b>	<b>Source to sink ASF</b> (Atomic System Function ( <b>ASF</b> ))

Notice the cascading levels of interaction:

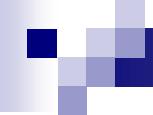
- Unit testing covers statement interaction,
- MM-Path testing covers method interaction,
- ASF testing covers MM-Path interaction,
- Thread testing covers object interaction, and all of this culminates in thread interaction.

# Two possible definitions of “unit”

- A *unit* is the smallest software component that can be compiled and executed
- A *unit* is a software component that would never be assigned to more than one developer
- The main question for testing a class is whether the class itself, or its methods, are considered as units
- Best choice: First bullet – a *class* is a *unit*

# More on Units

- If we consider methods as object-oriented units, then we must provide stub classes and a driver or “main program” class
- If we consider classes as units, other problems arise:
  - In a static view, inheritance is ignored
  - Classes have to be instantiated to be tested
  - Abstract classes cannot be tested
  - If this class inherits, all its parents are needed



# Class Testing

# Class Testing

- Main question: Are units methods or are units classes?
- If methods are units, testing is like traditional software.
- Classes as units makes more sense when there is little inheritance and a lot of complexity in the class

# Classes as Units

- Static view : Classes are simply code. This is fine for code reading, but inheritance is ignored, so “flattened” classes are needed
- Compile-time view: Inheritance occurs, but no testing
- Execution-time view: Instantiation occurs – this is where testing happens, of course
- In any case, Abstract Classes cannot be tested.

# Challenges of Class Testing

- Encapsulation:
  - Difficult to obtain a snapshot of a class without building extra methods which display the classes' state
- Inheritance:
  - Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
  - Other unaltered methods within the subclass may use the redefined method and need to be tested
- White box tests:
  - Basis path, condition, data flow and loop tests can all be applied to individual methods within a class but they don't test interactions between methods

# Saturn Windshield Wiper Controller

The windshield wiper on the Saturn automobiles is controlled by a lever with a dial. The lever has four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

*For Intermittent setting, there are 3 dial sub settings*

- 1: wiper wipes 4 times per minute
- 2: wiper wipes 6 times per minute
- 3: wiper wipes 12 times per minute

*For Low, the wiper wipes 30 times per minute*

*For High, the wiper wipes 60 times per minutes*

Lever Dial	OFF n/a	INT 1	INT 2	INT 3	LOW n/a	HIGH n/a
Wiper	0	4	6	12	30	60

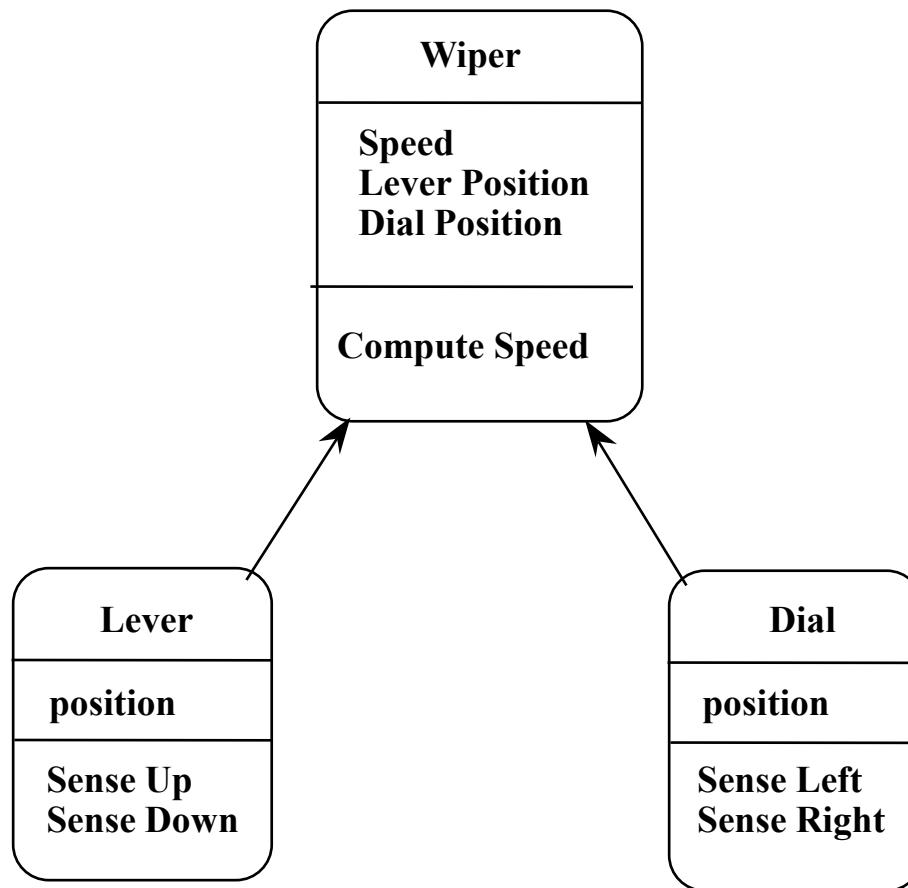
## Input Event Description

- ie1 lever from OFF to INT
- ie2 lever from INT to LOW
- ie3 lever from LOW to HIGH
- ie4 lever from HIGH to LOW
- ie5 lever from LOW to INT
- ie6 lever from INT to OFF
- ie7 dial from 1 to 2
- ie8 dial from 2 to 3
- ie9 dial from 3 to 2
- ie10 dial from 2 to 1

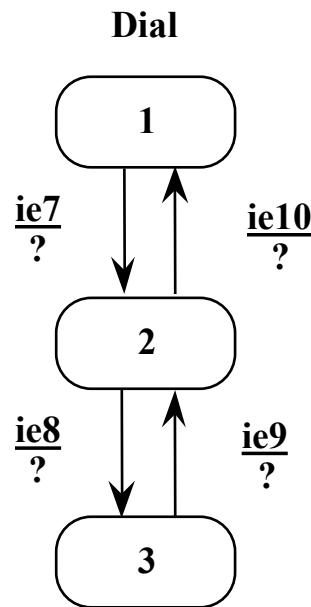
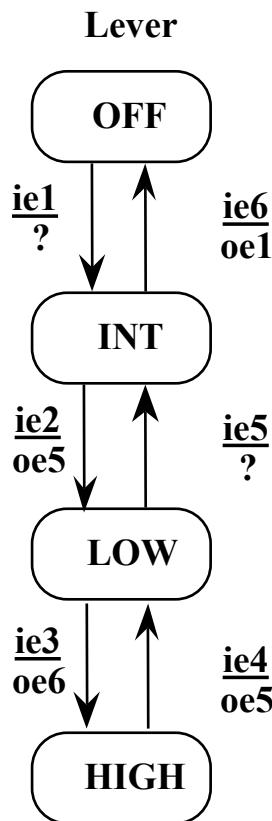
## Output Event

- oe1 0 w.p.m.
- oe2 4 w.p.m.
- oe3 6 w.p.m.
- oe4 12 w.p.m.
- oe5 30 w.p.m.
- oe6 60 w.p.m.

# Saturn Windshield Wiper Objects



# Saturn Windshield Wiper Object FSMs

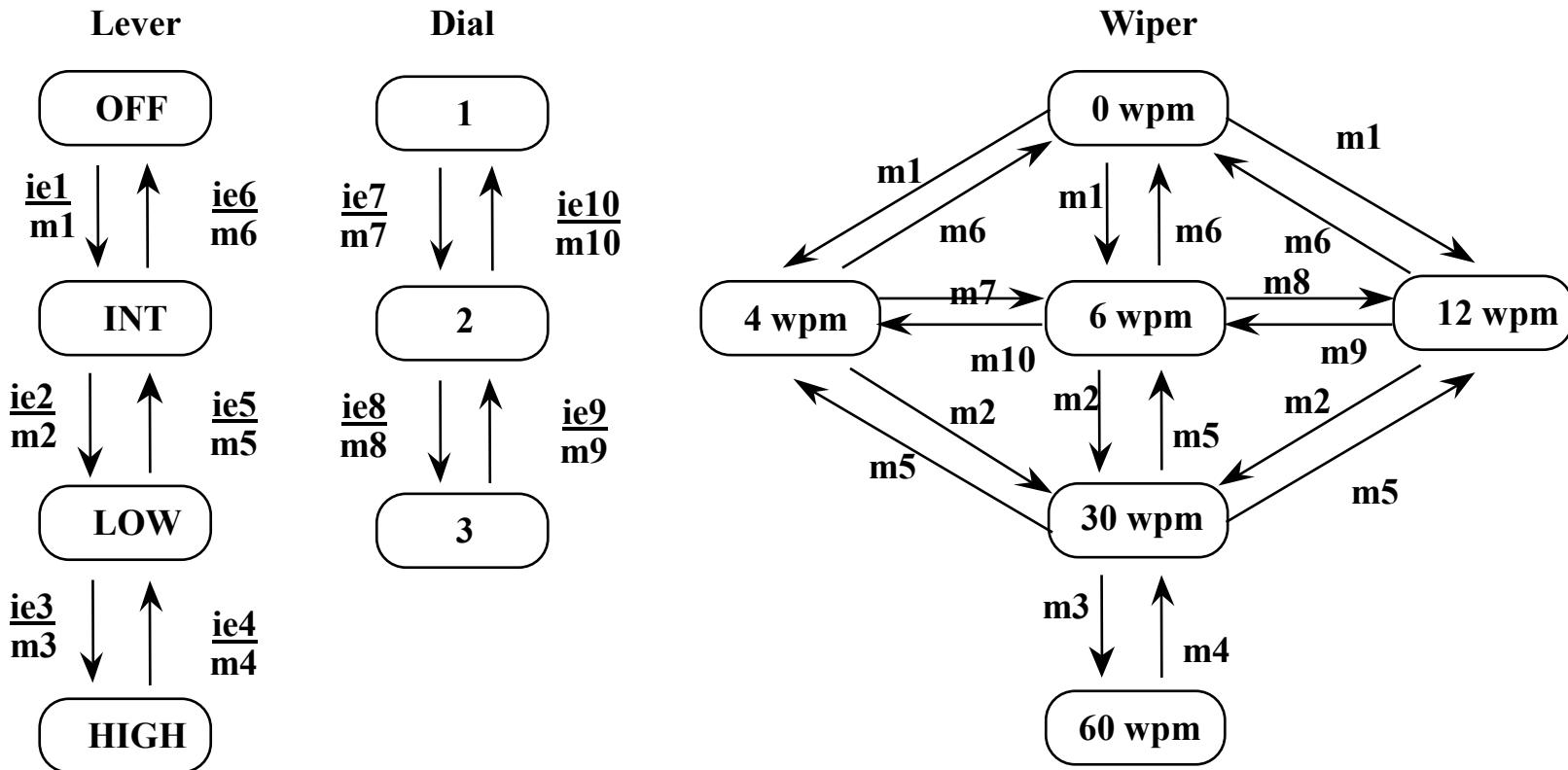


<b>Input Event</b>	<b>Description</b>
ie1	lever from OFF to INT
ie2	lever from INT to LOW
ie3	lever from LOW to HIGH
ie4	lever from HIGH to LOW
ie5	lever from LOW to INT
ie6	lever from INT to OFF
ie7	dial from 1 to 2
ie8	dial from 2 to 3
ie9	dial from 3 to 2
ie10	dial from 2 to 1

<b>Output Event Description</b>	
oe1	0 w.p.m.
oe2	4 w.p.m.
oe3	6 w.p.m.
oe4	12 w.p.m.
oe5	30 w.p.m.
oe6	60 w.p.m.

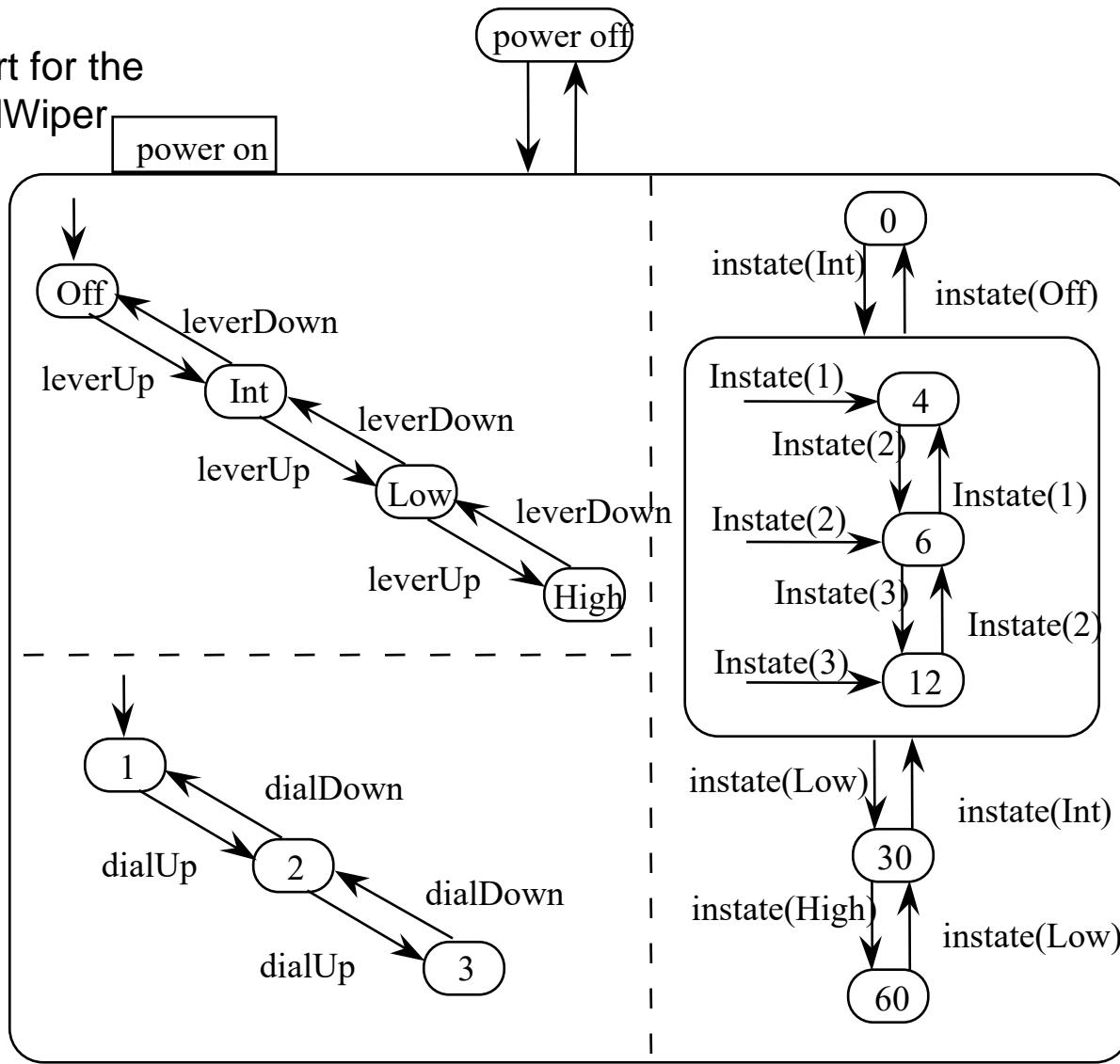
Note that several transition actions are indeterminate because of composition.

# Resolving the Indeterminacy with Messages



**Messages m1 to m6 inform the Wiper object of the state of the Lever object.**  
**Messages m7 to m10 inform the Wiper object of the state of the Dial object.**

## State Chart for the windshieldWiper class



# State Chart Metrics

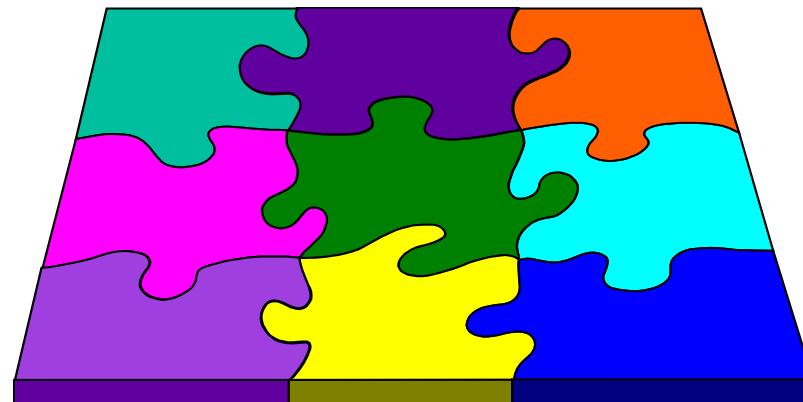
- State Charts like the preceding lend themselves to test metrics:
  - Test every event
  - Test every state in a component
  - Test every transition in a component
  - Test all pairs of interacting states between components
  - System Testing: Use scenarios/use cases



# Object-oriented Integration Testing

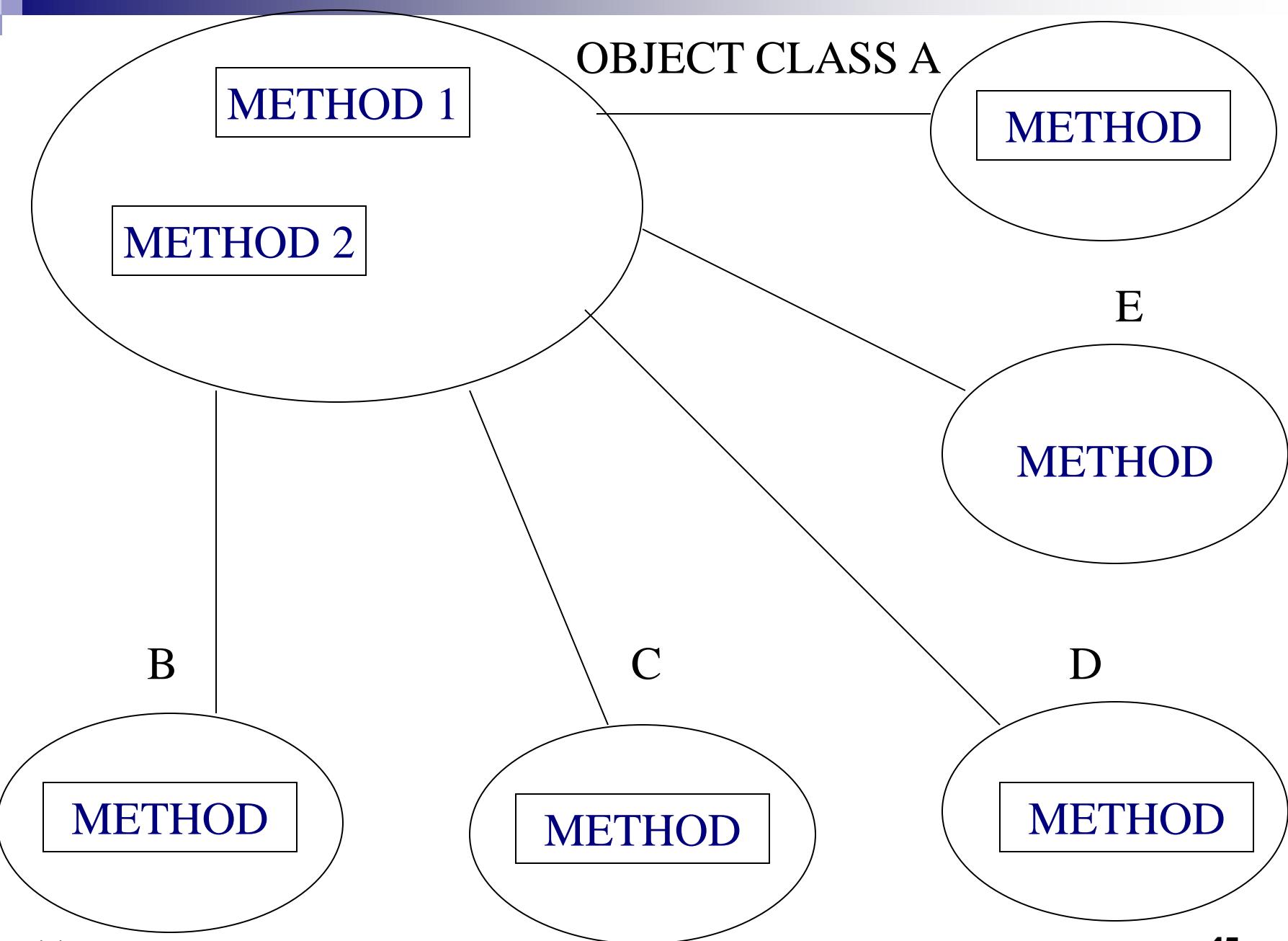
# Integration Testing

- OO does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning
- Integration applied three different incremental strategies
  - Thread-based testing: integrates classes required to respond to one input or event
  - Use-based testing: integrates classes required by one use case
  - Cluster testing: integrates classes required to demonstrate one collaboration



# OO Integration Testing

- As noted before, Integration Testing is *the* major issue of object-oriented testing
- If a method is a unit, then two levels of integration are needed:
  - Integrating the method/operation into the class
  - Integrating the class with other classes
- If a class is a unit, any flattened classes must be restored and any test code removed



F

OBJECT CLASS A

METHOD

METHOD 2

E

METHOD

B

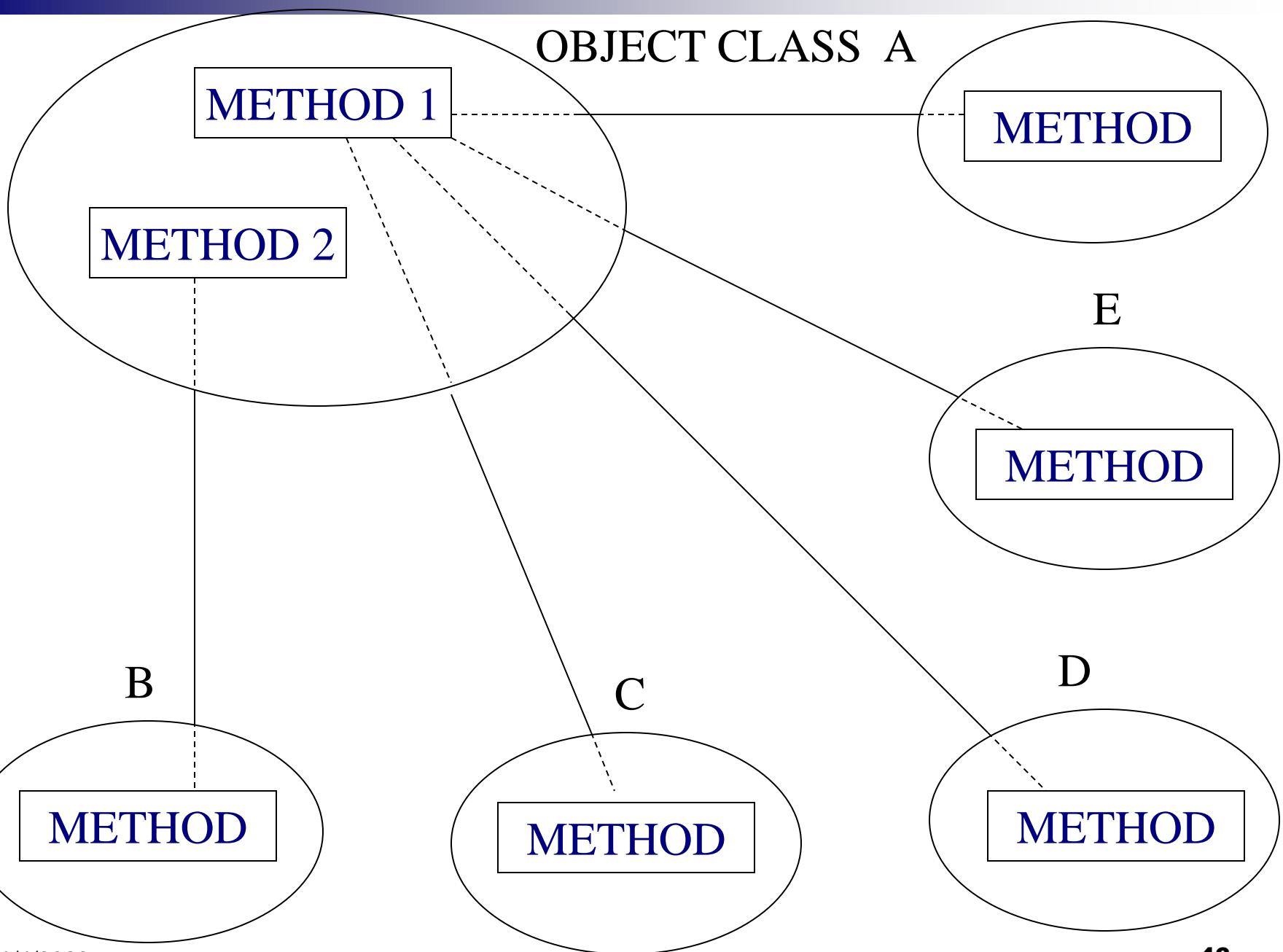
C

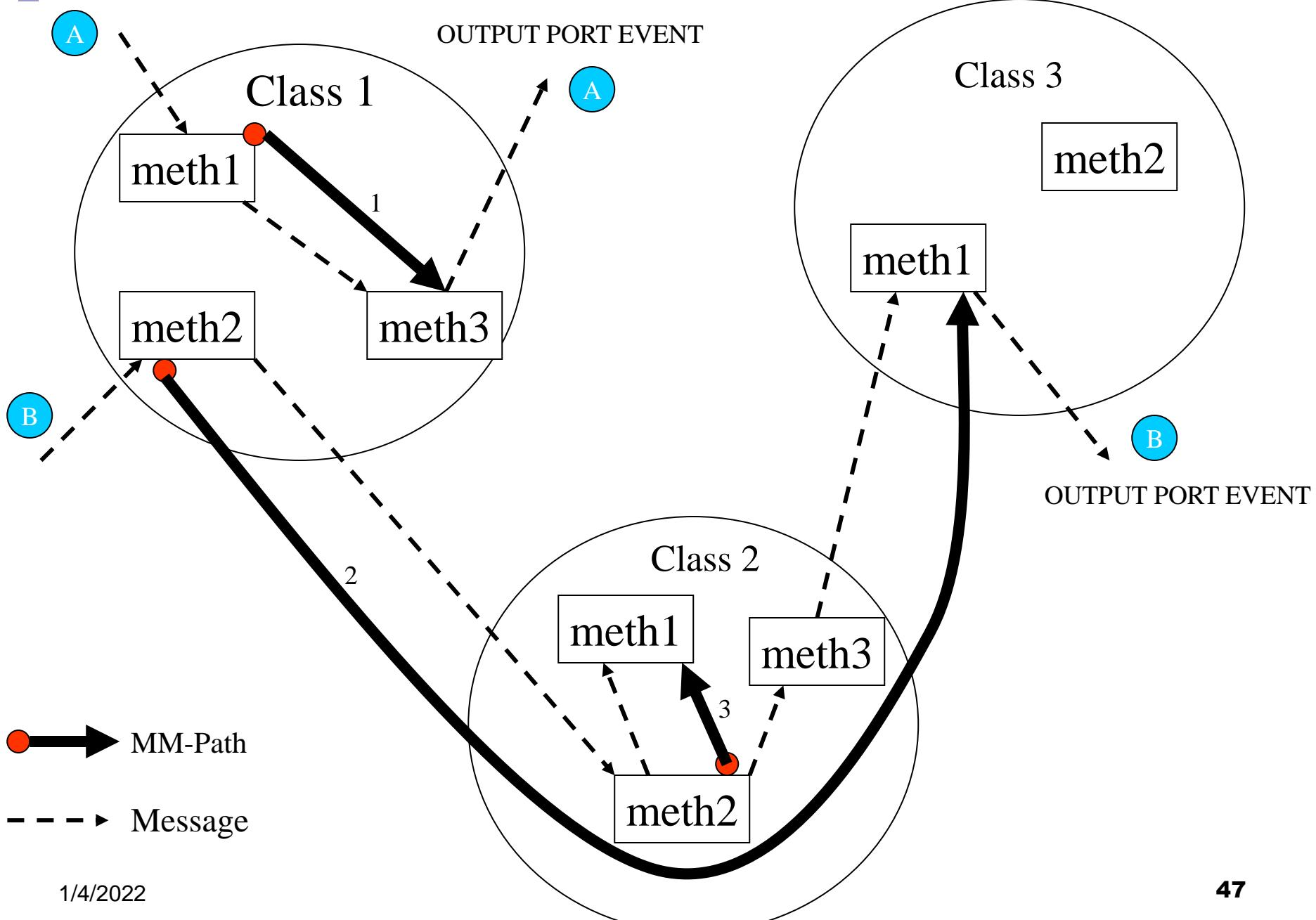
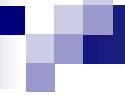
D

METHOD

METHOD

METHOD





# Integration Testing of Object-Oriented Software

- **Definition in UML**
- **Two kinds of o-o software:**
  - **data-driven**
  - **event-driven**
- **How are these described (for a tester)?**
- **What is an o-o unit:**
  - **a class?**
  - **a method?**
- **What is the basis for integration testing?  
(next slide)**

# UML Usage

- Basis for Integration Testing: collaboration and sequence diagrams
- Collaboration diagrams show the message traffic among classes
- Collaboration diagrams support both pairwise integration and neighborhood integration

## First Example (data-driven)

**The o-oCalendar program is an object-oriented implementation of the NextDate function.  
 $(\text{NextDate}(\text{Mar., 5, 2002})) = \text{Mar., 6, 2002.}$**

**When this is implemented in procedural code, it is approximately 50 lines long.**

**A "pure" (i.e., good practice) object-oriented implementation contains one abstract class and five classes. The next few slides show lead to the Collaboration Diagram.**

# Classes

testIt

**CalendarUnit** 'abstract class  
**currentPos** As Integer  
**CalendarUnit( pCurrentPos)**  
**setCurrentPos( pCurrentPos)**  
**increment()** 'boolean

**Month**

**private Year y**  
**private sizeIndex =**  
**<31, 28, 31, 30, 31, 30, 31, 31,**  
**30, 31, 30, 31>**  
  
**Month(pcur, Year pYear)**  
**setCurrentPos(pCurrentPos)**  
**setMonth(pcur, Year pYear)**  
**getMonth()**  
**getMonthSize()**  
**increment()**

**Day**

**Month m**

**Day( pDay, Month pMonth)**  
**setCurrentPos( pCurrentPos)**  
**setDay( pDay, Month pMonth)**  
**getDay()**  
**increment()**

**Date**

**Day d**  
**Month m**  
**Year y**

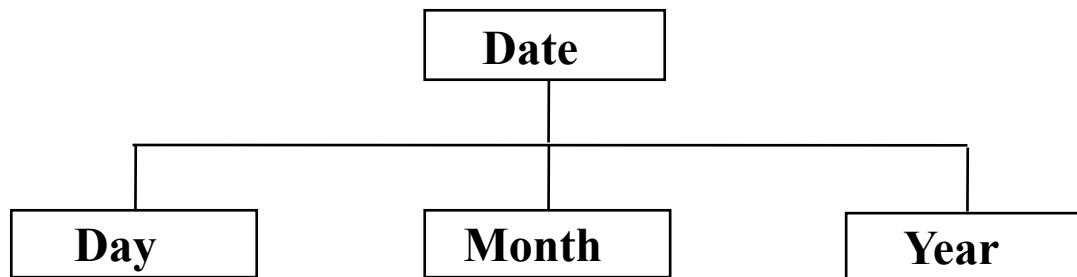
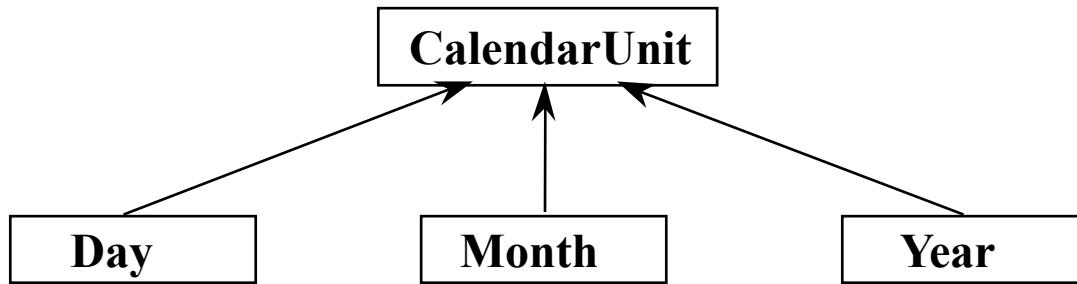
**Date( pDay, pMonth,, pYear)**  
**increment()**  
**printDate()**

**Year**

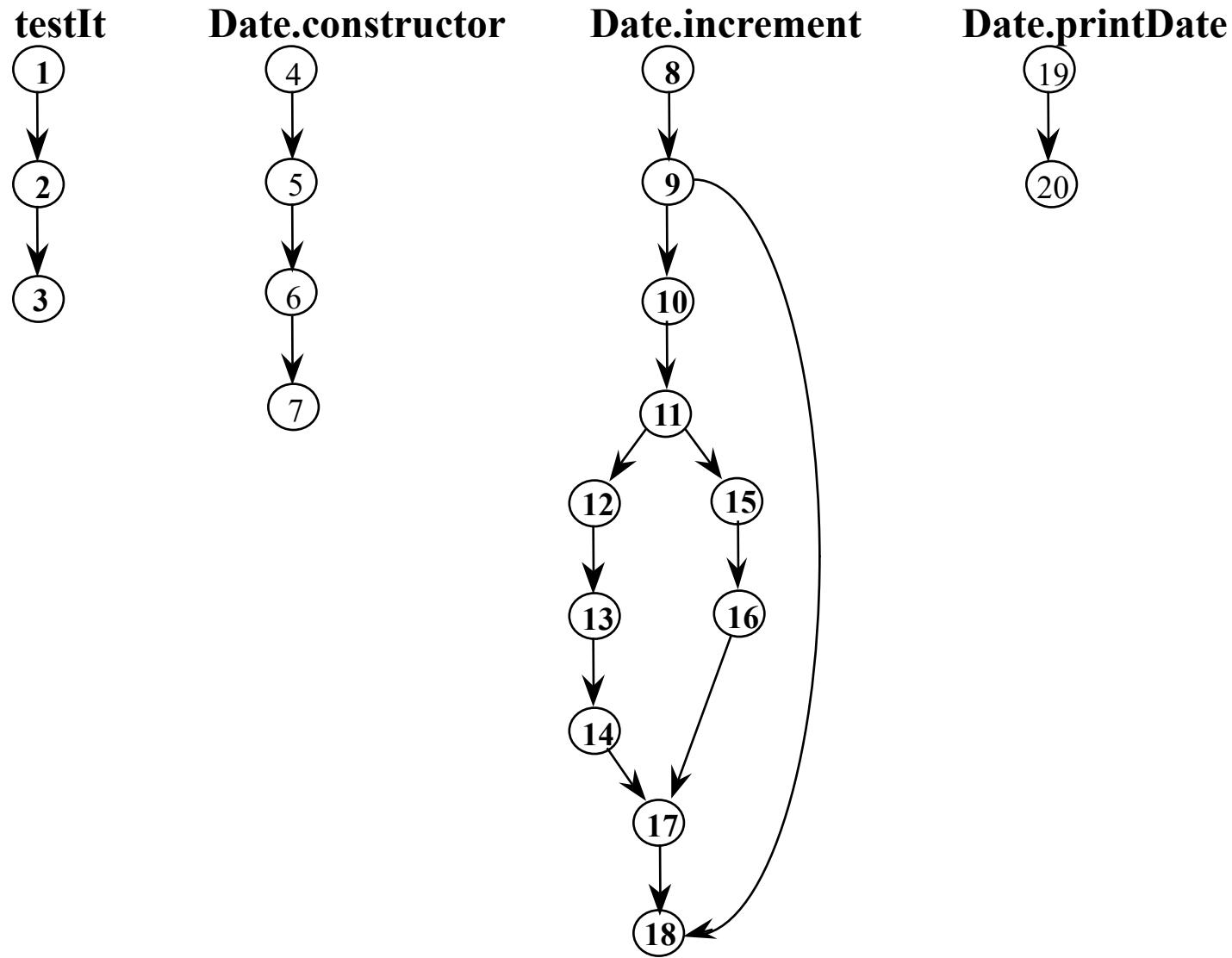
**Year(int pYear)**

**setCurrentPos( pCurrentPos)**  
**getYear()**  
**increment()**  
**isleap()** 'boolean

# Inheritance and Aggregation

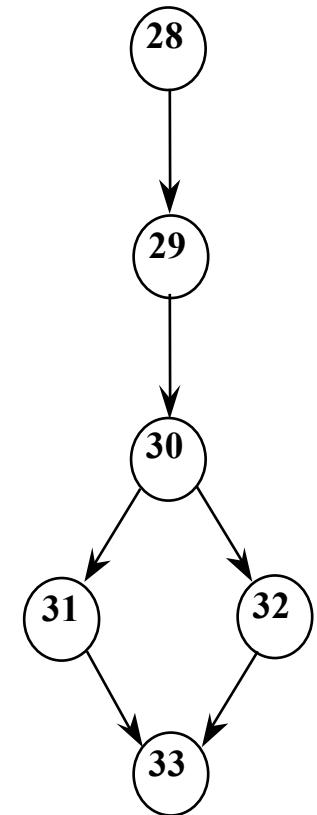
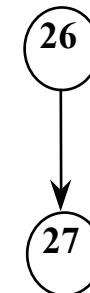
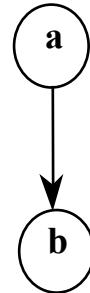


# Program Graphs of testIt and Date Classes



# Program Graphs of Day Class

**Day.constructor**   **Day.setCurrentPos**   **Day.setDay**   **Day.getDay**   **Day.increment**

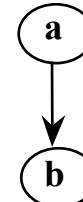


# Program Graphs for the Month Class

**Month.constructor**



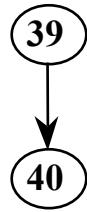
**Month.setCurrentPos**



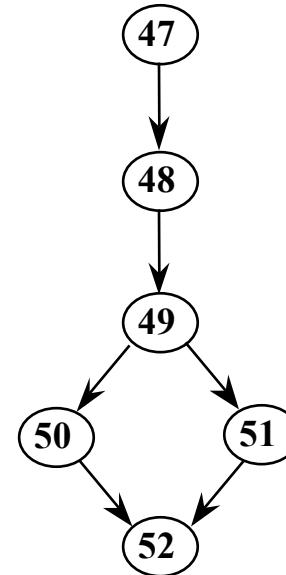
**Month.setMonth**



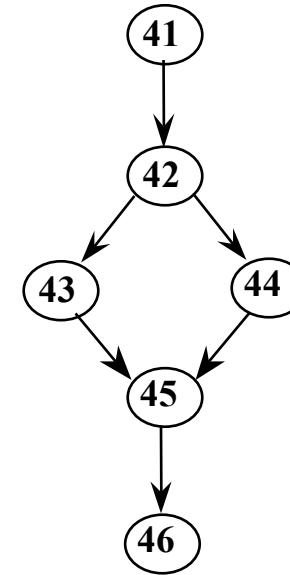
**Month.getMonth**



**Month.increment**

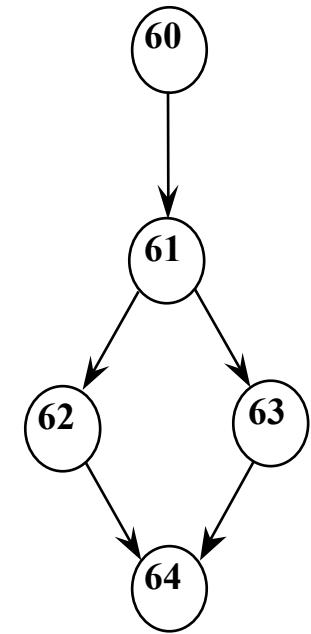
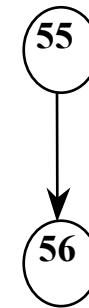
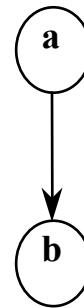
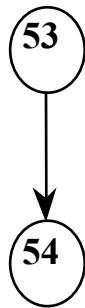


**Month.getMonthSize**

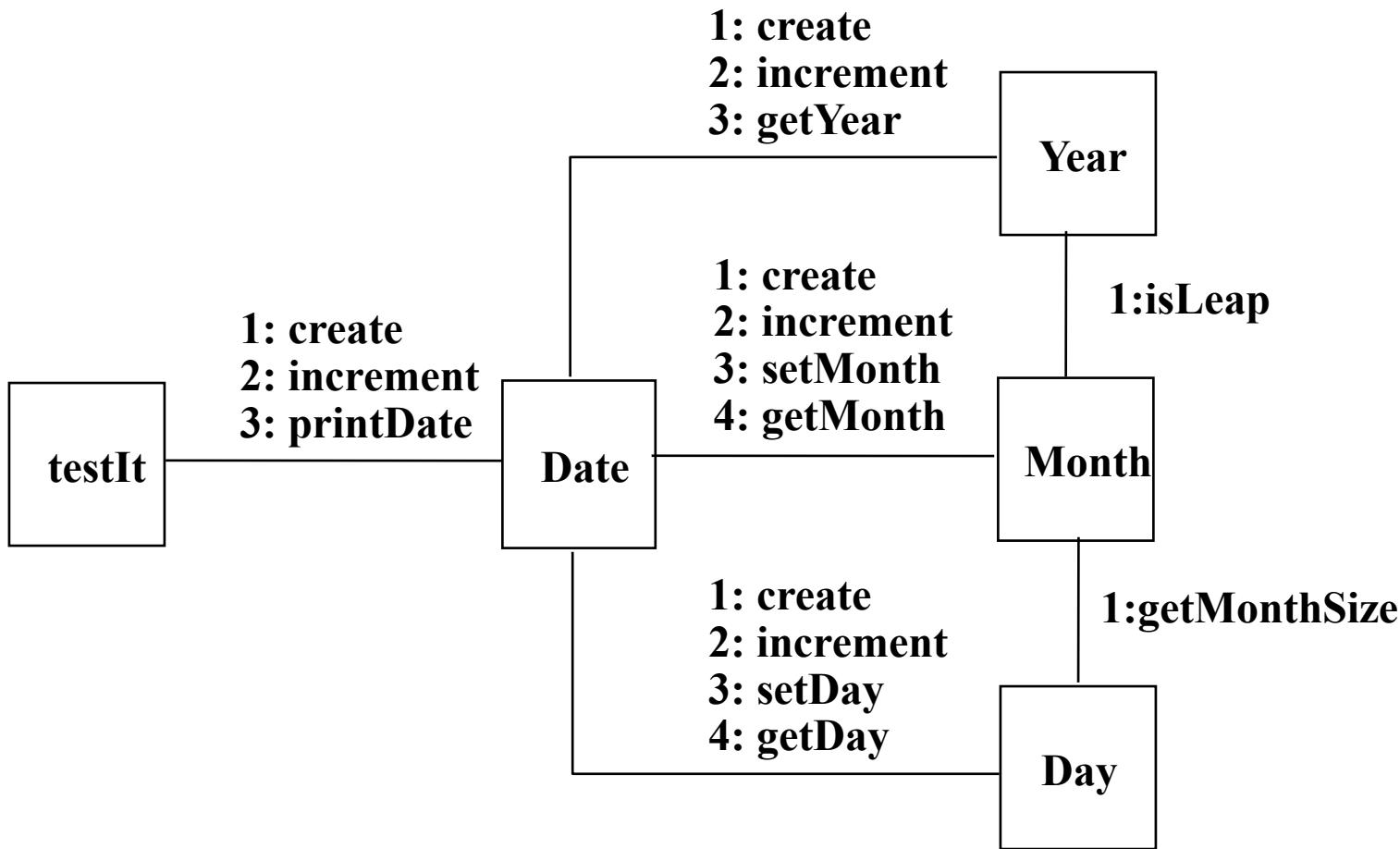


# Program Graphs of Year Methods

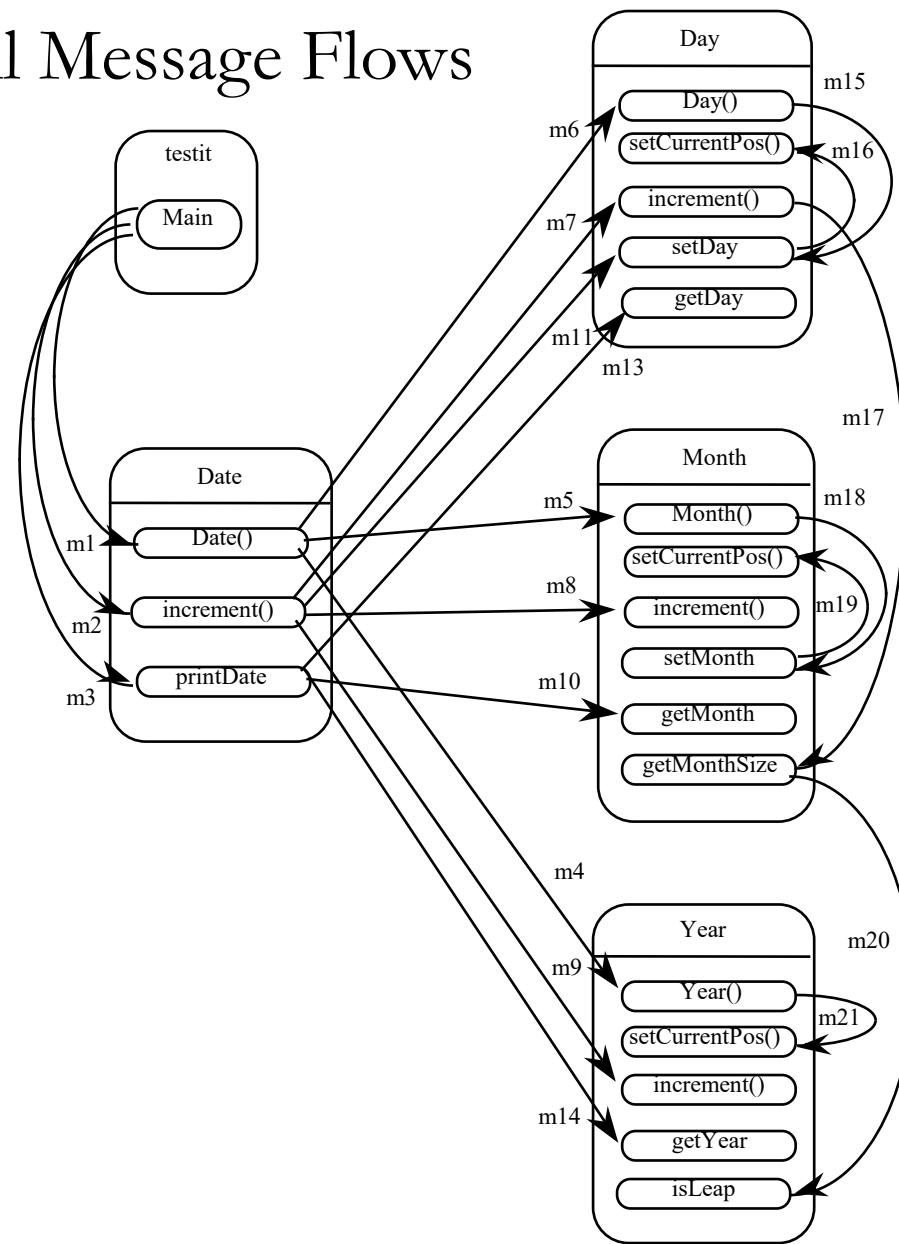
**Year.constructor**   **Year.setCurrentPos**   **Year.getYear**   **Year.increment**   **Year.isLeap**



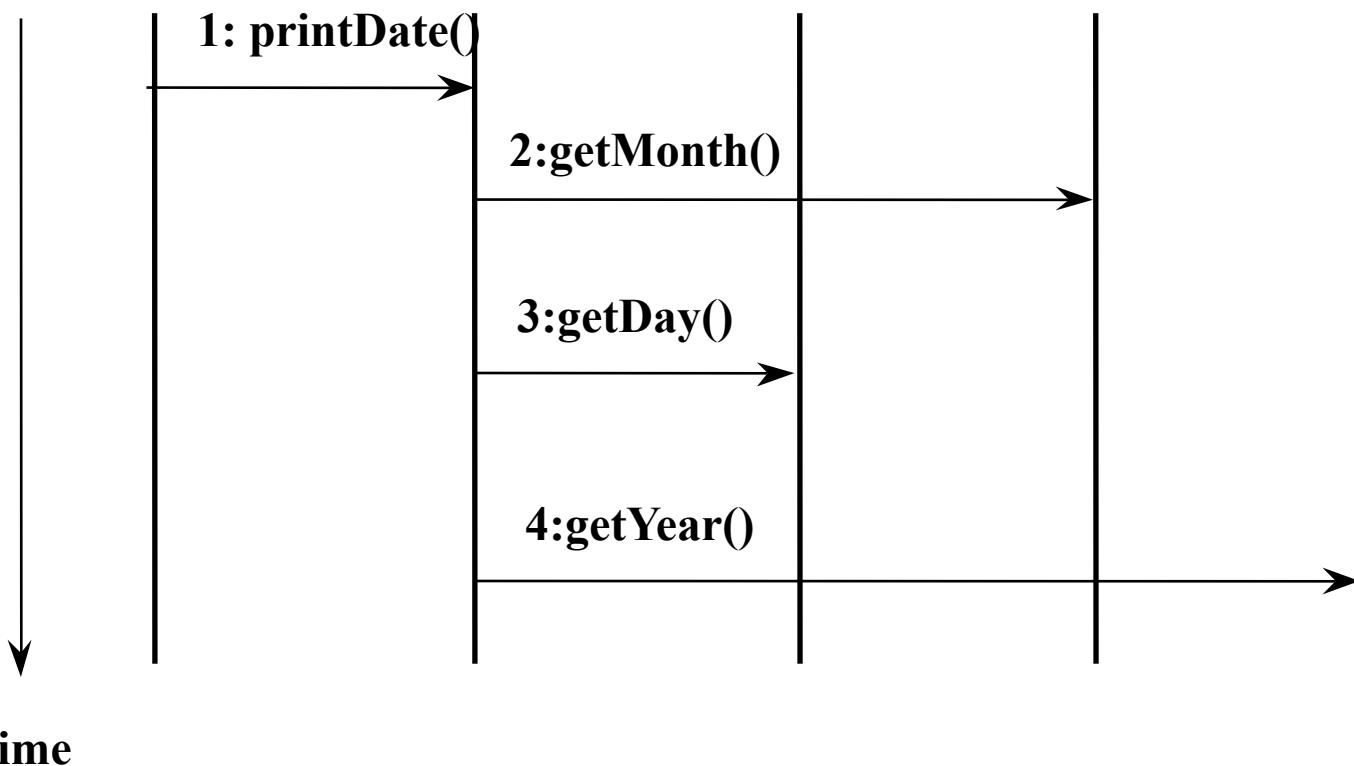
# Collaboration Diagram



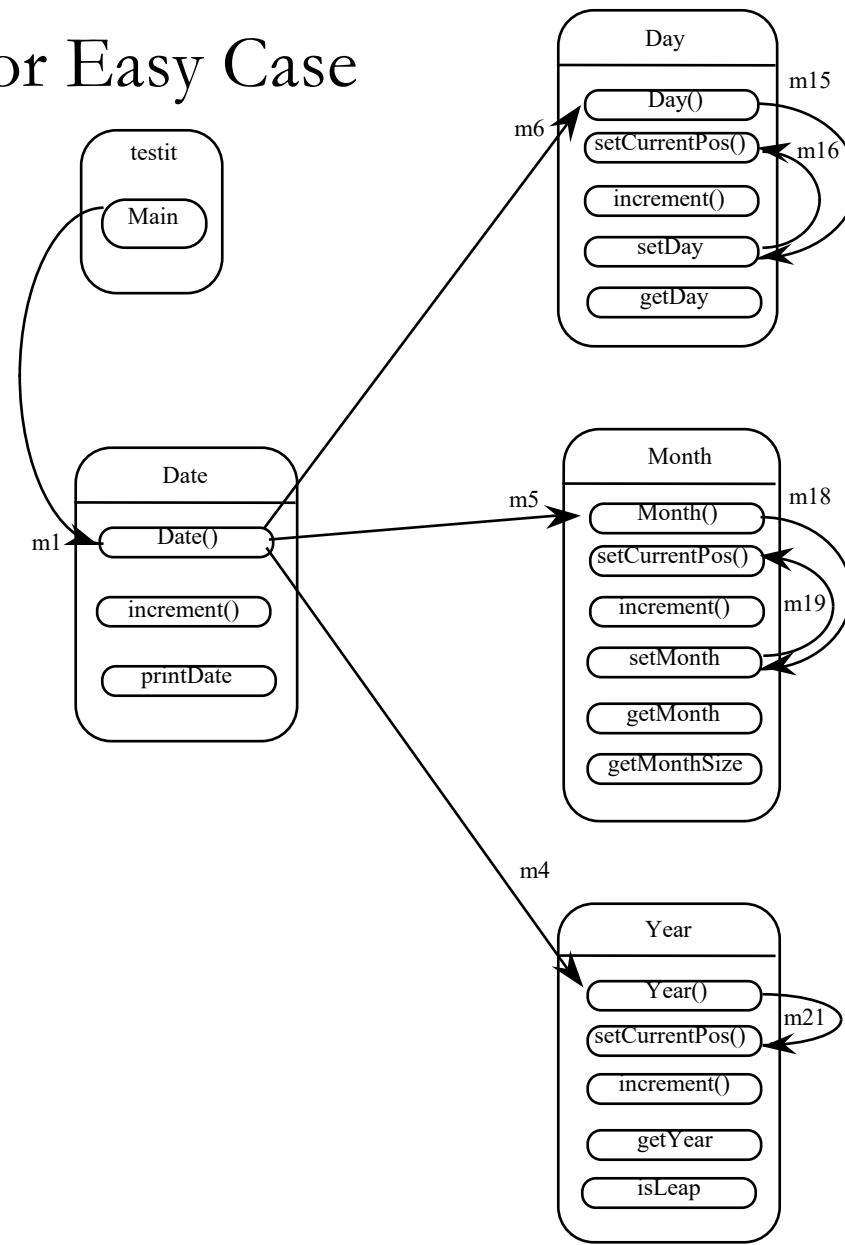
# All Message Flows



# Message Sequence Diagram (for an easy test case)



# Message Flow for Easy Case



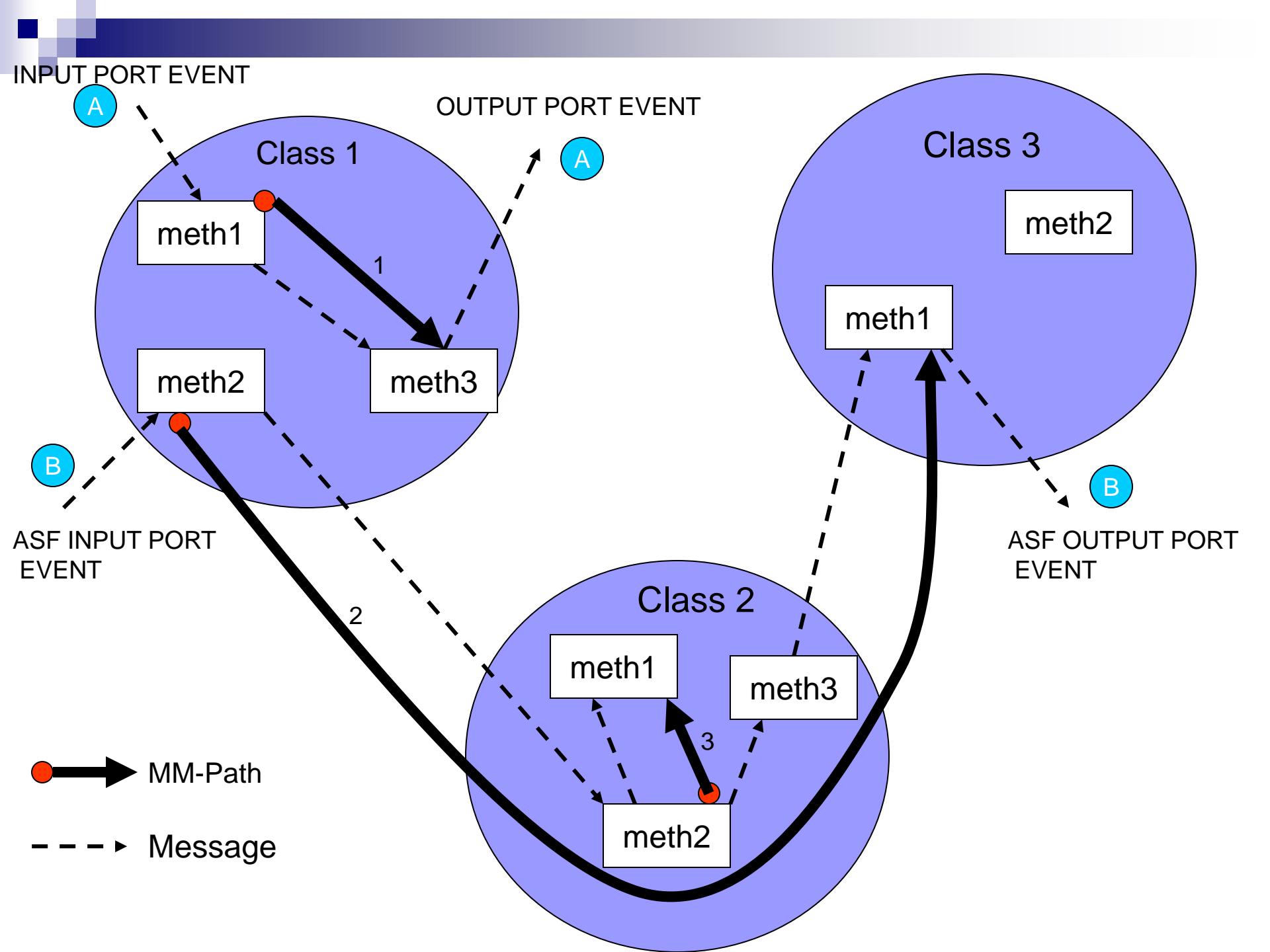
# CONSTRUCT DEFINITIONS

- MM-path (method message - path) [message quiescence]
  - Sequence of executions linked by messages.
  - Starts with method and ends with a method that doesn't produce a message

# CONSTRUCT DEFINITIONS

## ■ ASF (atomic system function) [event quiescence]

- Represents an input event
- Followed by a set of mm-paths
- Terminated by an ouput event





# Object-oriented System Testing

# System Testing

- True system testing does not require knowing whether the implementation is procedural or object-oriented
- The goal is to identify threads
- Procedural implementation: We used behavioral models
- OO implementation: UML use cases – high-level, essential, and expanded-essential

## Currency Converter UML Description

The currency converter application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Community euros, and Japanese yen. The user can revise inputs and perform repeated currency conversion.

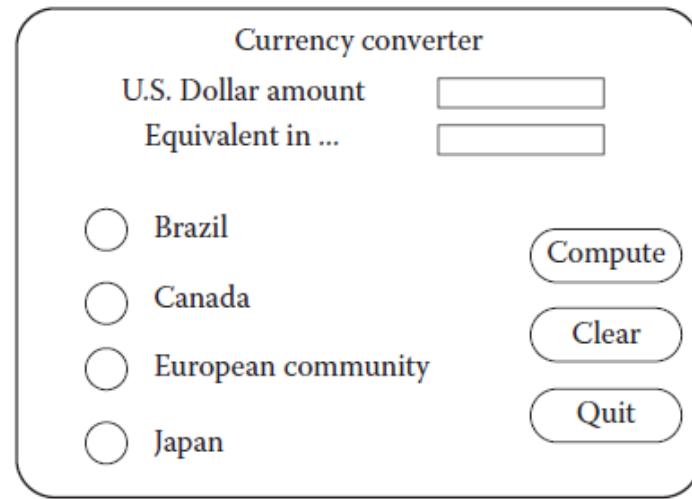
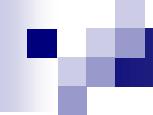


Figure 15.18 Currency converter GUI.

# Conclusion

- Class/Objects are MUCH more complicated than procedures
  - Need to consider encapsulation, inheritance and polymorphism
  - OO Design proceeds by composition instead of decomposition
- Encapsulation together with Inheritance, which intuitively ought to bring a reduction in testing problems, compounds them instead.
- Complexity is moved from methods to messaging among objects
- How can code be reused if it is not possible to demonstrate its quality?
- Above all, if at all possible, avoid multiple and repeated inheritance

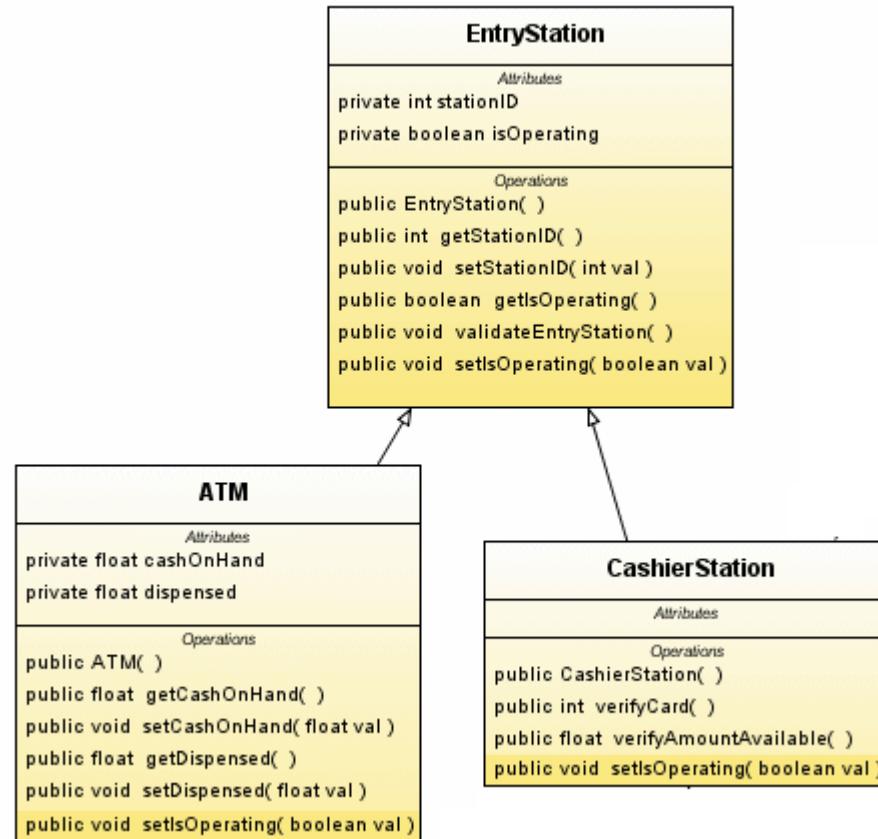


# Flattening the Class

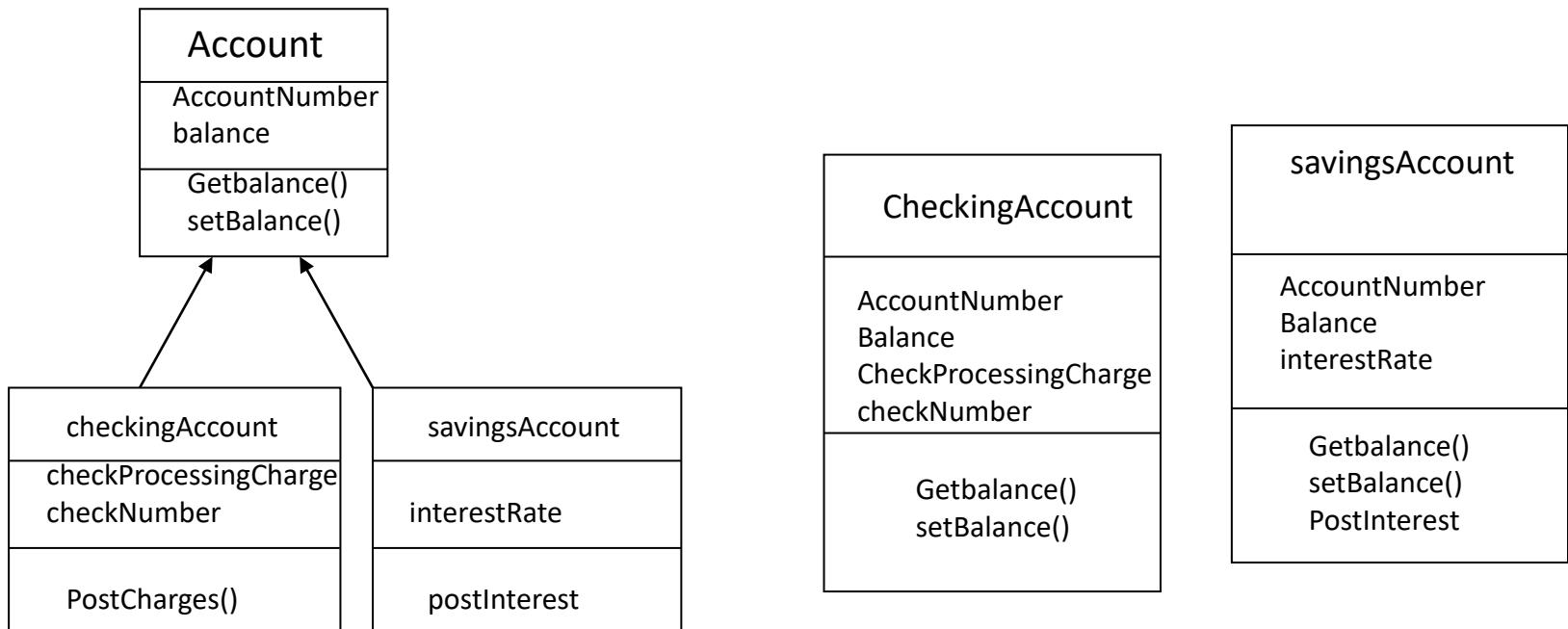
# Inheritance issue

- Inheritance is an important part of the object oriented paradigm
- Unit testing a class with a super class can be impossible to do without the super classes methods/variables

# Example1



# Issues in Object-Oriented Testing



Inheriting classes: we cannot test with unit testing (balance cannot be accessed)

Flattened classes: we test the same function several times

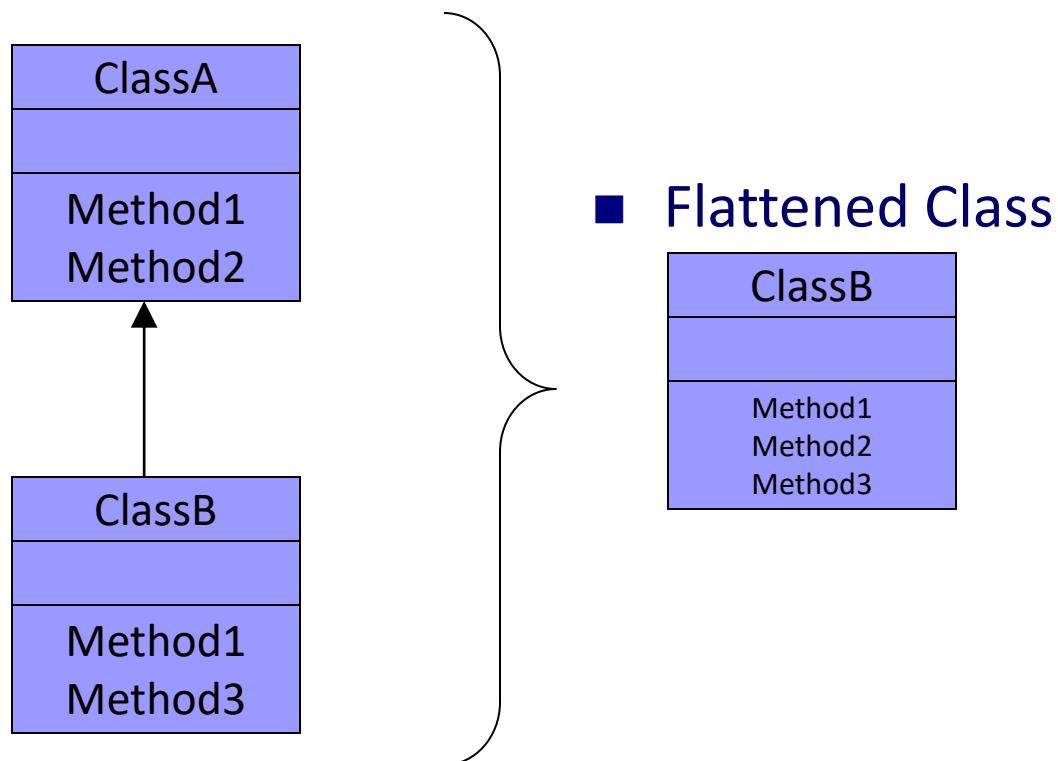
# Problem

```
1 public class Fruit {  
2  
3     private int weight;  
4  
5     public Fruit() {  
6         //Grows a piece of fruit  
7     }  
8  
9     public void prepare() {  
10        //Wash Fruit  
11    }  
12  
13    public void eat() {  
14        //Consume the piece of fruit  
15    }  
16  
17 }  
18  
19 public class Orange extends Fruit {  
20  
21     private int noOfSegments;  
22  
23     public void prepare() {  
24         //Peel Fruit  
25         peel();  
26     }  
27  
28     private void peel() {  
29         //Peel orange  
30     }  
31  
32 };
```

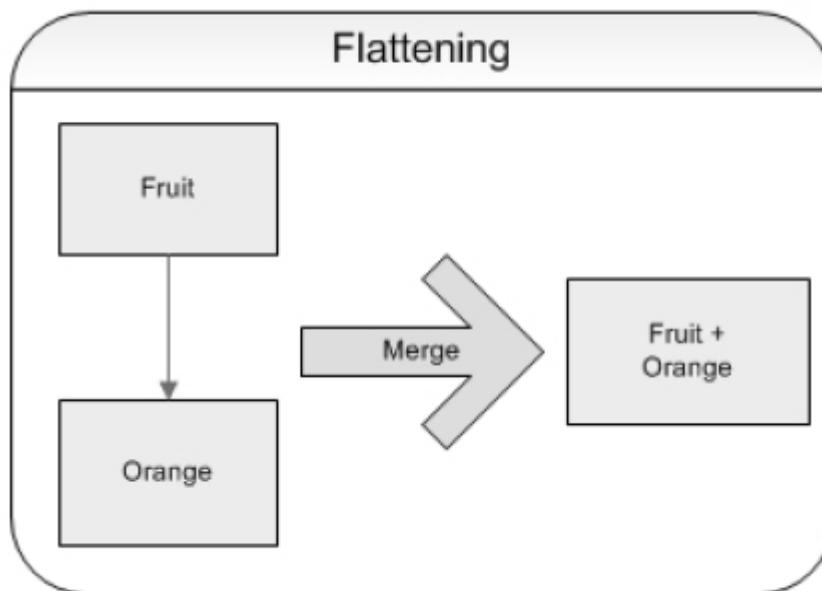
# One Solution - Flattening

- Merge the super class, and the class under test so all methods/variables are available
  - Solves initial unit test problems
- Problems:
  - The class won't be flattened in the final product so potential issues may still arise
  - Complicated when dealing with multiple inheritance

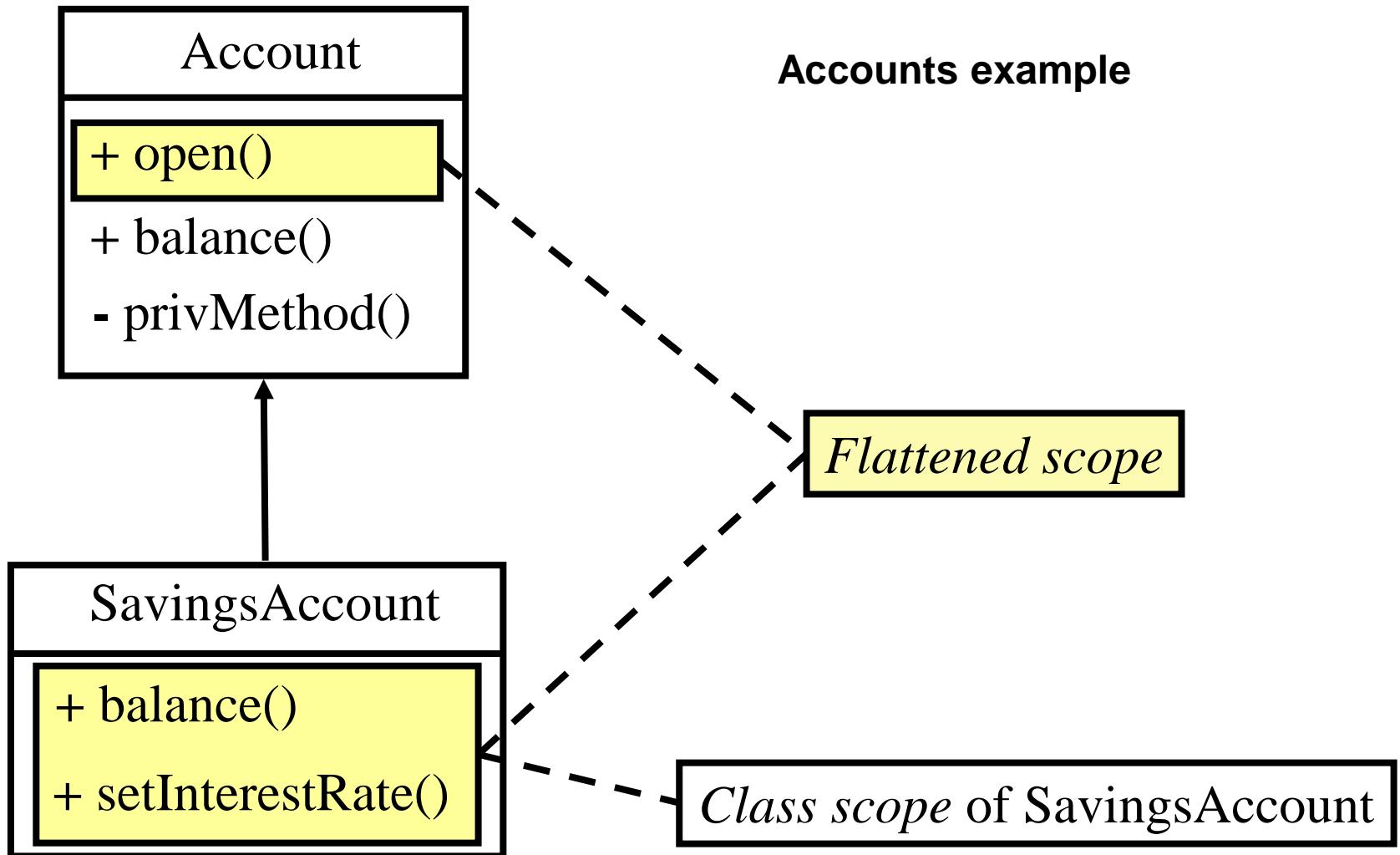
# Flattening sample solution



# Orange problem



```
1 public class Orange{  
2  
3     private int noOfSegments;  
4     private int weight;  
5  
6     public Orange() {  
7         //Grows a piece of fruit  
8     }  
9  
10    public void prepare() {  
11        //Peel Fruit  
12        peel();  
13    }  
14  
15    private void peel() {  
16        //Peel orange  
17    }  
18  
19    public void eat() {  
20        //Consume the piece of fruit  
21    }  
22  
23};
```



# Polymorphism Issues

- Repeatedly testing same methods
- Time can then be wasted if not addressed
- Potentially can be avoided, and actually save time

# Polymorphism example

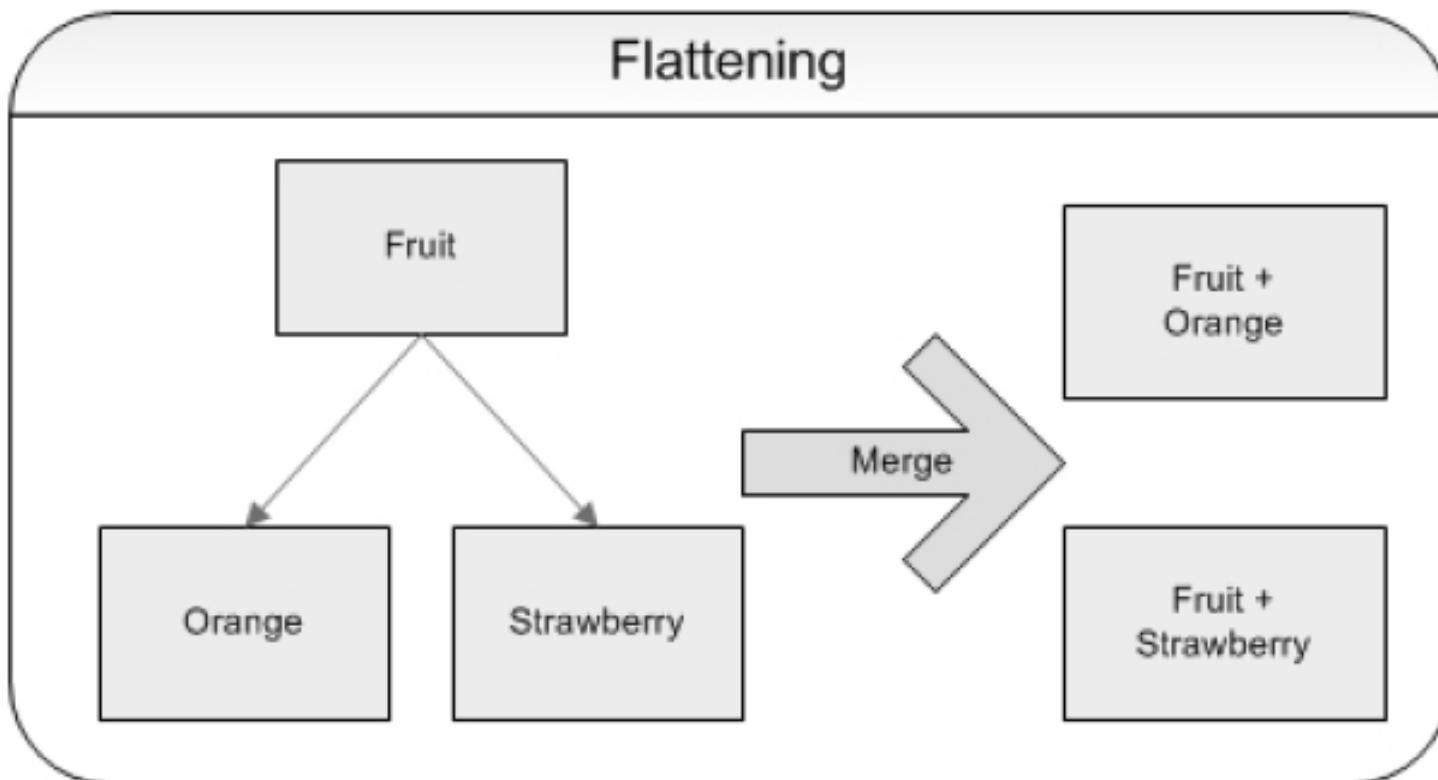
```
1 public class Fruit {  
2  
3     private int weight;  
4     public Fruit() {  
5         //Grows a piece of fruit  
6     }  
7     public void prepare() {  
8         //Wash fruit  
9     }  
10    public void eat() {  
11        //Consume the piece of fruit  
12    }  
13}  
14 }
```

```
16   public class Orange extends Fruit {  
17  
18       private int noOfSegments;  
19       public void prepare() {  
20           //Peel fruit  
21           peel();  
22       }  
23       public void peel() {  
24           //Peel orange  
25       }  
26   };  
27  
28  
29   public class Strawberry extends Fruit {  
30  
31       private int redness;  
32       //Returns how red the strawberry is.  
33       public int returnRedness() {  
34           return redness;  
35       }  
36  
37   };
```

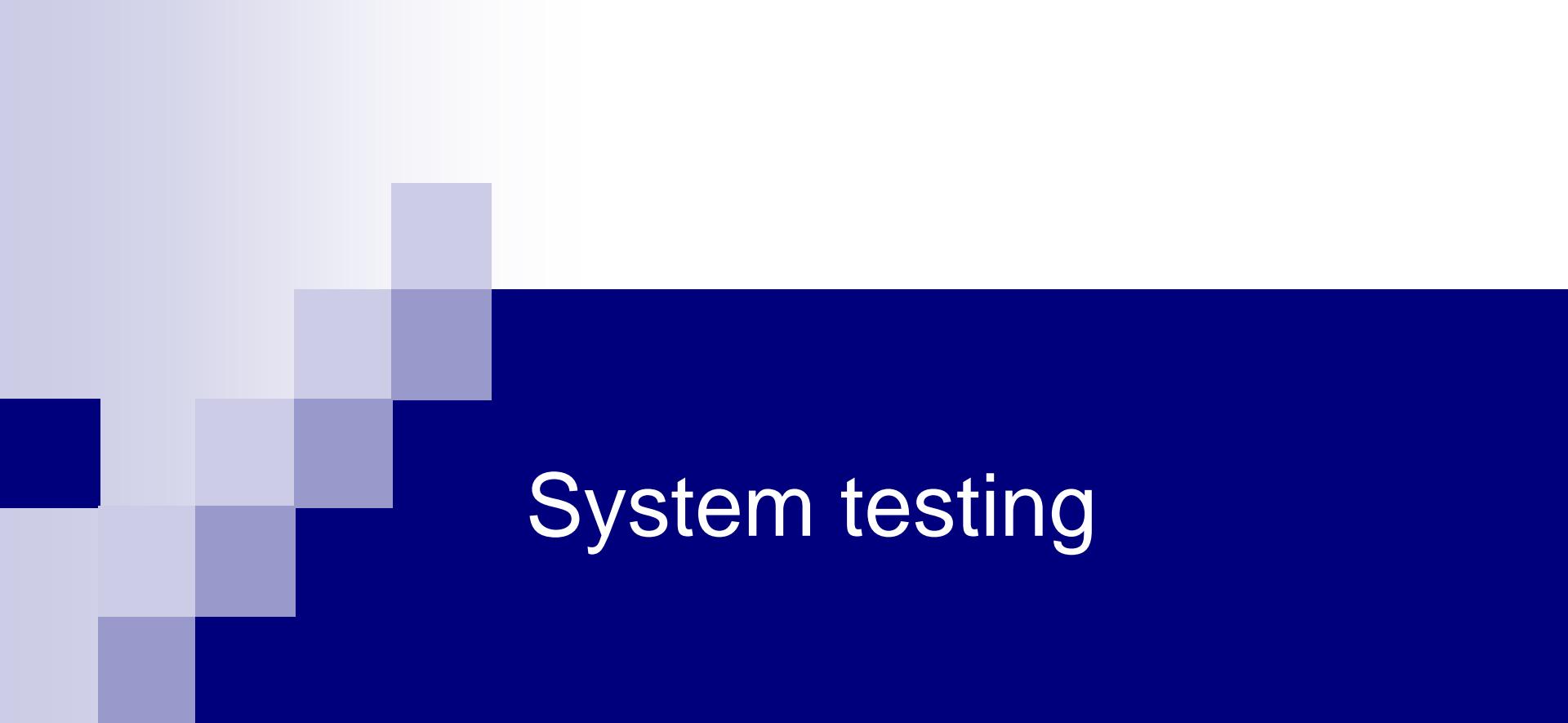
# Polymorphism Issues - Example Code

```
1 public class Strawberry extends Fruit      1 public class Orange{
2
3     private int weight;
4     private int redness;
5
6     public Strawberry() {
7         //Grows a strawberry
8     }
9     public void prepare() {
10        //Wash Fruit
11    }
12    //Returns how red the strawberry is
13    public int returnRedness() {
14        return redness;
15    }
16    public void eat() {
17        //Consume the piece of fruit
18    }
19 };
20
21
22
23 }
```

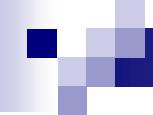
# Solution







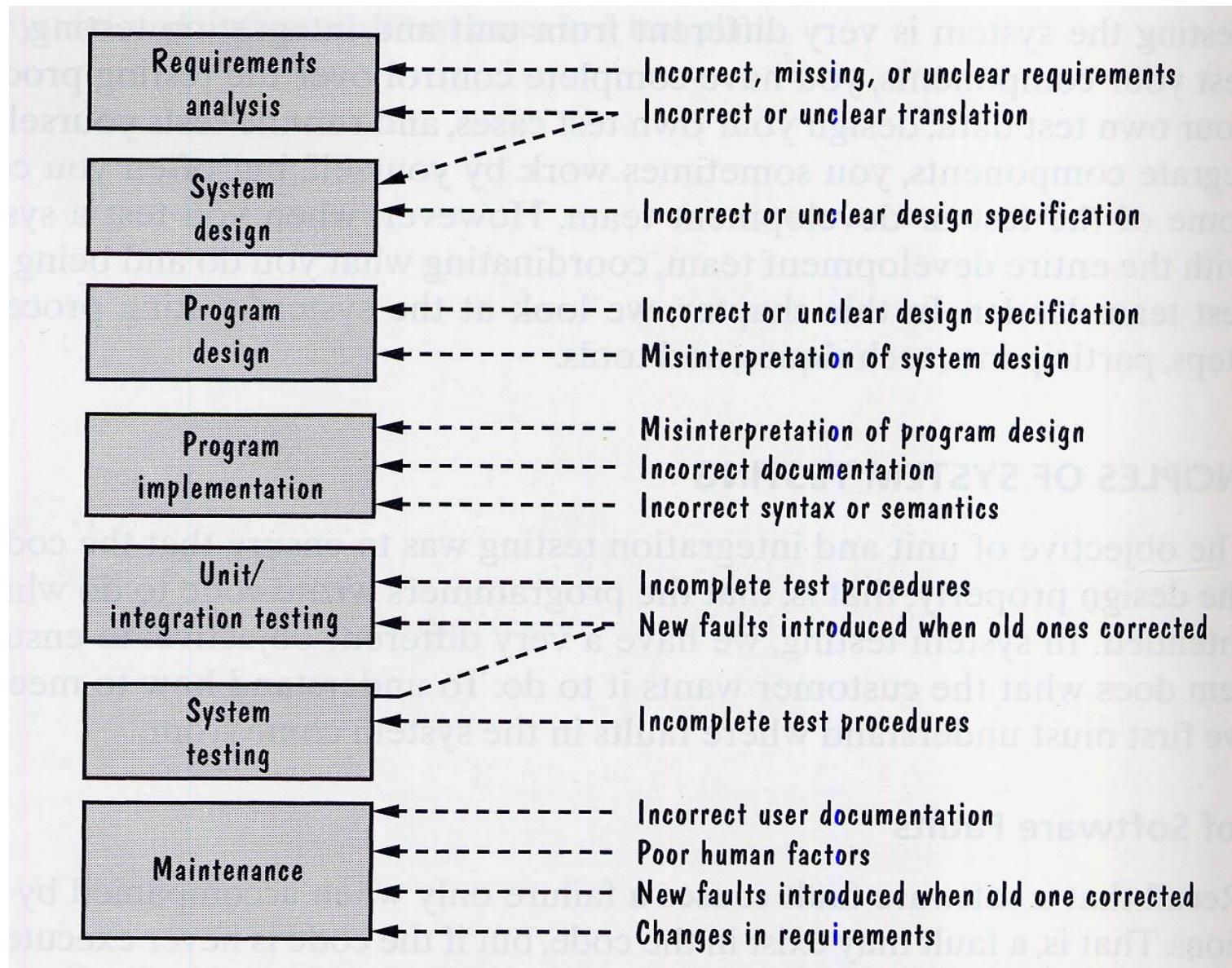
# System testing



# **System Testing**

- 1. Threads**
- 2. Basis concepts of requirements specification**
- 3. Identifying threads**
- 4. Metrics for system testing**

# Causes of Software Faults



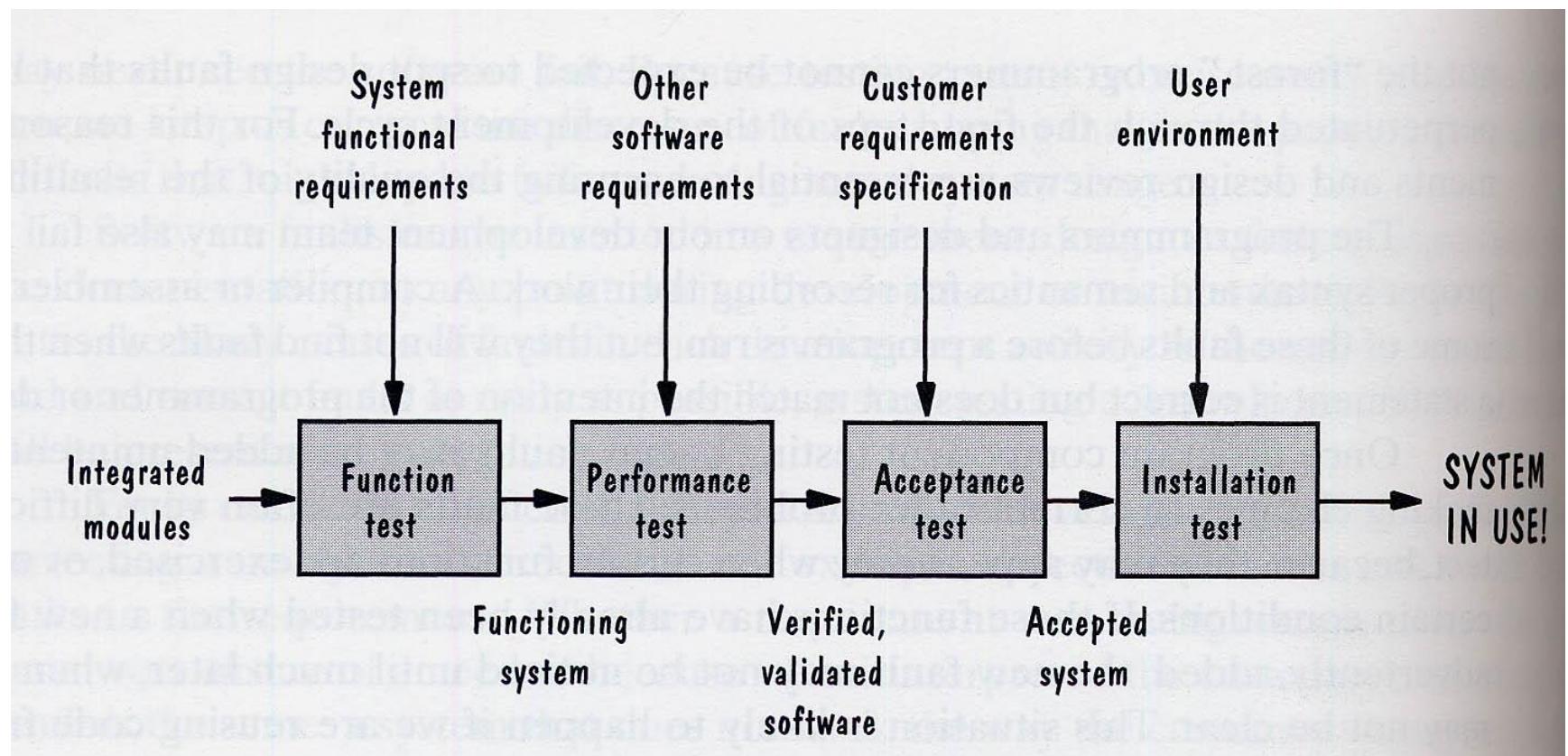
# System Testing

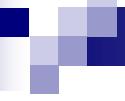
- Of the three levels of testing, the system level testing is closest to everyday experience, that is we evaluate a product with respect to our expectations
- Concerns with the app's **externals**
- Goal is not to find faults but to demonstrate performance
- We tend to approach system testing from a functional standpoint rather than from a structural one
- However, still much more than **functional**
  - Load/stress testing
  - Usability testing
  - Performance testing
  - Resource testing

# System Testing Process

- **Function Testing:** the system must perform functions specified in the requirements.
- **Performance Testing:** the system must satisfy security, precision, load and speed constraints specified in the requirements.
- **Acceptance Testing:** customers try the system (in the lab) to make sure that the system built is the system they requested.
- **Deployment Testing:** the software is deployed and tested in the production environment.

# System Testing Process





# System Testing

## ■ Functional testing

- **Objective:** Assess whether the app does what it is supposed to do
- **Basis:** Behavioral / functional specification
- **Test case:** A sequence of ASFs (thread)

# System Testing

## ■ Functional testing: coverage

### ■ Event-based coverage

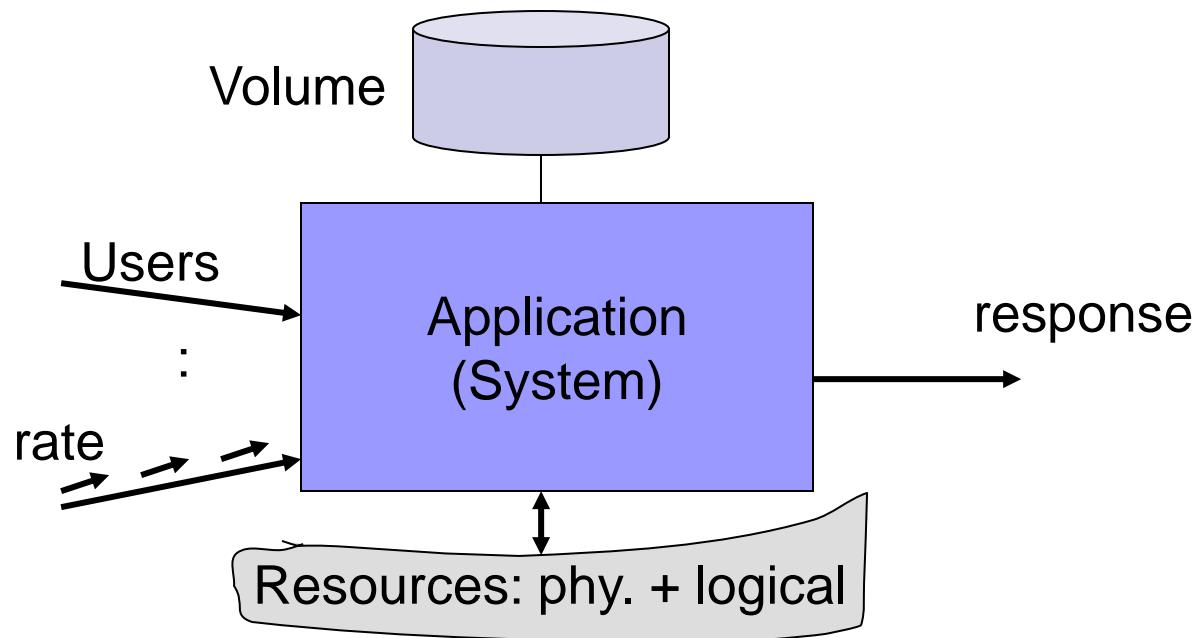
- PI1: **each port input** event occurs
- PI2: **common sequences** of port input event occurs
- PI3: each port input in every relevant **data context**
- PI4: for a given context, **all possible input** events
- PO1: each port **output** event
- PO2: each port **output event** occurs for each **cause**

### ■ Data-based

- DM1: Exercise cardinality of every relationship
- DM2: Exercise (functional) dependencies among relationships

# System Testing

- **Stress testing:** push it to its limit + beyond



# System Testing

## ■ Performance testing

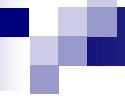
- Performance seen by
  - **users:** delay, throughput
  - **System owner:** memory, CPU, comm
- Performance
  - Explicitly specified or expected to do well
  - Unspecified → find the limit

## ■ Usability testing

- Human element in system operation
  - GUI, messages, reports, ...

# Thread in system testing

- Thread testing is defined as a software testing type, which verify the key functional capabilities of a specific task(thread). It is usually conducted at the early stage of Integration Testing phase.
- Thread based testing is one of the incremental strategies adopted during System Integration testing. That's why, thread test should probably more properly be called a "**thread interaction test.**"
- Thread based testing are classified into two categories
- **Single thread testing:** A single thread testing involves one application transaction at a time
- **Multi-thread testing:** A multi-thread testing involves several concurrently active transaction at a time



# Threads

- We view system testing in terms of threads of system level behavior.
- Many possible views of a thread:
  - a scenario of normal usage
  - a system level test case
  - a stimulus/response pair
  - behavior that results from a sequence of system level inputs
  - an interleaved sequence of port input and output events
  - a sequence of transitions in a state machine description of a system
  - an interleaved sequence of object messages and method executions
  - a sequence of machine instructions
  - a sequence of source instructions
  - a sequence of atomic system functions

# Thread Levels

- Threads have distinct levels:
  - Unit level thread is understood as an execution-time path of instructions or some path on a flow graph. (DD Paths)
  - Integration level thread is a sequence of MM-paths that implement some atomic function. Denoted perhaps as a sequence of module executions and messages.
  - System level thread is a sequence of atomic system functions. (ASF)
- An Atomic System Function (ASF) is an action that is observable at the system level in terms of port input and output events.
- Since ASFs have port events as their inputs and outputs, the sequence of ASFs implies an interleaved sequence of port input/port output events.
  - Threads provide a unifying view of the three levels of testing:
    - Unit testing tests individual functions
    - Integration tests examine interaction among units
    - System testing examines interactions among ASFs.

# Thread definitions

- An ASF begins with a port input event and terminates with a port output event.
- In the SATM system, digit entry is a good example of an ASF—so are card entry, cash dispensing, and session closing. PIN entry is probably too big; perhaps we should call it a molecular system function.
- Given a system defined in terms of ASFs, the ASF graph of the system is the directed graph in which nodes are ASFs and edges represent sequential flow.

# Thread Definitions (1)

- **Unit thread:** is a path in the program graph of a unit.
  - Two levels of threads in integration testing: MM-Paths and ASFs.
- **MM-Path:** is a path in the MM-Path graph of a set of units.
  - directed graph in which module execution paths are nodes and edges show execution time sequence
- **ASF Graph:** (for a system defines in terms of ASFs) is the directed graph in which nodes are ASFs and edges represent sequential flow.

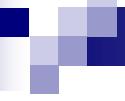
# Thread Definitions (2)

- **Source ASF:** an ASF that appears as a source node in the ASF graph of a system,
- **Sink ASF:** is an ASF that appears as sink node in the ASF graph.
  - In the SATM system, the card entry ASF is a source ASF, and the session termination ASF is a sink ASF. Notice that intermediary ASFs could never be tested at the system level by themselves—they need the predecessor ASFs to “get there.”
- **System thread:** a path from a source ASF to a sink ASF in the ASF graph of a system.
- **Thread graph:** (for a system defined in terms of system threads) is directed graph in which nodes are system threads and edges represent sequential execution of individual threads.
- The above definitions provide a coherent set of increasing broader views of threads.

# Basis Concepts for Requirements Specification

(tester's view of requirements specification to see how to identify threads)

- The objective is to discuss system testing with respect to a basis set of requirements specification constructs
- Consider the following requirements specification constructs:
  - Data
  - Actions
  - Ports
  - Events
  - Threads
- Every system can be specified in terms of these basic concepts.



# Thread identification process

(examine these fundamental concepts here to see how they support the tester's process of thread identification)

- Data
- Actions
- Ports
- Events

# Data-Centric Thread Identification

- For a system that is described in terms of its data,
  - the focus is on the information used/created by the system (described in terms of variables, data structures, fields, records, data stores, and files)
  - for example ER models are useful at the highest level.
- The data centered view is also starting point for many OO analysis methods.
- Data refers to information that is either initialized, stored, updated or possibly destroyed.
- Data-centric systems are often specified in terms of **CRUD** actions (Create, Retrieve, Update, Delete)

# Data-Centric Thread Identification

- Often, threads can be identified directly from the data model.
  - Relationships can be 1:1, n:1, etc. and these distinctions have implications for threads that process the data.
- Also possible to have read-only data (i.e. expected PIN pairs, etc.)
  - this must be part of system initialization process
    - if not, then there must be threads that create the data.
    - Hence read-only data is an indicator of source ASFs.

# Action-Centric Thread Identification

- Action centered modeling is most common requirements specification form.
  - Actions have inputs and outputs and these can be either data or port events.
  - Actions can also be decomposed into lower level actions (i.e. typical data flow diagrams).
    - methodology-specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service.
- The input/output view of actions is the basis of functional testing and,
  - the decomposition [and eventual implementation] of actions is the basis of structural testing.
  - The input/output view of actions is exactly the basis of specification-based testing, and the decomposition (and eventual implementation) of actions is the basis of code-based testing.

# Port-Centric Thread Identification

- Every system has ports (and port devices):
  - These are sources and destination of system level inputs and outputs. (port events)
  - distinction between port device which is connected to a system port
- The slight distinction between ports and port devices is sometimes helpful to testers.
- Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports.
- Physical actions (keystrokes and light emissions from a screen) occur on port devices, and these are translated from physical to logical (or logical to physical)
- If no physical port devices in system, much of system testing can be accomplished by moving the port boundary inward to the logical instances of port events.

The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doors, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on.

# Event-Centric Thread Identification

- Have characteristics of data and some of actions
  - a system level input which occurs at a port.
- An event is a system level input (or output) that occurs at a port.
- Like data, events can be inputs to or outputs of actions:
  - Can be either discrete (more generally) or continuous (temperature, altitude, etc.).
  - Events can be discrete (such as SATM keystrokes) or they can be continuous (such as temperature, altitude, or pressure).Events have characteristics of data and some of actions
    - i.e. a system level input which occurs at a port.

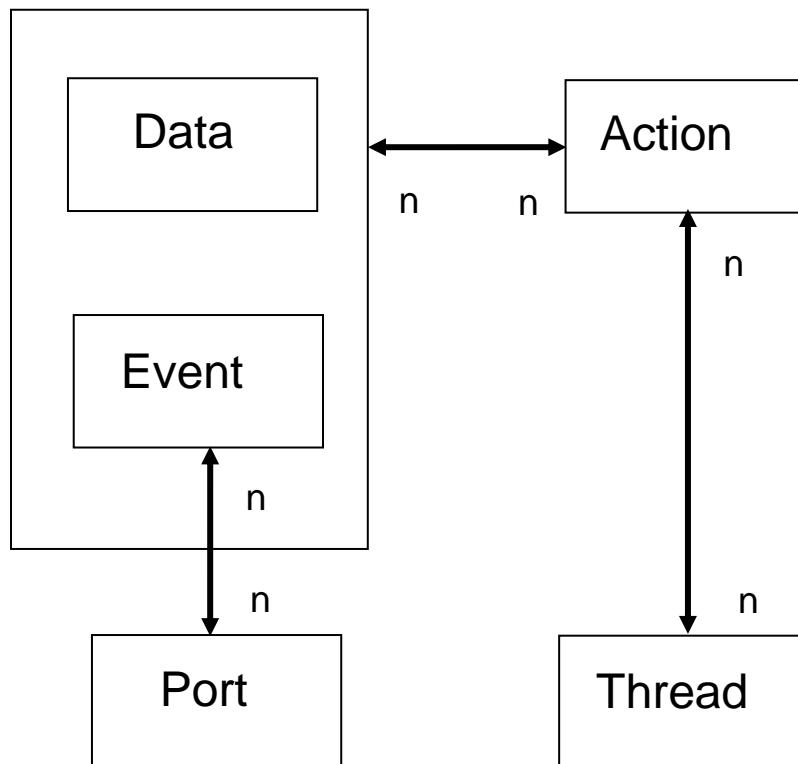
- Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations, and, symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers).

In the SATM system, for example, the port input event of depressing button B1 means “balance” when screen 5 is displayed, “checking” when screen 6 is displayed, and “yes” when screens 10, 11, and 14 are displayed. We refer to these situations as “context-sensitive port events,” and we would expect to test such events in each context.

# Threads

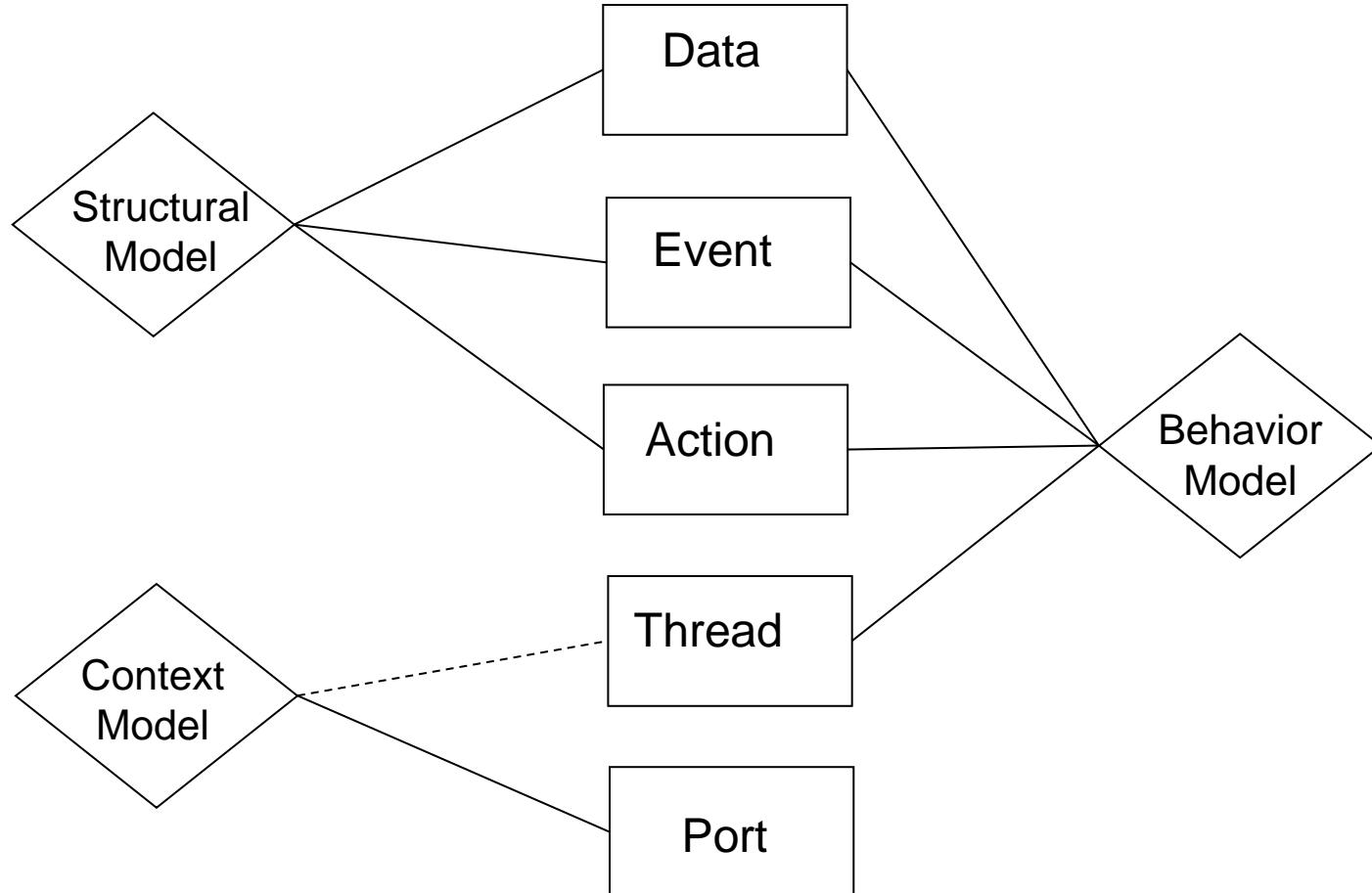
- Threads are the least frequently used of the fundamental constructs.
  - Since threads are tested, it is up to tester to find them in the interactions of the data, events, and actions.
  - Usually easy to find threads in a control model of the system.
- Finding Threads
  - A finite state machine model of the system is a good starting point to find threads since the paths are easily converted to threads.

# E/R Model of Basis Concepts



Notice that all relationships are many-to-many:  
Data and Events are inputs to or outputs of the Action entity. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions.

# Modeling Relationships



# Finding Threads (1)

- Usually, one deals with a hierarchy of state machines i.e. the card entry state of an ATM may be decomposed into lower levels that deal with details like:
  - jammed cards,
  - cards that are upside-down,
  - checking the card against the list of cards for which service is offered, etc.).

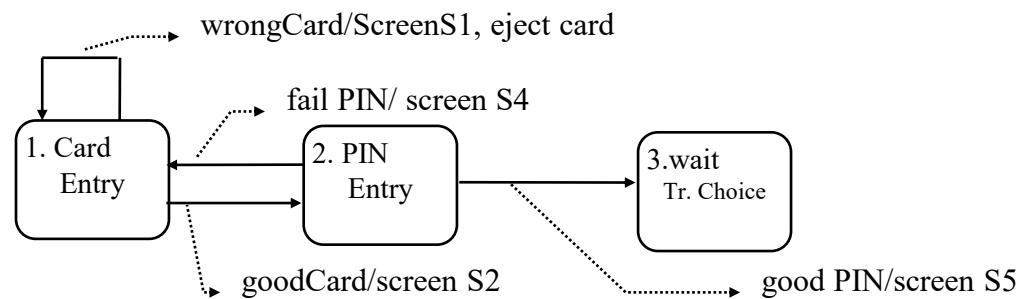


Figure 1.

# Finding Threads (2)

- At this level, states correspond to states of processing, and transitions are caused by logical (rather than port) events.
- Once the details of a macro-state are tested,
  - We then use an easy thread to get to the next macro-state.
  - Within the decomposition of the macro state,
    - identify the port input and port output events.[ i.e. within 2.1 on Figure 2, digit and cancel key port input occur]

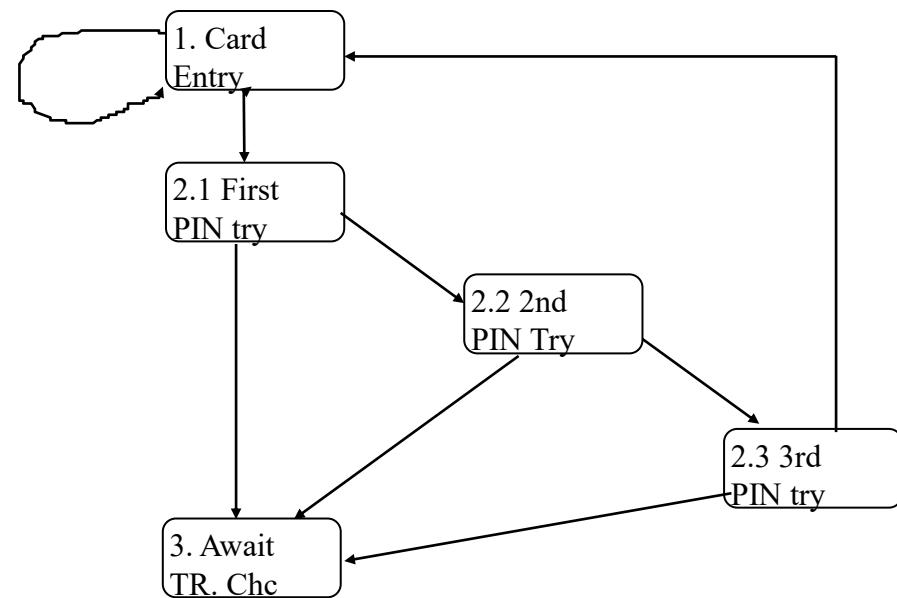


Figure 2.

# Finding Threads (3)

- the hierarchy of finite state machines multiplies the number of threads.
- ideal to reach a state machine in which transitions are caused by actual port input events, and the actions on transitions are port output events
  - then generating the test cases for these threads is mechanical
  - just follow a path of transitions,
    - and note the inputs and outputs as they occur along the path.

# Structural Strategies for Thread Testing

- Generating the threads may be easy, but to decide which one to test is complex.
- Encounter the same path explosion problem at system level as at unit level.
- Bottom Up Threads
  - When state machines are organized in a hierarchy, it is possible to work bottom up. i.e. in Fig 2, the “PIN Try” state machine might consist of 6 paths.
    - Traverse these paths and then go up one level to the “PIN Entry” [fig. 3] machine which has 4 basic paths and so on up the hierarchy

# Structural Strategies for Thread Testing

- As seen in unit testing, structural testing can be misleading.
  - The assumption is that path traversal uncovers faults and traversing a variety of paths reduces redundancy.
- A more serious flaw with these threads is that it is not really possible to execute them “by themselves” due to the hierarchical state machines.

### Screen 1

Welcome.

Please Insert your  
ATM card for service

### Screen 2

Enter your Personal  
Identification Number

— — — —  
Press Cancel if Error

### Screen 3

Your Personal  
Identification Number  
is incorrect. Please  
try again.

### Screen 4

Invalid identification.  
Your card will be  
retained. Please call  
the bank.

### Screen 5

Select transaction type:  
balance  
deposit  
withdrawal

Press Cancel if Error

### Screen 6

Select account type:

checking  
savings

Press Cancel if Error

### Screen 7

Enter amount.  
Withdrawals must be  
in increments of \$10

Press Cancel if Error

### Screen 8

Insufficient funds.  
Please enter a new  
amount.

Press Cancel if Error

### Screen 9

Machine cannot  
dispense that amount.

Please try again.

### Screen 10

Temporarily unable to  
process withdrawals.  
Another transaction?

yes  
no

### Screen 11

Your balance is being  
updated. Please take  
cash from dispenser.

### Screen 12

Temporarily unable to  
process deposits.  
Another transaction?

yes  
no

### Screen 13

Please put envelope into  
deposit slot. Your  
balance will be updated

Press Cancel if Error.

### Screen 14

Your new balance is  
printed on your receipt.  
Another transaction?

yes  
no

### Screen 15

Please take your  
receipt and ATM  
card. Thank you.

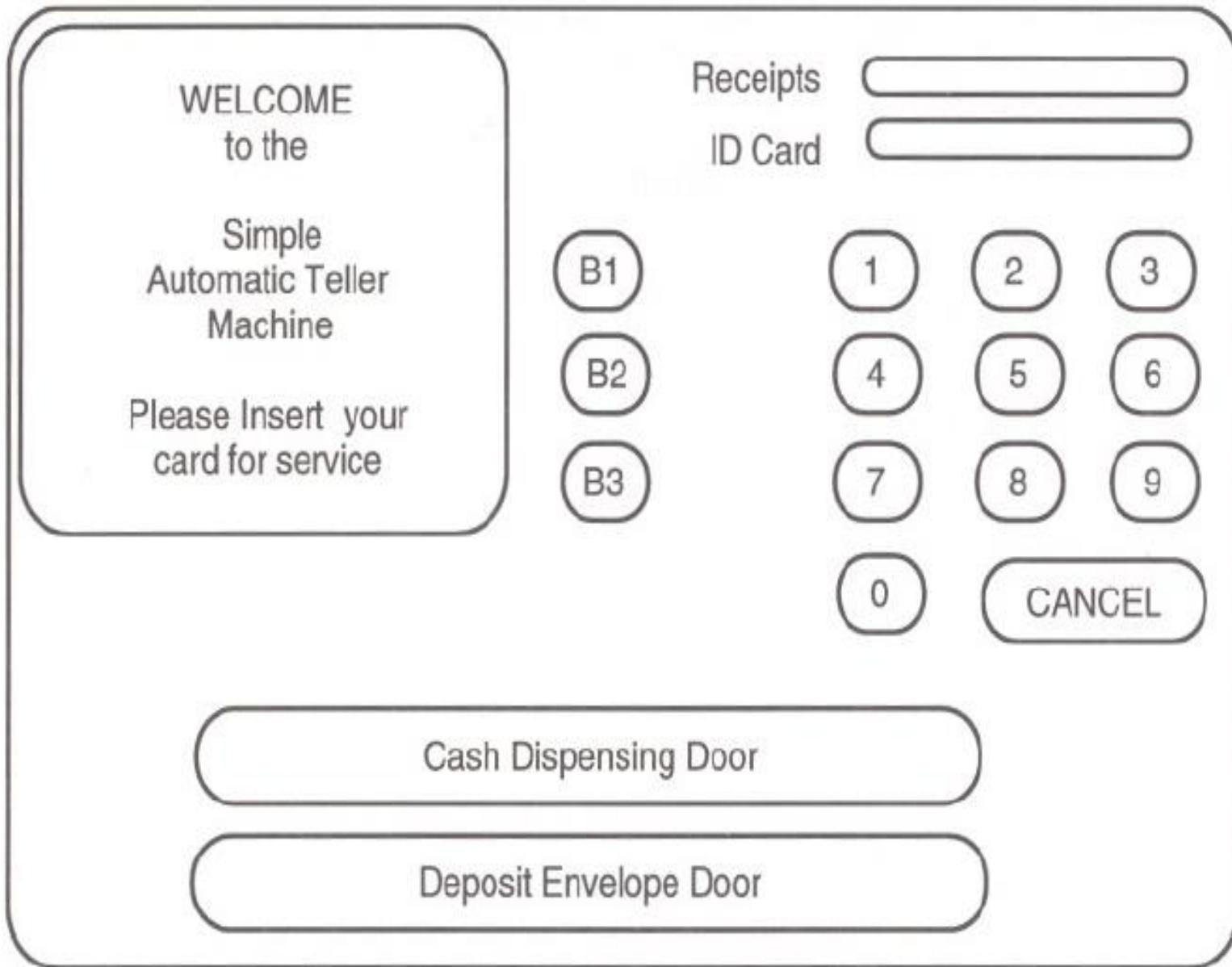
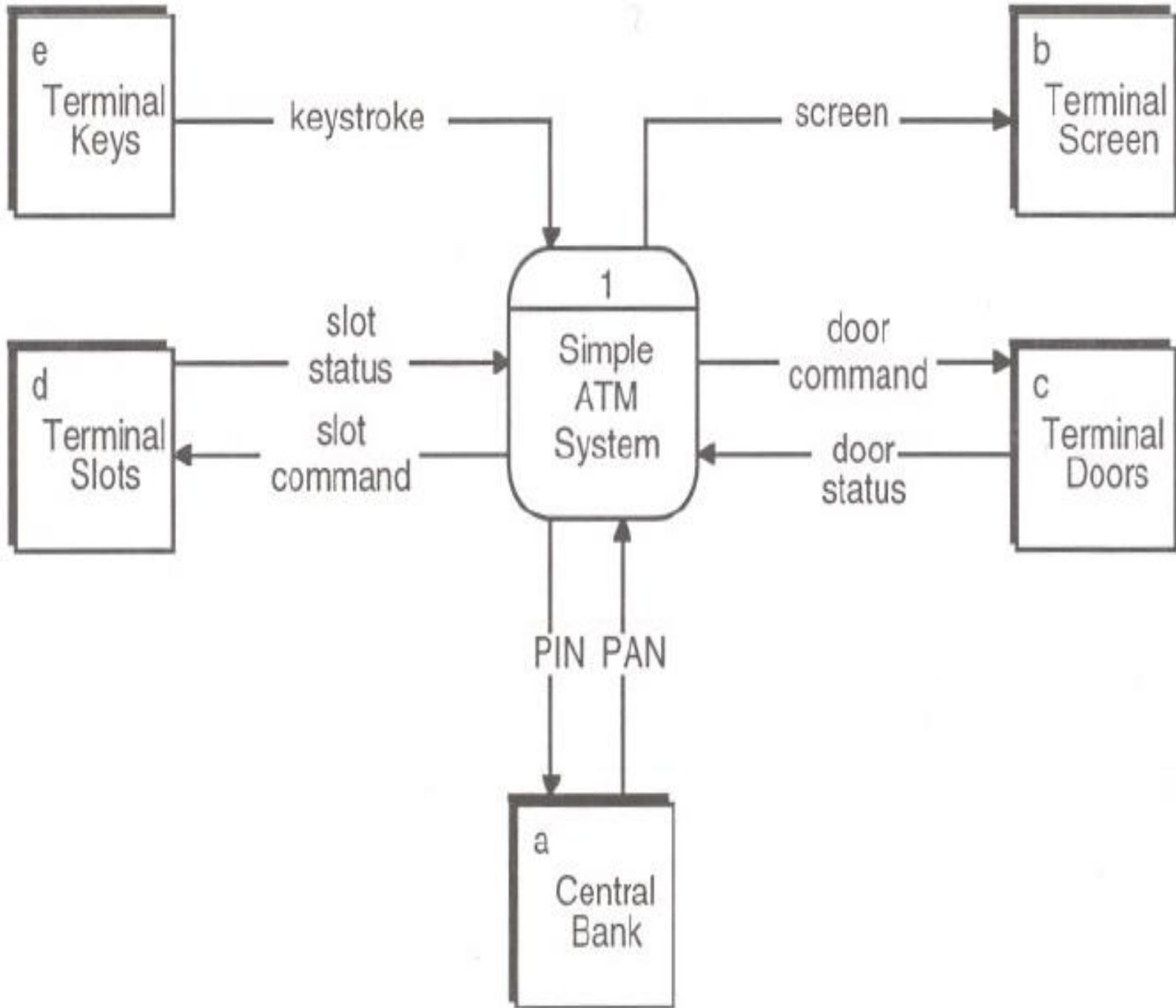


Figure 12.8 The SATM Terminal



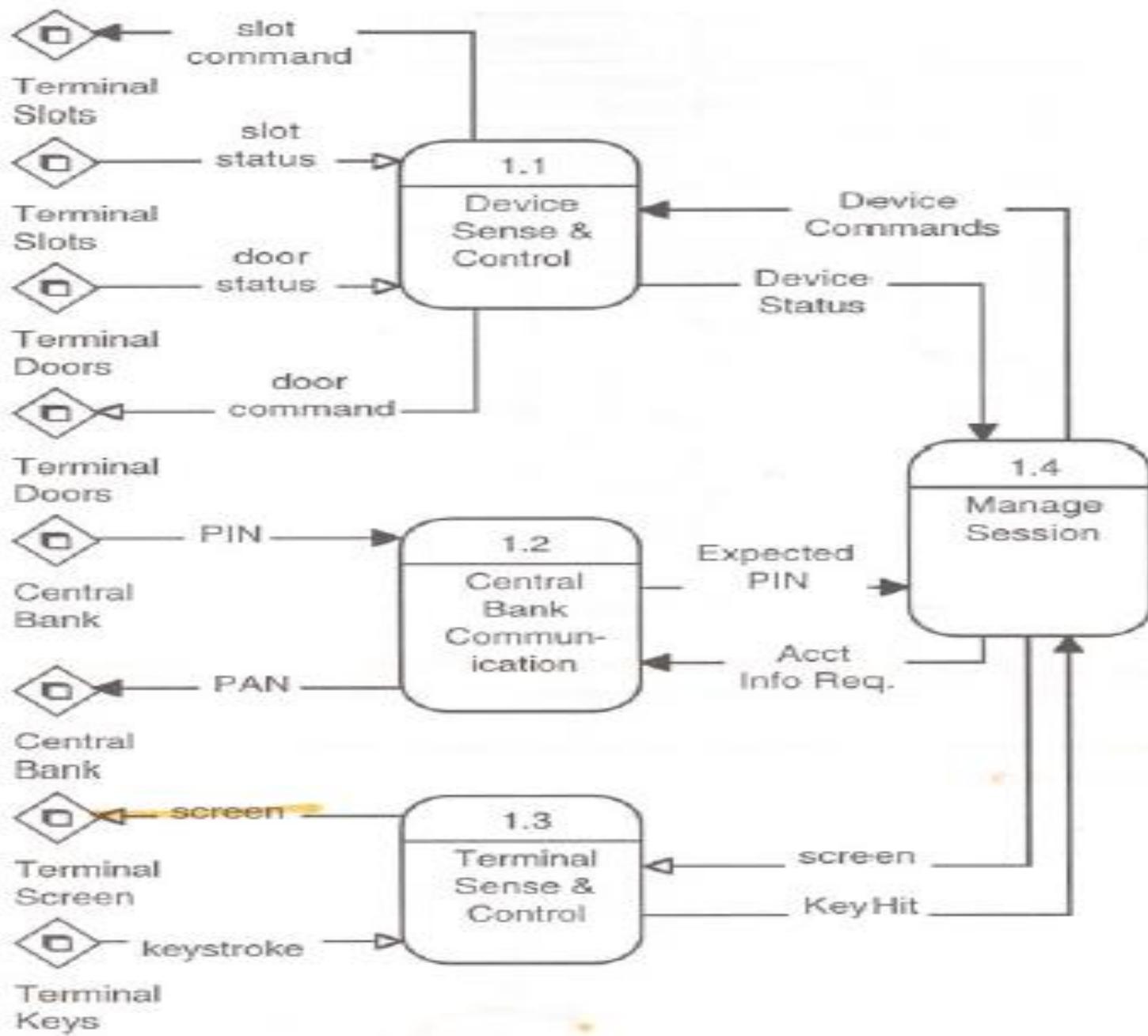


Figure 12.10 Level 1 Dataflow Diagram of the SATM System

# Candidate Threads

- **Entry of a digit**
- **Entry of a Personal Identification Number (PIN)**
- **A simple transaction: ATM Card Entry, PIN entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results.**
- **An ATM session, containing two or more simple transactions.**

# Classifying the Candidate Threads

- an MM-Path • Entry of a digit
- an ASF • Entry of a Personal Identification Number (PIN)
- a Thread • A simple transaction: ATM Card Entry, PIN entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results.
- a sequence of threads • An ATM session, containing two or more simple transactions.

ASF=Atomic system function  
FSM=Finite State machine

# **Closer Look at the PIN Entry ASF**

**a sequence of system level inputs and outputs:**

- 1. A screen requesting PIN digits**
- 2. An interleaved sequence of digit keystrokes and screen responses**
- 3. The possibility of cancellation by the customer before the full PIN is entered**
- 4. A system disposition: (A customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type; otherwise a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.)**

**Observe:**

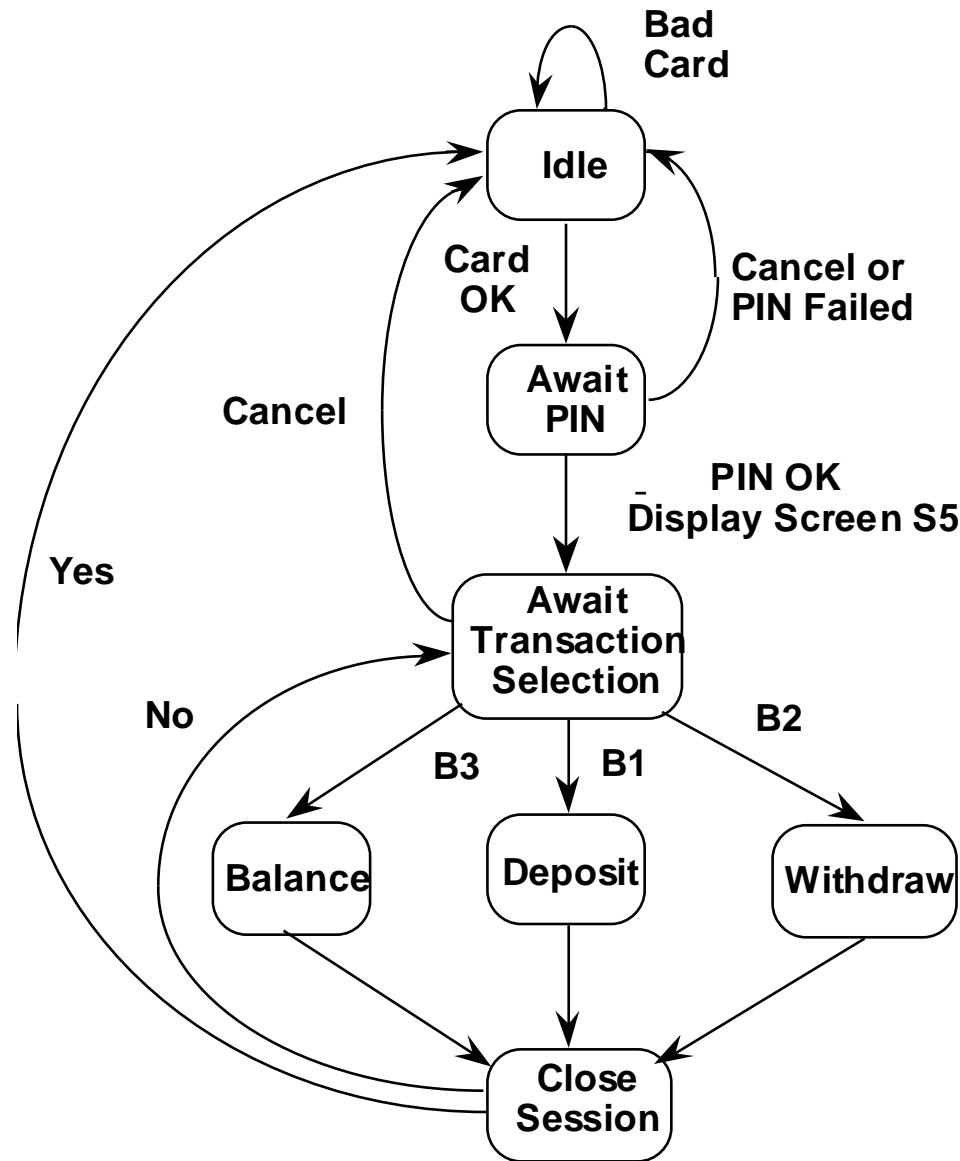
**several stimulus/response pairs**

**this is the cross-over point between integration and system testing**

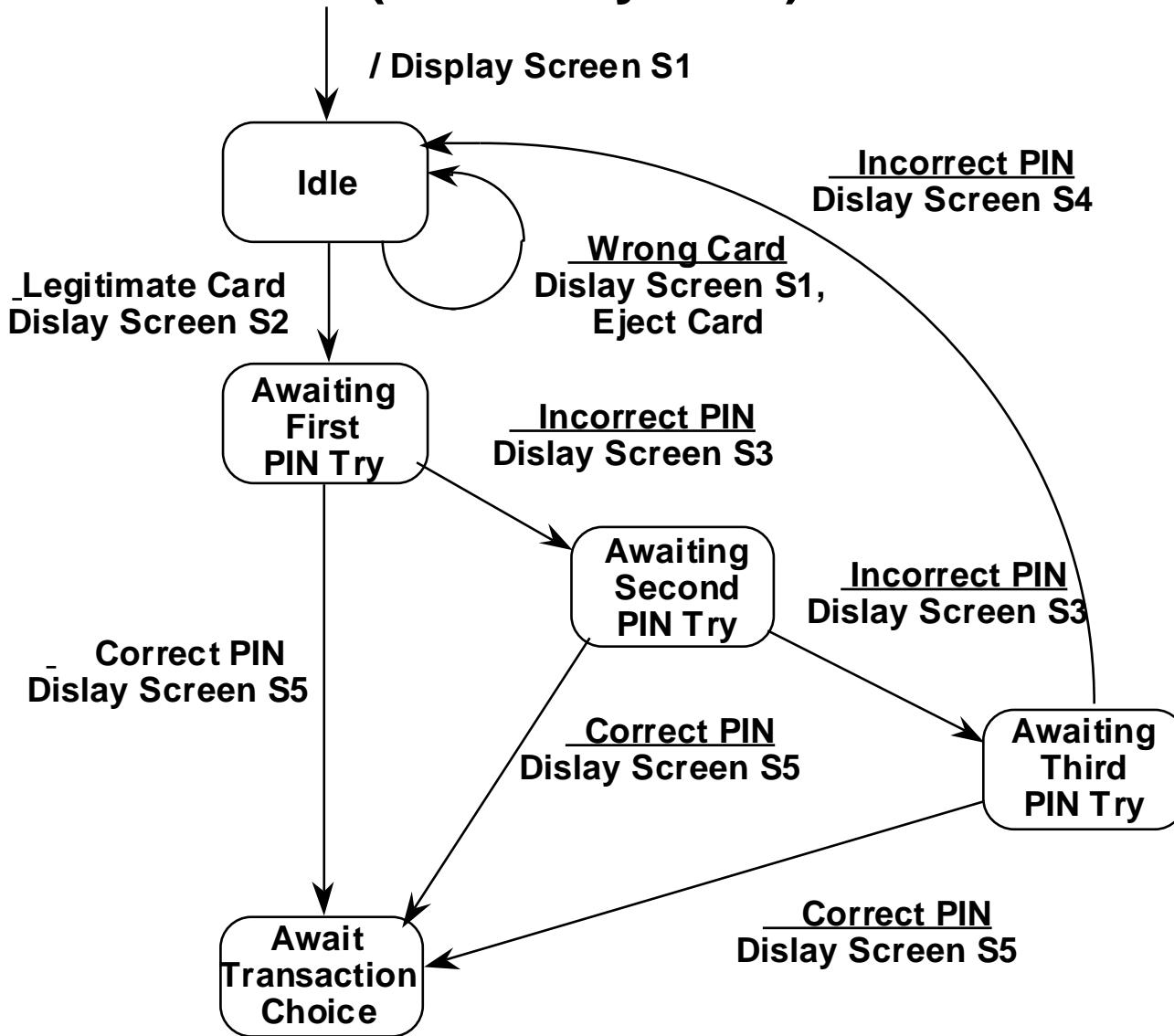
# Events in the PIN Entry Finite State Machine

Port Input Events	Port Output Events
Legitimate Card	Display screen 1
Wrong Card	Display screen 2
Correct PIN	Display screen 3
Incorrect PIN	Display screen 4
Canceled	Display screen 5

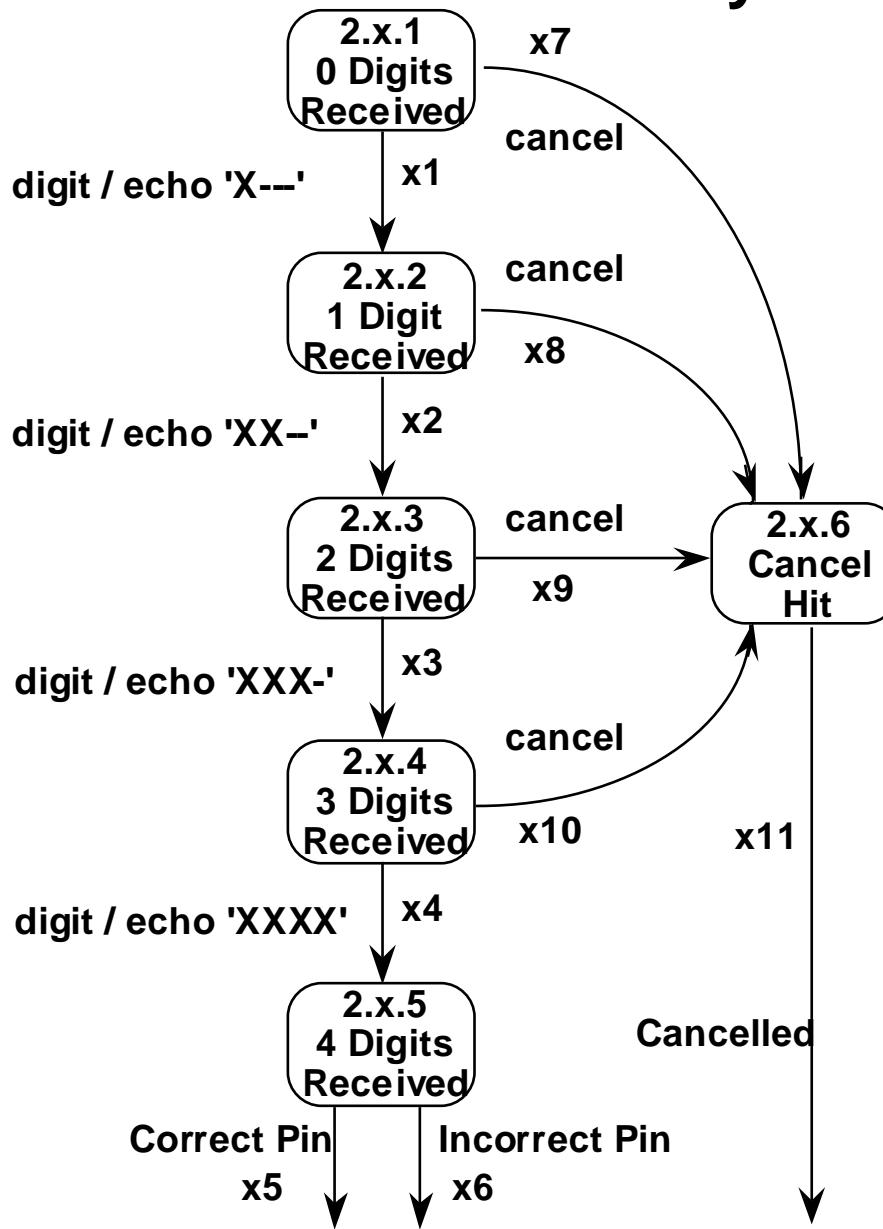
# SATM Top Level FSM



# Decomposition of Await PIN State (PIN Entry FSM)



# PIN Try finite state machine (PIN Try x)



**Port Input Events**

digit  
cancel

**Port Output Events**

echo 'X--'  
echo 'XX--'  
echo 'XXX--'  
echo 'XXXX'

**Logical Output Events**

Correct PIN  
Incorrect PIN  
Canceled

# Deriving An ASF Test Case

**Description: Correct PIN on First Try**

**Port Event Sequence in PIN Try FSM**

**Port Input Event**

**1 pressed**

**2 pressed**

**3 pressed**

**4 pressed**

**(Correct PIN)**

**Port Output Event**

**Screen 2 displayed with '----'**

**Screen 2 displayed with 'X---'**

**Screen 2 displayed with 'XX--'**

**Screen 2 displayed with 'XXX-'**

**Screen 2 displayed with 'XXXX'**

**Screen 5 displayed**

## Use Case–Based Threads

### Levels of Use Cases

- High level (very similar to an agile user story)
  - Essential
  - Expanded essential
  - Real

High-level use cases are at the level of the user stories used in agile development. A set of high-level use cases gives a quick overview of the does view of a system.

Essential use cases add the sequence of port input and output events. At this stage, the port boundary begins to become clear to both the customer/user and the developer.

Expanded essential use cases add pre and post conditions. We shall see that these are key to linking use cases when they are expressed as system test cases.

Real use cases are at the actual system test case level. Abstract names for port events, such as “invalid PIN,” are replaced by an actual invalid PIN character string.

# Converting Use Cases to Event-Driven Petri Nets

As the name implies, they are appropriate for any event-driven system, particularly those characterized by context-sensitive port input events.

In an EDPN drawing, port events are shown as triangles, data places are circles, transitions are narrow rectangles, and the input and output connections are arrows. Similarly, output port events are upward pointing,

Then Convert Finite State Machines to Event-Driven Petri Nets

# Long versus Short Use Cases

- A customer enters a valid card, followed by a valid PIN entry on the first try. The customer selects the Withdraw option, and enters \$20 for the withdrawal amount. The SATM system dispenses two \$10 notes and offers the customer a chance to request another transaction. The customer declines, the SATM system updates the customer account, returns the customer's ATM card, prints a transaction receipt, and returns to the Welcome screen.
- Here we suggest “short use cases,” which begin with a port input event and end with a port output event.
- The long use case above might be expressed in terms of four short use cases:
  - 1. Valid card
  - 2. Correct PIN on first try
  - 3. Withdrawal of \$20
  - 4. Select no more transitions

# Metrics for System Testing

In the PIN Entry ASF, for a given PIN, there are 156 distinct paths from the First PIN Try state to the Await Transaction Choice or Card Entry states in the PIN Entry FSM. Of these, 31 correspond to eventually correct PIN entries (1 on the first try, 5 on the second try, and 25 on the third try); the other 125 paths correspond to those with incorrect digits or with cancel keystrokes.

To control this explosion, we have two possibilities:

- **pseudo-structural coverage metrics**
- "true" structural coverage metrics

# Node and Edge Coverage Metrics

- Since FSMs are directed graphs, use same test coverage metrics as at the unit level.
- The hierarchical relationship indicates that the upper level machine treats the lower level machine as a procedure that is entered and returned from.
- Two fundamental choices are node coverage and edge coverage
  - node coverage is similar to statement coverage at unit level: bare minimum .
  - edge (state transition) coverage is more acceptable. If the state machines are ‘well formed’ [transitions in terms of port events], then edge coverage also guarantees port event coverage.

# Pseudo-structural Coverage Metrics

**Behavioral models provide "natural" metrics:**

**Decision Table Metrics:** • every condition  
• every action  
• every rule

**FSM Metrics:** • every state  
• every transition

**Petri Net Metrics:** • every place  
• every port event  
• every transition  
• every marking

**These are pseudo-structural because they  
are just models of the eventual system.**

# Pseudo-structural Coverage Metrics for PIN Try

Input Event Sequence	State Sequence
1,2,3,4	2.x.1, 2.x.2, 2.x.3, 2.x.4, 2.x.5
1,2,3,5	2.x.1, 2.x.2, 2.x.3, 2.x.4, 2.x.5
C	2.x.1, 2.x.6
1,C	2.x.1, 2.x.2, 2.x.6
1,2,C	2.x.1, 2.x.2, 2.x.3, 2.x.6
1,2,3,C	2.x.1, 2.x.2, 2.x.3, 2.x.4, 2.x.6

Two test cases yield state coverage

Input Event Sequence	Path of Transitions
1,2,3,4	x1, x2, x3, x4, x5
1,2,3,5	x1, x2, x3, x4, x6
C	x7, x11
1,C	x1, x8, x11
1,2,C	x1, x2, x9, x11
1,2,3,C	x1, x2, x3, x10, x11

All six are needed for transition coverage

# Pseudo-structural Coverage Metrics for PIN Entry FSM

Input Event Sequence	State Sequence
----------------------	----------------

1,2,3,4

1,2,3,5,1,2,3,4

1,2,3,5,C,1,2,3,4

C,C,C

**How many test cases yield state coverage?**

Input Event Sequence	Path of Transitions
----------------------	---------------------

1,2,3,4

1,2,3,5,1,2,3,4

1,2,3,5,C,1,2,3,4

C,C,C

**How many are needed for transition coverage?**

# **Consequences of Pseudo-structural Coverage Metrics**

## **1. Combinatoric explosion is controlled**

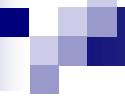
Selecting test cases from the FSM decomposition reduced 156 threads to 10 test cases

## **2. Fault isolation is improved**

When a "verify" operation fails, use the FSMs to determine:

- what went wrong**
- where it went wrong**
- when it went wrong**

## **3. Base information for testing management**



# SATM System Threads

1. Insertion of an invalid card. (this is probably the "shortest" system thread)
2. Insertion of a valid card, followed by three failed PIN entry attempts.
3. Insertion of a valid card, a correct PIN entry attempt, followed by a balance inquiry.
4. Insertion of a valid card, a correct PIN entry attempt, followed by a deposit.
5. Insertion of a valid card, a correct PIN entry attempt, followed by a withdrawal.
6. Insertion of a valid card, a correct PIN entry attempt, followed by an attempt to withdraw more cash than the account balance.

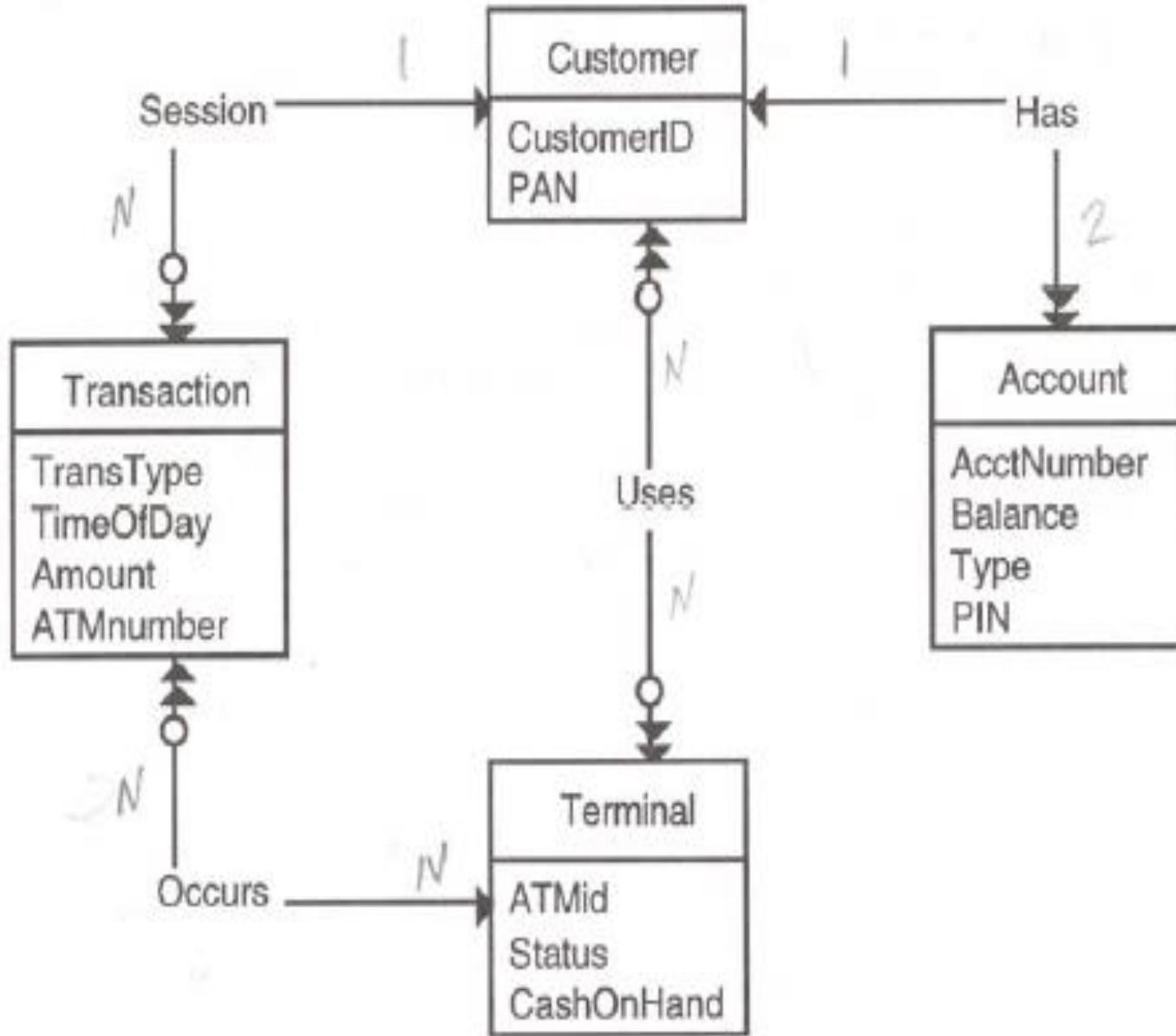


Figure 12.11 Entity/Relationship Model of the SATM System

# SATM System Thread Testing

## SATM Test Data

ATM Card	PAN	Expected PIN	Checking Balance	Savings Balance
1	100	1234	\$1000.00	\$800.00
2	200	4567	\$100.00	\$90.00
3	300	6789	\$25.00	\$20.00
4		(invalid)		

## Port Input Events

- Insert ATM Card (n)
- Key Press Digit (n)
- Key Press Cancel
- Key Press Button B(n)
- Insert Deposit Envelope

## Port Output Events

- Display Screen(n,text)
- Open Door(dep, withdraw)
- Close Door(dep, withdraw)
- Dispense Notes (n)
- Print Receipt (text)
- Eject ATM Card

## Thread 1 Test Procedure

**Description:** invalid card

## **Test operator instructions**

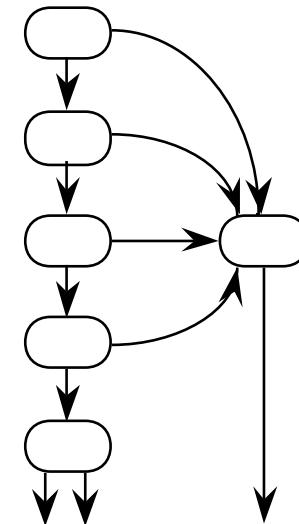
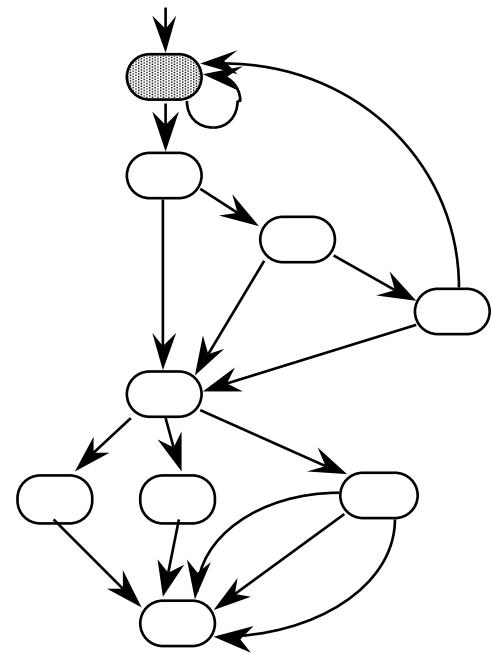
## Initial Conditions: screen 1 being displayed

## **Perform the following sequence of steps:**

1. Cause: Insert ATM Card 4
  2. Verify: Eject ATM Card
  3. Verify: Display Screen(1, null)

## **Post Condition: Screen 1 displayed**

**Test Result:**  Pass  
 Fail



## Thread 2 Test Procedure

**Description:** valid card, 3 failed PIN attempts

## Initial Conditions: screen 1 being displayed

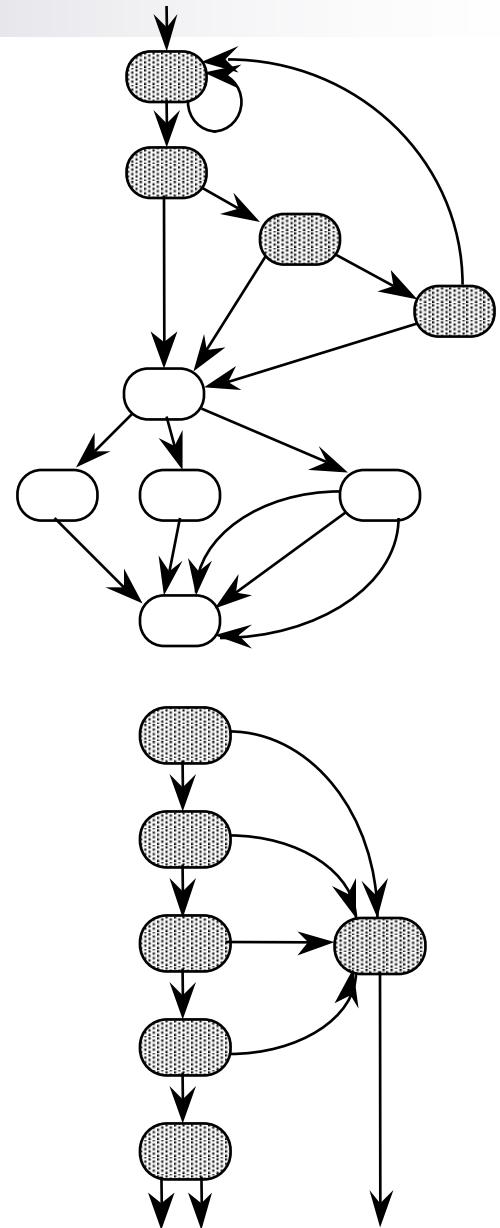
**Perform the following sequence of steps:**

- |    |         |                           |
|----|---------|---------------------------|
| 1. | Cause:  | Insert ATM Card 1         |
| 2. | Verify: | Display Screen(2, '----') |
| 1. | Cause:  | Key Press Digit (1)       |
| 2. | Verify: | Display Screen(2,'X---')  |
| 1. | Cause:  | Key Press Cancel          |
| 2. | Verify: | Display Screen(2,'----')  |
| 1. | Cause:  | Key Press Digit (1)       |
| 2. | Verify: | Display Screen(2,'X---')  |
| 1. | Cause:  | Key Press Digit (2)       |
| 2. | Verify: | Display Screen(2,'XX--')  |
| 1. | Cause:  | Key PressCancel           |
| 2. | Verify: | Display Screen(2,'----')  |
| 1. | Cause:  | Key Press Digit (1)       |
| 2. | Verify: | Display Screen(2,'X---')  |
| 1. | Cause:  | Key Press Digit (2)       |
| 2. | Verify: | Display Screen(2,'XX--')  |
| 1. | Cause:  | Key Press Digit (3)       |
| 2. | Verify: | Display Screen(2,'XXX-')  |
| 1. | Cause:  | Key Press Digit (5)       |
| 2. | Verify: | Display Screen(2,'XXXX')  |
| 3. | Verify: | Display Screen(4, null)   |
| 3. | Verify: | Display Screen(1, null)   |

**Post Condition: Screen 1 displayed**

## Test Result:

Pass  
Fail



# Thread 3 Test Procedure

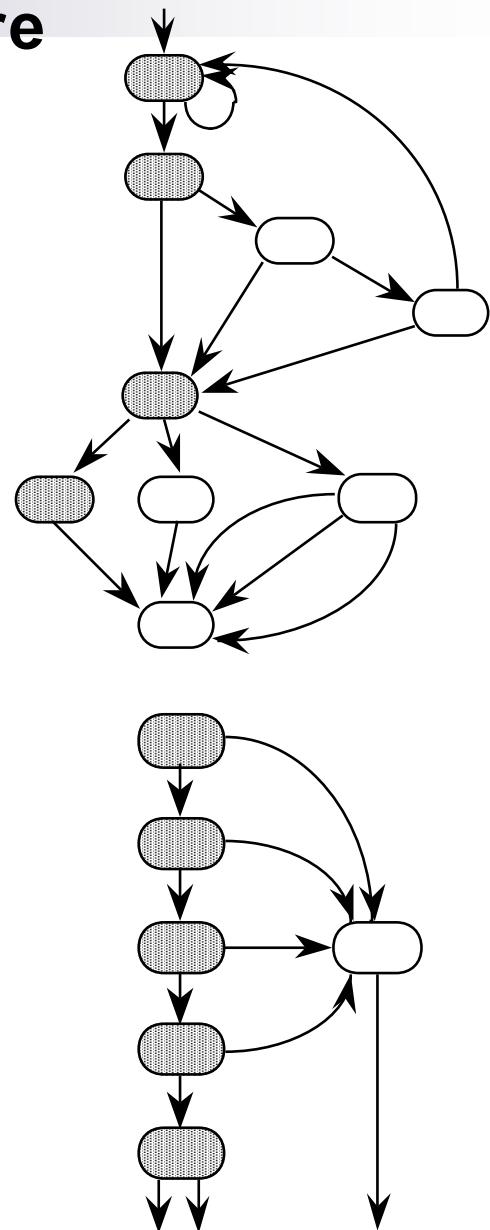
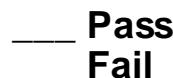
**Description:** valid card, a correct PIN entry attempt, followed by a balance inquiry of the checking account.

## Initial Conditions: screen 1 being displayed

**Perform the following sequence of steps:**

1. Cause: Insert ATM Card 1
  2. Verify: Display Screen(2, '---')
  3. Cause: Key Press Digit (1)
  4. Verify: Display Screen(2,'X--')
  5. Cause: Key Press Digit (2)
  6. Verify: Display Screen(2,'XX--')
  7. Cause: Key Press Digit (3)
  8. Verify: Display Screen(2,'XXX-')
  9. Cause: Key Press Digit (4)
  10. Verify: Display Screen(2,'XXXX')
  11. Verify: Display Screen(5, null)
  12. Cause: Key Press Button B(1)
  13. Verify: Display Screen(6, null)
  14. Cause: Key Press Button B(1)
  15. Verify: Display Screen(14,null)
  16. Cause: Key Press Button B(2)
  17. Verify: Print Receipt ('\$1000.00')
  18. Verify: Display Screen(15, null)
  19. Verify: Eject ATM Card
  20. Verify: Display Screen(1, null)

**Post Condition: Screen 1 displayed Test Result:**



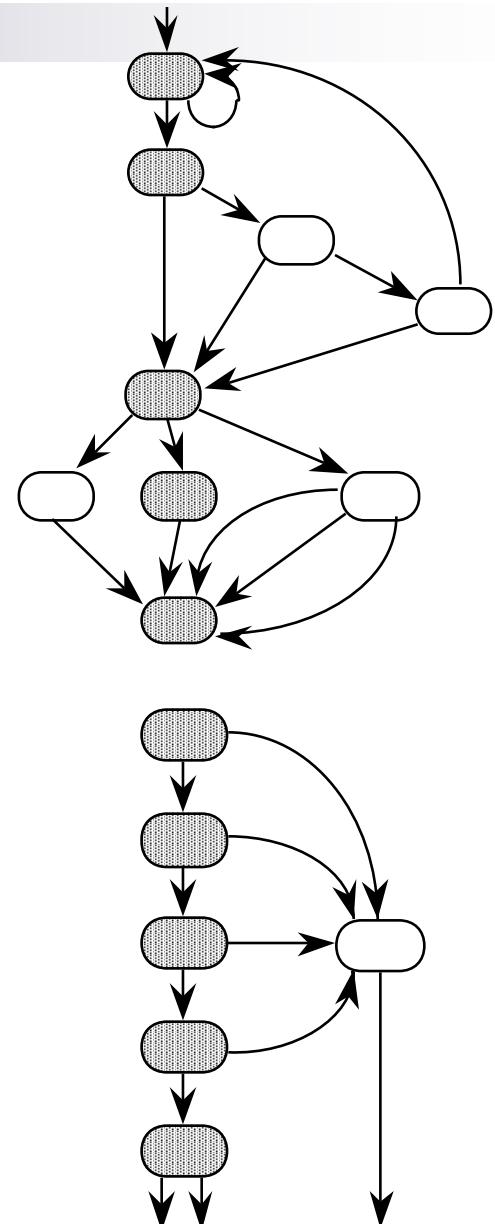
# Thread 4 Test Procedure

Description: valid card, a correct PIN entry attempt, followed by a \$25.00 deposit to the checking account

1. Cause: Insert ATM Card 1
2. Verify: Display Screen(2, '---')
- 3,5,7,9. Cause: Key Press Digit (1,2,3,4)
- 4,6,8,10. Verify: Display Screen(2,'XXXX')
11. Verify: Display Screen(5, null)
12. Cause: Key Press Button B(2)
13. Verify: Display Screen(6, null)
14. Cause: Key Press Button B(1)
15. Verify: Display Screen(7, '\$---.-')
16. Cause: Key Press Digit (2)
17. Verify: Display Screen(7, '\$---.-2')
18. Cause: Key Press Digit (5)
19. Verify: Display Screen(7, '\$---.25')
20. Cause: Key Press Digit (0)
21. Verify: Display Screen(7, '\$--2.50')
22. Cause: Key Press Digit (0)
23. Verify: Display Screen(7, '\$-25.00')
24. Verify: Display Screen(13, null)
25. Verify: Open Door(deposit)
26. Cause: Insert Deposit Envelope
27. Verify: Close Door(deposit)
28. Verify: Display Screen(14, null)
29. Cause: Key Press Button B(2)
30. Verify: Print Receipt ('\$1025.00')
31. Verify: Display Screen(15, null)
32. Verify: Eject ATM Card
33. Verify: Display Screen(1, null)

Post Condition: Screen 1 displayed Test Result:

Pass  
 Fail



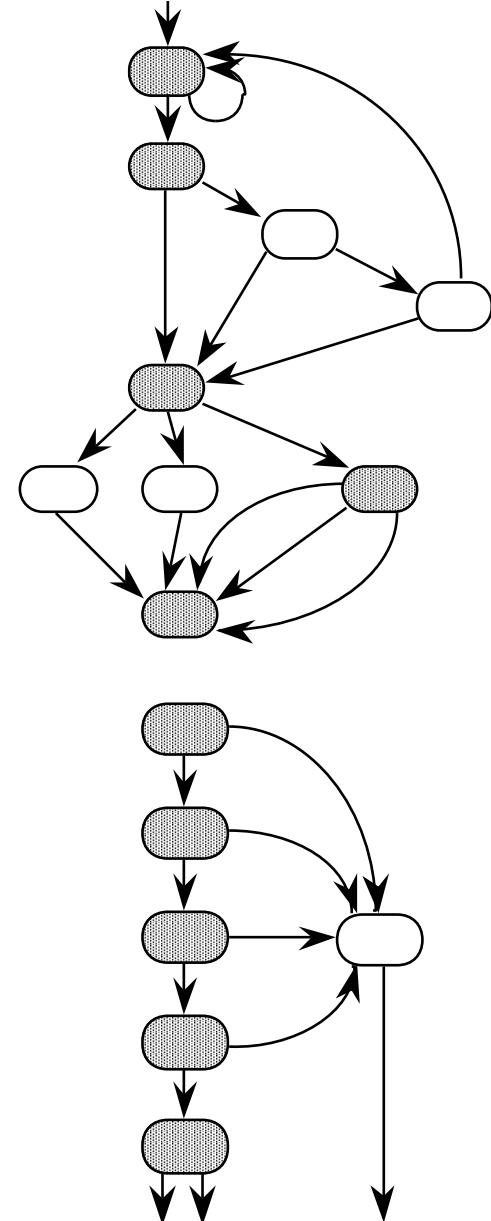
# Thread 5 Test Procedure

Description: valid card, a correct PIN entry attempt, followed by an attempt to withdraw more cash than the savings account balance.

1. Cause: Insert ATM Card 2
2. Verify: Display Screen(2, '---')
- 3,5,7,9. Cause: Key Press Digit (4,5,6,7)
- 4,6,8,10. Verify: Display Screen(2,'XXXX')
11. Verify: Display Screen(5, null)
12. Cause: Key Press Button B(3)
13. Verify: Display Screen(6, null)
14. Cause: Key Press Button B(2)
15. Verify: Display Screen(7, '\$---.-')
- 16,18,20,22,24 Cause: Key Press Digit (1,1,0,0,0)
- 17,19,21,23,25 Verify: Display Screen(7, '\$-110.00')
24. Verify: Display Screen(8, '---.-')
26. Cause: Key Press Cancel
28. Verify: Display Screen(14, null)
29. Cause: Key Press Button B(2)
30. Verify: Print Receipt ('\$90.00')
31. Verify: Eject ATM Card
32. Verify: Display Screen(1, null)

Post Condition: Screen 1 displayed Test Result:

Pass  
 Fail



# Operational Profiles

**Zipf's Law: 80% of the activities occur in 20% of the space**

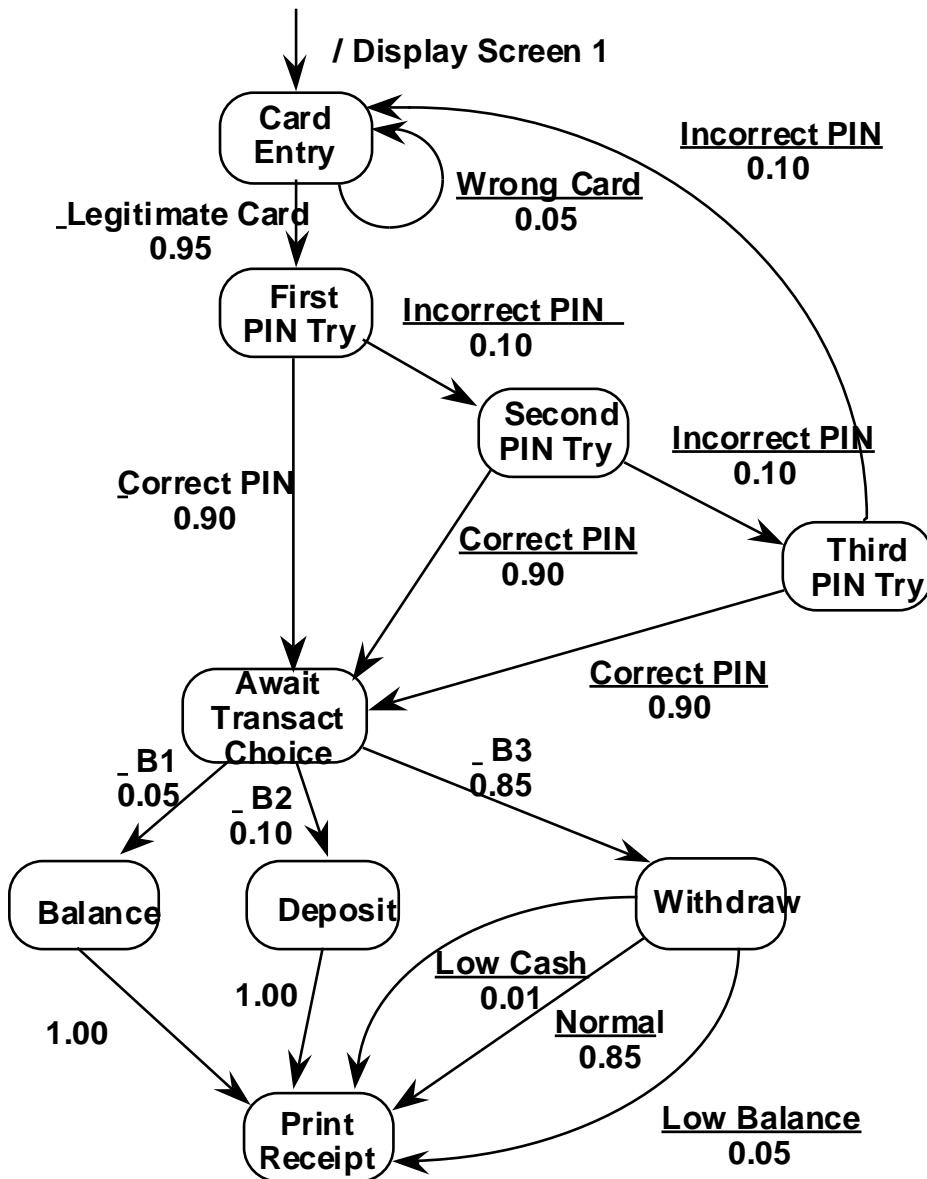
- **productions of a language syntax**
- **natural language vocabulary**
- **menu options of a commercial software package**
- **area of an office desktop**
- **floating point divide on the Pentium chip**

**For Threads:** a small fraction of all possible threads represents the majority of system execution time.

**Therefore:** find the occurrence probabilities of threads and use these to order thread testing.

The idea of operational profiles is to determine the execution frequencies of various threads and to use this information to select threads for system testing. Particularly when test time is limited (usually), operational profiles maximize the probability of finding faults by inducing failures in the most frequently traversed threads.

# Operational Profiles of SATM



Common Thread	Probabilities
Legitimate Card	0.95
PIN ok 1st try	0.90
Withdraw	0.85
Normal	0.85
	0.6177375

Rare Thread	Probabilities
Legitimate Card	0.95
Bad PIN 1st try	0.10
Bad PIN 2nd try	0.10
PIN ok 3rd try	0.90
Withdraw	0.85
Low Cash	0.01
	0.00072675

# SATM Atomic System Functions

**ASF1: Examine ATM Card**

Inputs: PAN from card, List of acceptable cards

Outputs: Legitimate Card, Wrong Card

**ASF2: Control PIN Entry**

Inputs: Expected PIN, Offered PIN

Outputs: PIN OK, Wrong PIN

**ASF3: Get Transaction Type**

Inputs: Button1, Button2, or Button3 depressed

Outputs: call Get Account Type (not externally visible)

**ASF4: Get Account Type**

Inputs: Button1 or Button2 depressed

Outputs: call one of Process Withdrawal, Process Deposit,  
or Display Balance (not externally visible)

**ASF5: Process Withdrawal**

Inputs: Amount Requested, Cash Available, Local Balance

Outputs: Process Request (call Dispense Cash)

Reject Request (insufficient funds or insufficient  
balance)

# SATM Atomic System Functions

## ASF6: Process Deposit

Inputs: Deposit Amount , Deposit Envelope, Deposit Door Status, Local Balance

Outputs: Process Request (call Credit Local Balance)  
Reject Request

## ASF7: Display Balance

Inputs: Local Balance

Outputs: (call Screen Handler)

## ASF8: Manage Session

Inputs: New Transaction Requested, Done

Outputs: (call Get Transaction Type or call Print Receipt)

## ASF9: Print Receipt

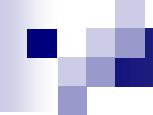
Inputs: Account Number, transaction type and amount,  
new local balance, time, date

Outputs: formatted text for receipt, (call Eject Card)

## ASF10: Eject Card

Inputs: (invoked)

Outputs: (control rollers)



# **"Hidden" Atomic System Functions**

**ASF11: Dispense \$10 Note**

**ASF12: Screen Handler**

**ASF13: Button Sensor**

**ASF14: Digit Keypad Sensor**

**ASF15: Cancel Sensor**

**ASF16: Card Roller Controller**

**ASF17: Control Deposit Door**

**ASF18: Control Deposit Rollers**

**ASF19: Control Cash Door**

**ASF20: Count Cash on Hand**

**ASF21: Timer**

# SATM Threads

**Thread 1: Wrong Card**

**State Sequence:** Idle, Idle

**Event Sequence:** Display Screen 1, Wrong Card

**ASF Sequence:**

**Thread 2: Wrong PIN**

**State Sequence:** Idle, Await 1st PIN Try, Await 2nd PIN Try, Await 3rd PIN Try, Idle

**Event Sequence:** Display Screen 1, Legitimate Card, Wrong PIN, Wrong PIN, Wrong PIN, Display Screen 4, Display Screen 1

**ASF Sequence:**

**Thread 3: Balance Inquiry**

**State Sequence:** Idle, Await 1st PIN Try, Acquire Transaction Data, Display Balance, Display Balance, Close Session, Idle

**Event Sequence:** Display Screen 1, Legitimate Card, Wrong PIN, Wrong PIN, Wrong PIN, Display Screen 4, Display Screen 1

**ASF Sequence:**

# SATM Threads

**Thread 4: Balance then Deposit then Withdraw**

**State Sequence:** Idle, Await 1st PIN Try, Acquire Transaction Data, Display Balance, Close Session, Acquire Transaction Data, Process Deposit, Close Session, Acquire Transaction Data, Process Withdrawal, Close Session, Idle

**Event Sequence:** Display Screen 1, Legitimate Card, Wrong PIN, Wrong PIN, Wrong PIN, Display Screen 4, Display Screen 1

**ASF Sequence:**

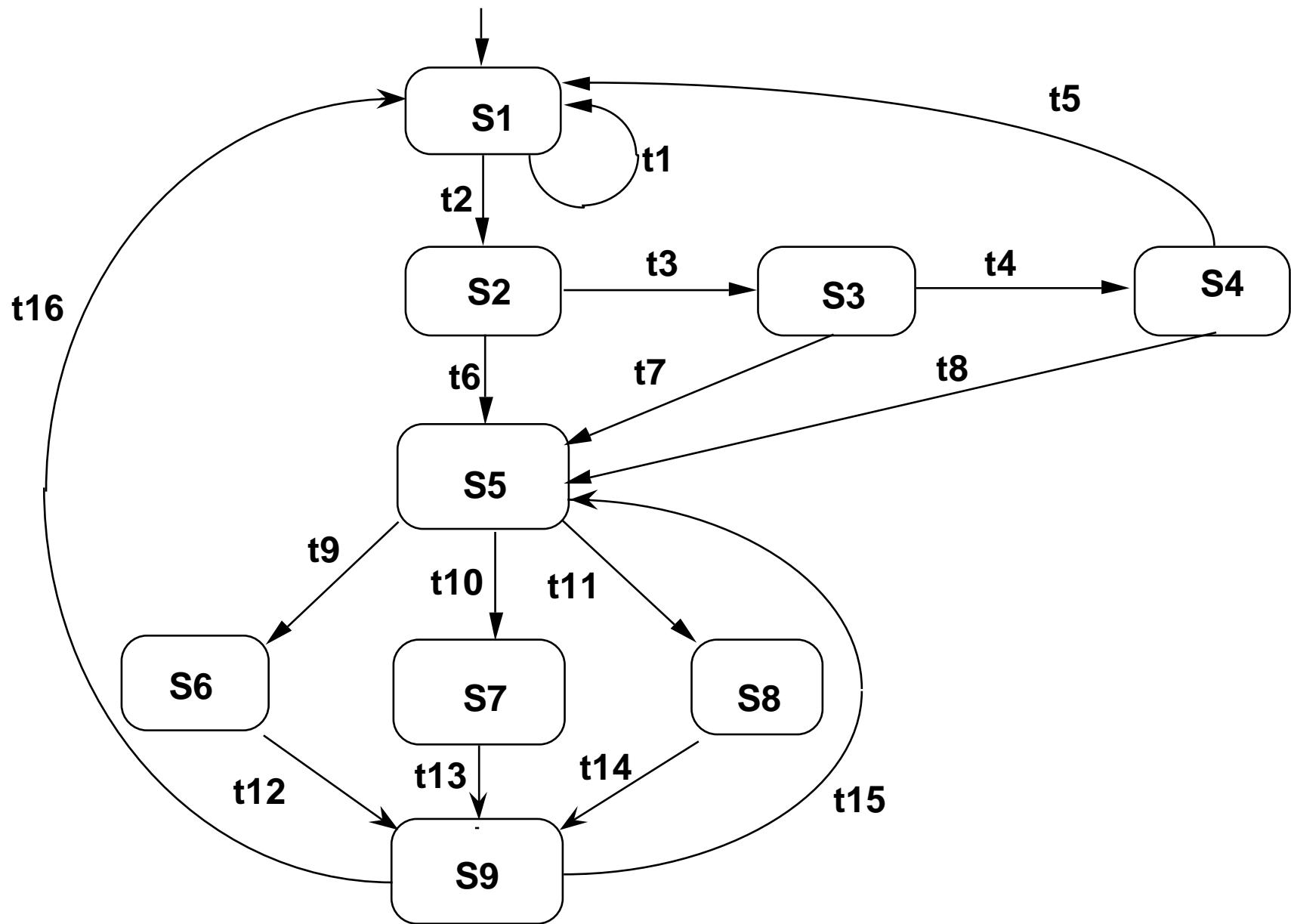
**Thread 5: 3 PIN Tries then Balance then Deposit then Withdraw**

**State Sequence:** Idle, Await 1st PIN Try, Await 2nd PIN Try, Await 3rd PIN Try, Acquire Transaction Data, Display Balance, Close Session, Acquire Transaction Data, Process Deposit, Close Session, Acquire Transaction Data, Process Withdrawal, Close Session, Idle

**Event Sequence:** Display Screen 1, Legitimate Card, Wrong PIN, Wrong PIN, Wrong PIN, Display Screen 4, Display Screen 1

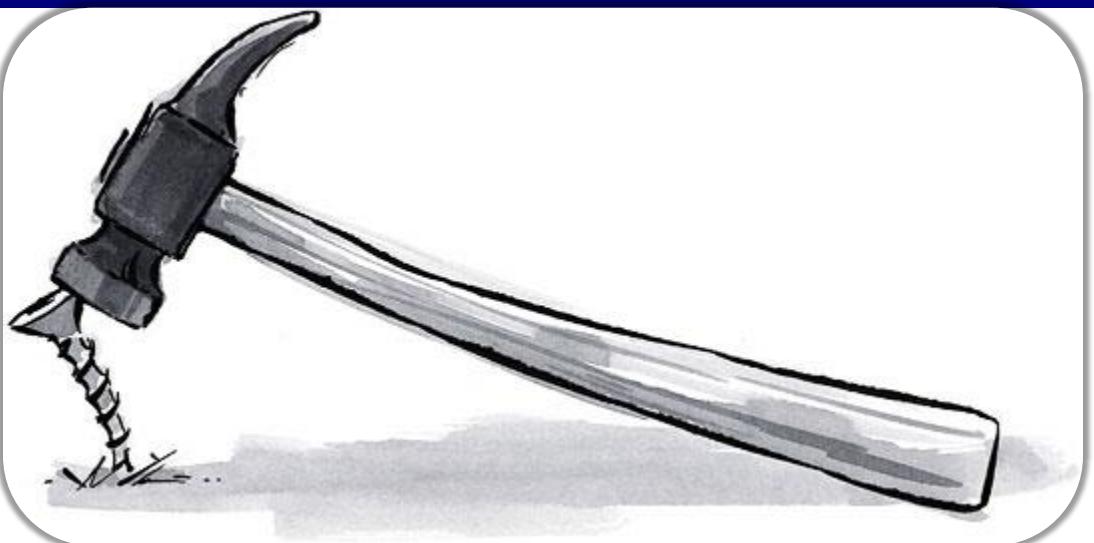
**ASF Sequence:**

# ATM States and Transitions



# Non-functional Testing

Testing Non-functional Software Characteristics



# Testing the System Attributes

- "How well" or with what quality the system should carry out its function
- Attributive characteristics:
  - Reliability
  - Usability
  - Efficiency



# Testability of Requirements

- Nonfunctional requirements are often *not clearly defined*
- How would you test:
  - “The system should be easy to operate”
  - “The system should be fast”
- Requirements should be expressed in a *testable way*
  - Make sure every requirement is testable
    - Make it early in the development process

# Nonfunctional Tests

- Performance test
  - Processing speed and response time
- Load test
  - Behavior in increasing system loads
    - Number of simultaneous users
    - Number of transactions
- Stress test
  - Behavior when overloaded

# Nonfunctional Tests (2)

data

- Testing of security
  - Against unauthorized access
  - Service attacks
- Stability
  - Mean time between failures
  - Failure rate with a given user profile
  - Etc.



# Nonfunctional Tests (3)

- Robustness test
  - Response
  - Examination of exception handling and recovery to errors
- Compatibility and data conversion
  - Compatibility to given systems
  - Import/export of data



# Nonfunctional Tests (4)

- Different configurations of the system
  - Back-to-back testing
- Usability test
  - Ease of learning the system
  - Ease and efficiency of operation
  - Understandability of the system



# Regression Testing

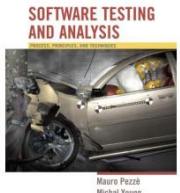
- Regression testing ensures that no new faults are introduced when the system is modified, expanded or enhanced as well as when the previously uncovered faults are corrected.
- Procedure:
  - Insert / modify code
  - Test the directly affected functions
  - Execute all available tests to make sure that nothing else got broken

# Test Case Prioritization

- Prioritization is necessary:
  - during regression testing
  - when there is not enough time or resources to run the entire test suite
  - when there is a need to decide which test cases to run first, i.e. which give the biggest bang for the buck
- Prioritization can be based on:
  - which test cases are likely to find the most faults
  - which test cases are likely to find the most severe faults
  - which test cases are likely to cover the most statements



# System, Acceptance, and Regression Testing



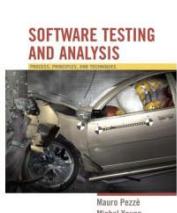
---

(c) 2007 Mauro Pezzè & Michal Young

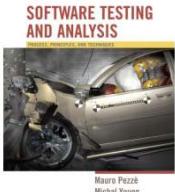
Ch 22, slide 1

# Learning objectives

- Distinguish system and acceptance testing
  - How and why they differ from each other and from unit and integration testing
- Understand basic approaches for quantitative assessment (reliability, performance, ...)
- Understand interplay of validation and verification for usability and accessibility
  - How to continuously monitor usability from early design to delivery
- Understand basic regression testing approaches
  - Preventing accidental changes

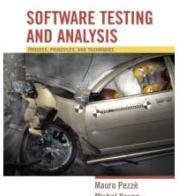


	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
Verification		<i>Validation</i>	Verification



22.2

## ➤ System testing



---

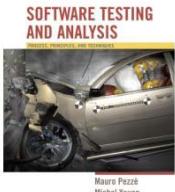
(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 4

# System Testing

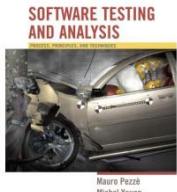
- Key characteristics:
  - Comprehensive (the whole system, the whole spec)
  - Based on specification of observable behavior
    - Verification against a requirements specification, not validation, and not opinions
  - Independent of design and implementation

*Independence:* Avoid repeating software design errors in system test design



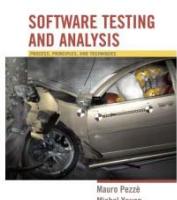
# Independent V&V

- *One strategy for maximizing independence:* System (and acceptance) test performed by a different organization
  - Organizationally isolated from developers (no pressure to say “ok”)
  - Sometimes outsourced to another company or agency
    - Especially for critical systems
    - Outsourcing for independent judgment, not to save money
    - May be *additional* system test, not replacing internal V&V
  - Not all outsourced testing is IV&V
    - Not *independent* if controlled by development organization



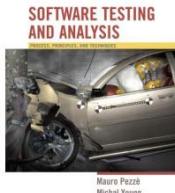
# Independence without changing staff

- If the development organization controls system testing ...
  - Perfect independence may be unattainable, but we can reduce undue influence
- Develop system test cases early
  - As part of requirements specification, before major design decisions have been made
    - Agile “test first” and conventional “V model” are both examples of designing system test cases before designing the implementation
    - An opportunity for “design for test”: Structure system for critical system testing early in project



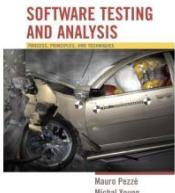
# Incremental System Testing

- System tests are often used to measure progress
  - System test suite covers all features and scenarios of use
  - As project progresses, the system passes more and more system tests
- Assumes a “threaded” incremental build plan:  
Features exposed at top level as they are developed



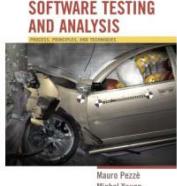
# Global Properties

- Some system properties are inherently global
  - Performance, latency, reliability, ...
  - Early and incremental testing is still necessary, but provide only estimates
- A major focus of system testing
  - The only opportunity to verify global properties against actual system specifications
  - Especially to find unanticipated effects, e.g., an unexpected performance bottleneck



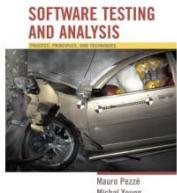
# Context-Dependent Properties

- Beyond system-global: Some properties depend on the system context and use
  - Example: Performance properties depend on environment and configuration
  - Example: Privacy depends both on system and how it is used
    - Medical records system must protect against unauthorized use, and authorization must be provided only as needed
  - Example: Security depends on threat profiles
    - And threats change!
- Testing is just one part of the approach



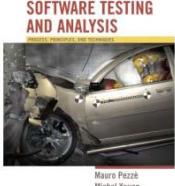
# Establishing an Operational Envelope

- When a property (e.g., performance or real-time response) is parameterized by use ...
  - requests per second, size of database, ...
- Extensive stress testing is required
  - varying parameters within the envelope, near the bounds, and beyond
- Goal: A well-understood model of how the property varies with the parameter
  - How sensitive is the property to the parameter?
  - Where is the “edge of the envelope”?
  - What can we expect when the envelope is exceeded?



# Stress Testing

- Often requires extensive simulation of the execution environment
  - With systematic variation: What happens when we push the parameters? What if the number of users or requests is 10 times more, or 1000 times more?
- Often requires more resources (human and machine) than typical test cases
  - Separate from regular feature tests
  - Run less often, with more manual control
  - Diagnose deviations from expectation
    - Which may include difficult debugging of latent faults!



## Capacity Testing

- **When:** systems that are intended to cope with high volumes of data should have their limits tested and we should consider how they fail when capacity is exceeded
- **What/How:** usually we will construct a harness that is capable of generating a very large volume of simulated data that will test the capacity of the system or use existing records
- **Why:** we are concerned to ensure that the system is fit for purpose say ensuring that a medical records system can cope with records for all people in the UK (for example)
- **Strengths:** provides some confidence the system is capable of handling high capacity
- **Weaknesses:** simulated data can be unrepresentative; can be difficult to create representative tests; can take a long time to run

# Security Testing

- **When:** most systems that are open to the outside world and have a function that should not be disrupted require some kind of security test. Usually we are concerned to thwart malicious users.
- **What/How:** there are a range of approaches. One is to use league tables of bugs/errors to check and review the code (e.g. SANS top twenty-five security-related programming errors). We might also form a team that attempts to break/break into the system.
- **Why:** some systems are essential and need to keep running, e.g. the telephone system, some systems need to be secure to maintain reputation.
- **Strengths:** this is the best approach we have most of the effort should go into design and the use of known secure components.
- **Weaknesses:** we only cover known ways in using checklists and we do not take account of novelty using a team to try to break does introduce this.

## Performance Testing

- **When:** many systems are required to meet performance targets laid down in a service level agreement (e.g. does your ISP give you 2Mb/s download?).
- **What/How:** there are two approaches - modelling/simulation, and direct test in a simulated environment (or in the real environment).
- **Why:** often a company charges for a particular level of service - this may be disputed if the company fails to deliver. E.g. the VISA payments system guarantees 5s authorisation time delivers faster and has low variance. Customers would be unhappy with less.
- **Strengths:** can provide good evidence of the performance of the system, modelling can identify bottlenecks and problems.
- **Weaknesses:** issues with how representative tests are.

## Compliance Testing

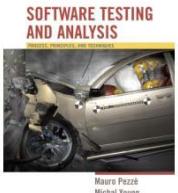
- **When:** we are selling into a regulated market and to sell we need to show compliance. E.g. if we have a C compiler we should be able to show it correctly compiles ANSI C.
- **What/How:** often there will be standardised test sets that constitute good coverage of the behaviour of the system (e.g. a set of C programs, and the results of running them).
- **Why:** we can identify the problem areas and create tests to check that set of conditions.
- **Strengths:** regulation shares the cost of tests across many organisations so we can develop a very capable test set.
- **Weaknesses:** there is a tendency for software producers to orient towards the compliance test set and do much worse on things outside the compliance test set.

## Documentation Testing

- **When:** most systems that have documentation should have it tested and should be tested against the real system. Some systems embed test cases in the documentation and using the doc tests is an essential part of a new release.
- **What/How:** test set is maintained that verifies the doc set matches the system behaviour. Could also just get someone to do the tutorial and point out the errors.
- **Why:** the user gets really confused if the system does not conform to the documentation.
- **Strengths:** ensures consistency.
- **Weaknesses:** not particularly good on checking consistency of narrative rather than examples.

22.3

## ➤ Acceptance testing



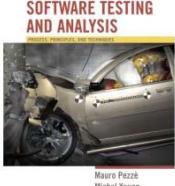
---

(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 13

# Estimating Dependability

- Measuring quality, not searching for faults
  - Fundamentally different goal than systematic testing
- Quantitative dependability goals are statistical
  - Reliability
  - Availability
  - Mean time to failure
  - ...
- Requires valid statistical samples from *operational profile*
  - Fundamentally different from systematic testing



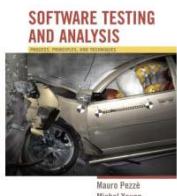
# Definitions

---

- **Reliability:** Survival Probability
  - When function is critical during the mission time.
- **Availability:** The fraction of time a system meets its specification.
  - Good when continuous service is important but it can be delayed or denied
- **Failsafe:** System fails to a known safe state
- **Dependability:** Generalisation - System does the right thing at right time

# Statistical Sampling

- We need a valid *operational profile* (model)
  - Sometimes from an older version of the system
  - Sometimes from operational environment (e.g., for an embedded controller)
  - *Sensitivity testing* reveals which parameters are most important, and which can be rough guesses
- And a clear, precise definition of what is being measured
  - Failure rate? Per session, per hour, per operation?
- And many, many random samples
  - Especially for high reliability measures



# System Reliability

---

- The reliability,  $R_F(t)$  of a system is the probability that no fault of the class F occurs (i.e. system survives) during time t.

$$R_F(t) = P(t_{init} \leq t < t_f \forall f \in F)$$

where  $t_{init}$  is time of introduction of the system to service,  
 $t_f$  is time of occurrence of the first failure f drawn from F.

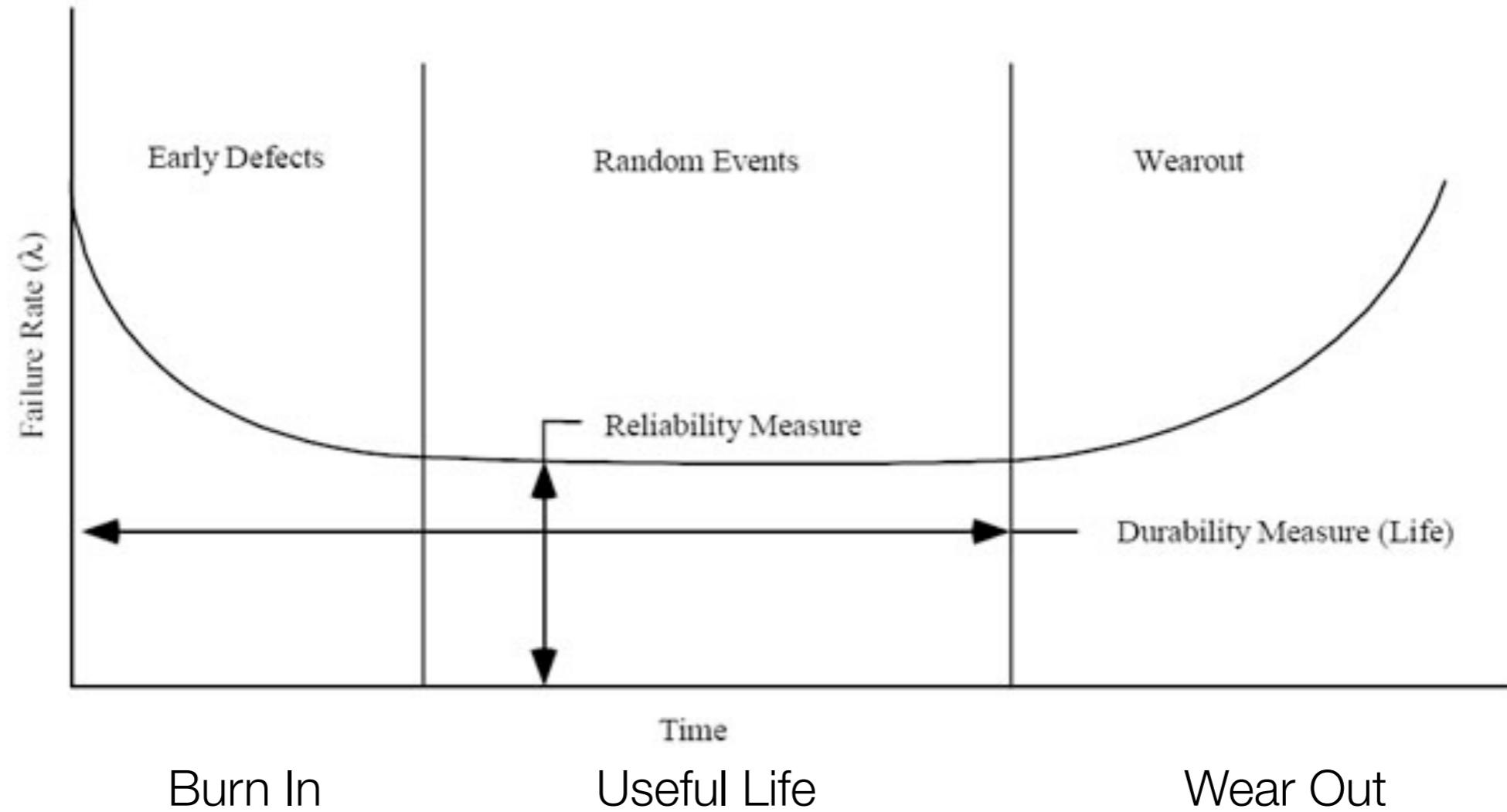
- Failure Probability,  $Q_F(t)$  is complementary to  $R_F(t)$

$$R_F(t) + Q_F(t) = 1$$

- We can take off the F subscript from  $R_F(t)$  and  $Q_F(t)$

- When the lifetime of a system is exponentially distributed, the reliability of the system is:  $R(t) = e^{-\lambda t}$  where the parameter  $\lambda$  is called the failure rate

# Component Reliability Model



During useful life, components exhibit a constant failure rate  $\lambda$ . Reliability of a device can be modelled using an exponential distribution  $R(t) = e^{-\lambda t}$

# Component Failure Rate

---

- Failure rates often expressed in failures / million operating hours

Automotive Embedded System Component	Failure Rate $\lambda$
Military Microprocessor	0.022
Typical Automotive Microprocessor	0.12
Electric Motor Lead/Acid battery	16.9
Oil Pump	37.3
Automotive Wiring Harness (luxury)	775

# MTTF: Mean Time To Failure

---

- **MTTF:** Mean Time to Failure or Expected Life
- **MTTF:** Mean Time To (first) Failure is defined as the expected value of  $t_f$

$$MTTF = E(t_f) = \int_0^{\infty} R(t)dt = \frac{1}{\lambda}$$

where  $\lambda$  is the failure rate.

- **MTTF** of a system is the expected time of the first failure in a sample of identical initially perfect systems.
- **MTTR:** Mean Time To Repair is defined as the expected time for repair.
- **MTBF:** Mean Time Between Failure

# Serial System Reliability

---

- Serially Connected Components
- $R_k(t)$  is the reliability of a single component k:  $R_k(t) = e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.
- The overall system reliability  $R_{ser}(t)$

$$R_{ser}(t) = R_1(t) \times R_2(t) \times R_3(t) \times \dots \times R_n(t)$$

$$R_{ser}(t) = \prod_{i=1}^n R_i(t)$$

- No redundancy: Overall system reliability depends on the proper working of each component

$$R_{ser}(t) = e^{-t(\sum_{i=1}^n \lambda_i)}$$

- Serial failure rate

$$\lambda_{ser} = \sum_{i=1}^n \lambda_i$$

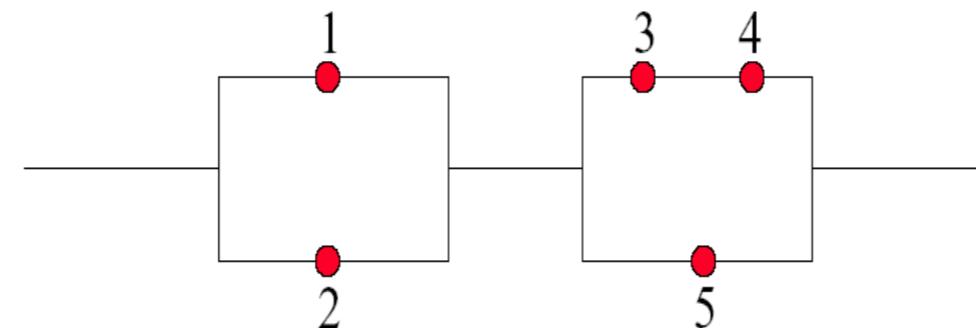
# System Reliability

---

- Building a reliable serial system is extraordinarily difficult and expensive.
- For example: if one is to build a serial system with 100 components each of which had a reliability of 0.999, the overall system reliability would be

$$0.999^{100} = 0.905$$

- Reliability of System of Components



- Minimal Path Set:  
Minimal set of components whose functioning ensures the functioning of the system: {1,3,4} {2,3,4} {1,5} {2,5}

# Parallel System Reliability

---

- Parallel Connected Components
- $Q_k(t)$  is  $1 - R_k(t)$ :  $Q_k(t) = 1 - e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.

$$Q_{par}(t) = \prod_{i=1}^n Q_i(t)$$

- Overall system reliability:  $R_{par}(t) = 1 - \prod_{i=1}^n (1 - R_i(t))$

# Example

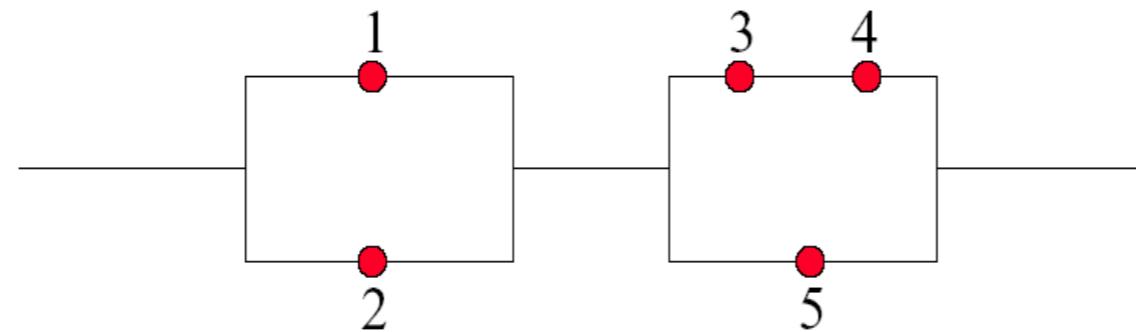
---

- Consider 4 identical modules are connected in parallel
- System will operate correctly provided at least one module is operational. If the reliability of each module is 0.95.
- The overall system reliability is  $1 - (1 - 0.95)^4 = 0.99999375$

# Parallel-Serial Reliability

---

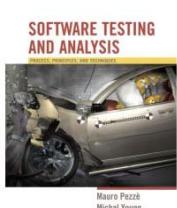
- Parallel and Serial Connected Components



- Total reliability is the reliability of the first half, in serial with the second half.
- Given  $R_1=0.9$ ,  $R_2=0.9$ ,  $R_3=0.99$ ,  $R_4=0.99$ ,  $R_5=0.87$
- $R_t = (1 - (1 - 0.9)(1 - 0.9))(1 - (1 - 0.87)(1 - (0.99 \times 0.99))) = 0.987$

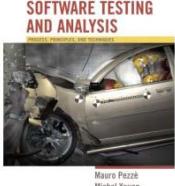
# Is Statistical Testing Worthwhile?

- Necessary for ...
  - Critical systems (safety critical, infrastructure, ...)
- But difficult or impossible when ...
  - Operational profile is unavailable or just a guess
    - Often for new functionality involving human interaction
      - But we may factor critical functions from overall use to obtain a good model of only the critical properties
  - Reliability requirement is very high
    - Required sample size (number of test cases) might require years of test execution
    - Ultra-reliability can seldom be demonstrated by testing



# Process-based Measures

- Less rigorous than statistical testing
  - Based on similarity with prior projects
- System testing process
  - Expected history of bugs found and resolved
- Alpha, beta testing
  - Alpha testing: Real users, controlled environment
  - Beta testing: Real users, real (uncontrolled) environment
  - May statistically sample users rather than uses
  - Expected history of bug reports



## Usability Testing

- **When:** where the system has a significant user interface and it is important to avoid user error — e.g. this could be a critical application e.g. cockpit design in an aircraft or a consumer product that we want to be an enjoyable system to use or we might be considering efficiency (e.g. call-centre software).
- **What/How:** we could construct a simulator in the case of embedded systems or we could just have many users try the system in a controlled environment. We need to structure the test with clear objectives (e.g. to reduce decision time,...) and have good means of collecting and analysing data.
- **Why:** there may be safety issues, we may want to produce something more useable than competitors' products...
- **Strengths:** in well-defined contexts this can provide very good feedback – often underpinned by some theory e.g. estimates of cognitive load.
- **Weaknesses:** some usability requirements are hard to express and to test, it is possible to test extensively and then not know what to do with the data.

## Reliability Testing

- **When:** we may want to guarantee some system will only fail very infrequently (e.g. nuclear power control software we might claim no more than one failure in 10,000 hours of operation). This is particularly important in telecommunications.
- **What/How:** we need to create a representative test set and gather enough information to support a statistical claim (system structured modelling supports demonstrating how overall failure rate relates to component failure rate).
- **Why:** we often need to make guarantees about reliability in order to satisfy a regulator or we might know that the market leader has a certain reliability that the market expects.
- **Strengths:** if the test data is representative this can make accurate predictions.
- **Weaknesses:** we need a lot of data for high-reliability systems, it is easy to be optimistic.

## Availability/Reparability Testing

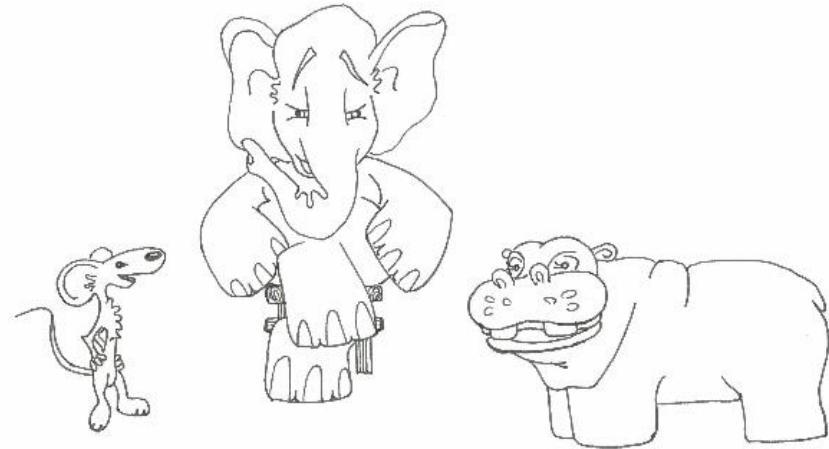
- **When:** we are interested in avoiding long down times we are interested in how often failure occurs and how long it takes to get going again. Usually this is in the context of a service supplier and this is a Key Performance Indicator.
- **What/How:** similar to reliability testing – but here we might seed errors or cause component failures and see how long they take to fix or how soon the system can return once a component is repaired.
- **Why:** in providing a critical service we may not want long interruptions (e.g. 999 service).
- **Strengths:** similar to reliability.
- **Weaknesses:** similar to reliability – in the field it may be much faster to fix common problems because of learning.

## Summary

- There are a very wide range of potential tests that should be applied to a system.
- Not all systems require all tests.
- Managing the test sets and when they should be applied is a very complex task.
- The quality of test sets is critical to the quality of a running implementation.

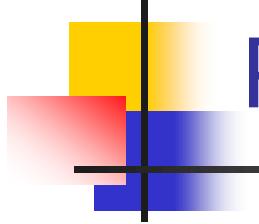
# Foundations of Software Testing

## Chapter 5: Test Selection, Minimization, and Prioritization for Regression Testing



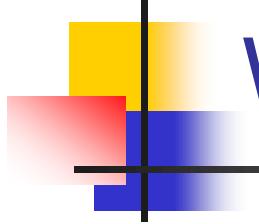
What is regression testing?





# Regression testing

Version 1	Version 2
1. Develop $P$	4. Modify $P$ to $P'$
2. Test $P$	5. Test $P'$ for new functionality
3. Release $P$	6. Perform regression testing on $P'$ to ensure that the code carried over from $P$ behaves correctly
	7. Release $P'$



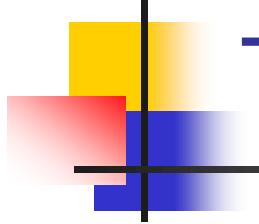
# What tests to use?

## TEST-ALL:

All valid tests from the previous version and new tests created to test any added functionality.

The TEST-ALL approach is best when you want to be certain that the new version works on all tests developed for the previous version and any new tests.

But what if you have limited resources to run tests and have to meet a deadline?



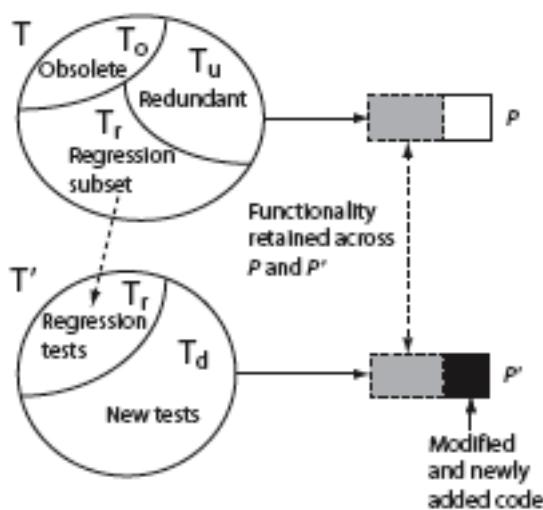
# Test selection

Select a subset  $T_r$  of the original test set  $T$  such that successful execution of the modified code  $P'$  against  $T_r$  implies that all the functionality carried over from the original code  $P$  to  $P'$  is intact.

$T_r$  can be found using several methods.

- test minimization
- test prioritization

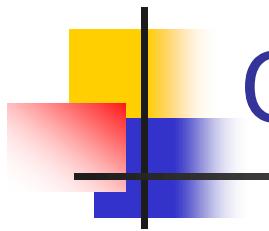
# Regression Test Selection problem



Given test set  $T$ , our goal is to determine  $T_r$  such that successful execution of  $P'$  against  $T_r$  implies that modified or newly added code in  $P'$  has not broken the code carried over from  $P$ .

Some tests might become obsolete when  $P$  is modified to  $P'$ . Such tests are not included in the regression subset  $T_r$ .

Test selection using execution trace  
and execution slice



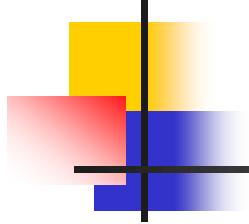
# Overview of a test selection method

**Step 1:** Given  $P$  and test set  $T$ , find the **execution trace** of  $P$  for each test in  $T$ .

**Step 2:** Extract the **test vector --  $\text{test}(n)$**  -- from the execution traces for each node  $n$  in the CFG of  $P$ .

**Step 3:** Build **syntax trees** for each node in the CFGs of  $P$  and  $P'$ .

**Step 4:** Traverse the CFGs and determine a subset of  $T$  appropriate for regression testing of  $P'$ .



# Execution Trace

Let  $G=(N, E)$  denote the CFG of program P.

N is a finite set of nodes and E a finite set of edges.

Suppose that nodes in N are numbered 1, 2, and so on, and that Start and End are two special nodes.

Let  $T_{no}$  be the set of all valid tests for  $P'$ .

$T_{no}$  is obtained by discarding all tests that have become **obsolete**.

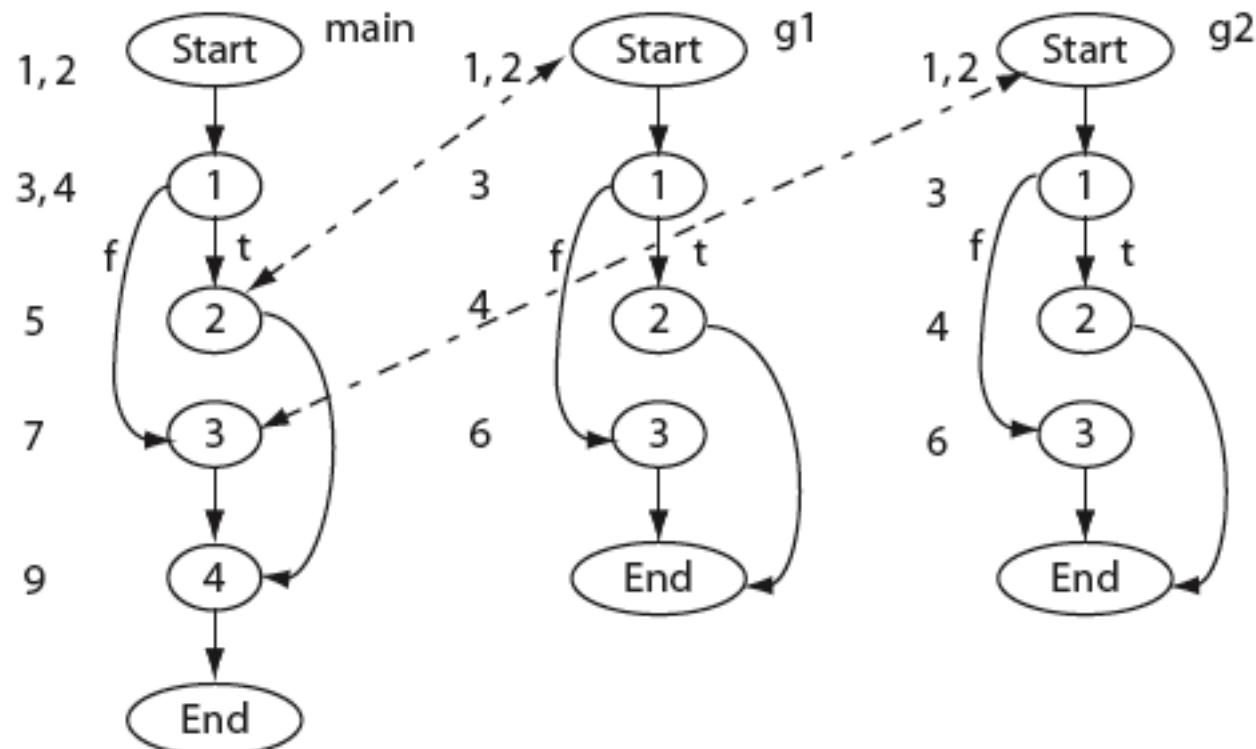
# Execution Trace

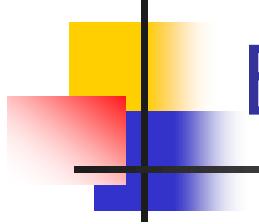
An execution trace of program P for some test t in T<sub>no</sub> is the sequence of nodes in G traversed when P is executed against t.  
As an example, consider the following program.

```
1 main(){           1 int g1(int a, b){ 1 int g2 (int a, b){  
2   int x,y,p;     2   int a,b;          2   int a,b;  
3   input (x,y);  3   if(a+ 1==b)        3   if(a==(b+1))  
4   if (x<y)       4   return(a*a);    4   return(b*b);  
5   p=g1(x,y);   5   else             5   else  
6   else           6   return(b*b);    6   return(a*a);  
7   p=g2(x,y);   7 }                  7 }  
8 endif  
9 output (p);  
10 end  
11 }
```

# Execution Trace

Here is a CFG for our example program.





# Execution Trace

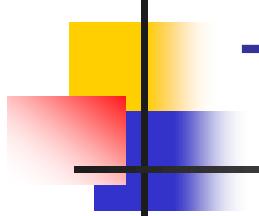
Consider the following set of three tests and the corresponding trace.

$$T = \left\{ \begin{array}{l} t_1 : \langle x = 1, y = 3 \rangle \\ t_2 : \langle x = 2, y = 1 \rangle \\ t_3 : \langle x = 3, y = 1 \rangle \end{array} \right\}$$

---

Test ( $t$ )	Execution trace ( $trace(t)$ )
$t_1$	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
$t_2$	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
$t_3$	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

---



# Test vector

A test vector for node  $n$ , denoted by  $\text{test}(n)$ , is the set of tests that traverse node  $n$  in the CFG. For program  $P$  we obtain the following test vectors.

---

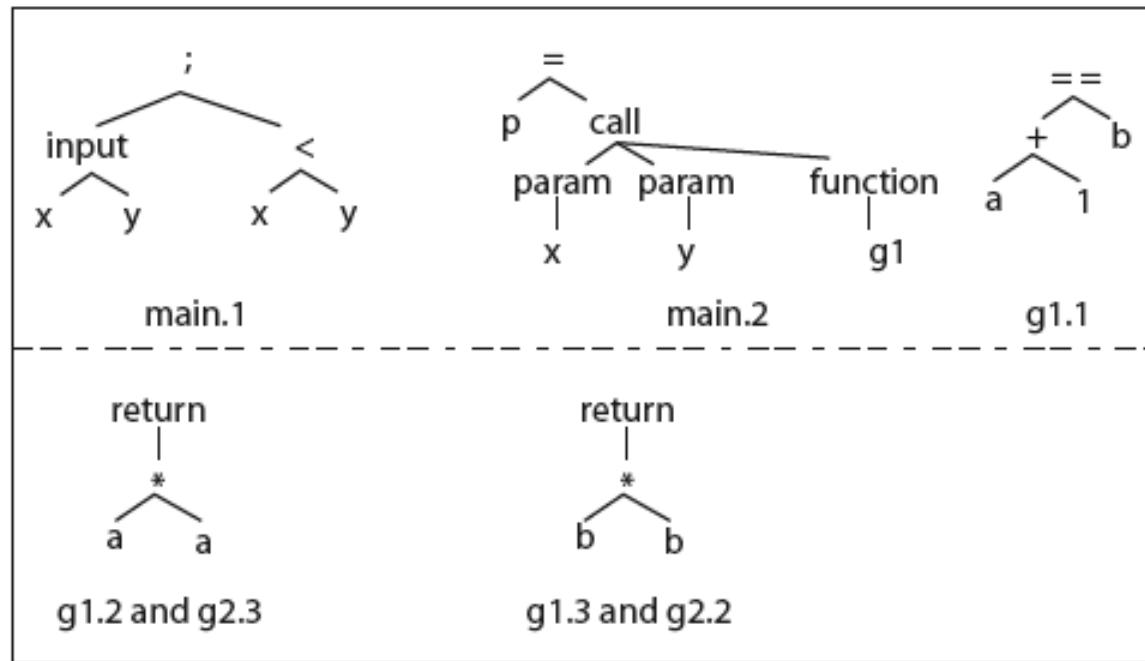
Test vector ( $\text{test}(n)$ ) for node $n$				
Function	1	2	3	4
main	$t_1, t_2, t_3$	$t_1, t_3$	$t_2$	$t_1, t_2, t_3$
g1	$t_1, t_3$	$t_3$	$t_1$	—
g2	$t_2$	$t_2$	None	—

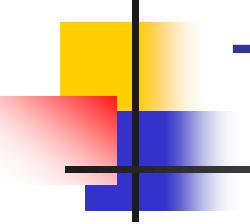
# Syntax trees

A syntax tree is built for each node of  $\text{CFG}(P)$  and  $\text{CFG}(P')$ .

Recall that each node represents a basic block.

Here are sample syntax trees for the example program.

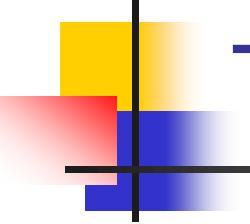




# Test selection

Given the execution traces and the CFGs for  $P$  and  $P'$ , the following three steps are executed to obtain a subset  $T'$  of  $T$  for regression testing of  $P'$ .

- Step 1 Set  $T' = \emptyset$ . Unmark all nodes in  $G$  and in its child CFGs.
- Step 2 Call procedure `SelectTests` ( $G$ .Start,  $G'$ .Start'), where  $G$ .Start and  $G'$ .Start' are, respectively, the start nodes in  $G$  and  $G'$ .
- Step 3  $T'$  is the desired test set for regression testing  $P'$ .

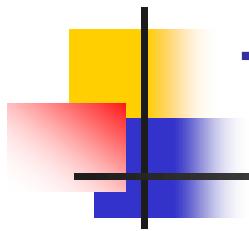


# Test selection

The idea underlying procedure **SelectTests** is to traverse the two CFGs from their respective START nodes using a recursive descent procedure.

The descent proceeds in parallel and corresponding nodes are compared.

If two corresponding nodes  $n$  in  $\text{CFG}(P)$  and  $n'$  in  $\text{CFG}(P')$  are found to be syntactically different, all tests in  $\text{test}(n)$  are added to  $T'$ .



# Test selection example

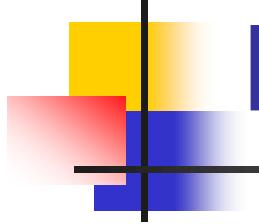
Suppose that function g1 in P is modified as follows.

```
1 int g1(int a, b){ ← Modified g1.  
2 int a, b;  
3 if(a-1==b) ← Predicate modified.  
4   return(a*a),  
5 else  
6   return(b*b),  
7 }
```

Try the SelectTests algorithm and check if you get  $T' = \{t_1, t_3\}$ .

# Test selection using dynamic slicing





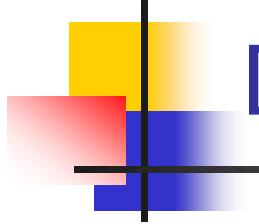
# Dynamic slice

Let  $L$  be a location in program  $P$  and  $v$  a variable used at location  $L$ .

Let  $\text{trace}(t)$  be the execution trace of  $P$  when executed against test  $t$ .

The dynamic slice of program  $P$  with respect to test  $t$ , variable  $v$ , and location  $L$ , denoted as  $\text{DS}(t, v, L)$ , is the set of statements in  $P$  that lie in  $\text{trace}(t)$  and affect the value of  $v$  at location  $L$ .

*Question: What is the dynamic slice of  $P$  with respect to  $v$ ,  $t$ , and  $L$ , if  $L$  is not in  $\text{trace}(t)$ ?*



# Dynamic dependence graph (DDG)

The DDG is needed to obtain a dynamic slice.  
Here is how a DDG G is built.

**Step 1:** Initialize G with a node for each declaration.

There are no edges among these nodes.

**Step 2:** Add to G the first node in  $\text{trace}(t)$ .

**Step 3:** For each successive statement in  $\text{trace}(t)$ , add to G a new node n. Then add *control and data dependence* edges from n to the existing nodes in G.

[Recall from Chapter 1 the definitions of control and data dependence edges.]

# Construction of a DDG: Example [1]

```
1 input (x, y);      t: <x=2, y=4>
2 while (x < y){
3   if (f1(x)== 0)
4     z=f2(x);
      else
5     z=f3(x);
6   x=f4(x);
7   w=f5(z);
}
8 output (w)
end
```

Assume successive values of x to be 2, 0, and 5 and for these f1(x) is 0, 2, and 3 respectively.

trace(t)={1, 2, 3, 4, 6, 7, 2', 3', 5, 6', 7', 2'', 8}

For simplicity, we ignore declarations.

Add a node to G corresponding to statement 1.

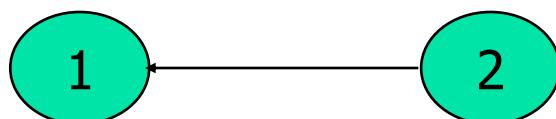
1

# Construction of a DDG: Example [2]

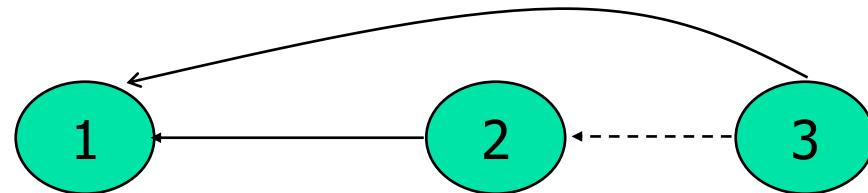
```
1 input (x, y);
2 while (x < y){
3   if (f1(x)== 0)
4     z=f2(x);
      else
5     z=f3(x);
6   x=f4(x);
7   w=f5(z);
}
8 output (w)
end
```

trace(t)={1, 2, 3, 4, 6, 7, 2, 3, 5, 6, 7, 2, 8}

Add another node corresponding to statement 2 in trace(t). Also add a data dependence edge from 2 to 1 as statement 2 is data dependent on statement 1.



Add yet another node corresponding to statement 3 in trace(t). Also add a data dependence edge from node 3 to node 1 as statement 3 is data dependent on statement 1 and a control edge from node 3 to 2.

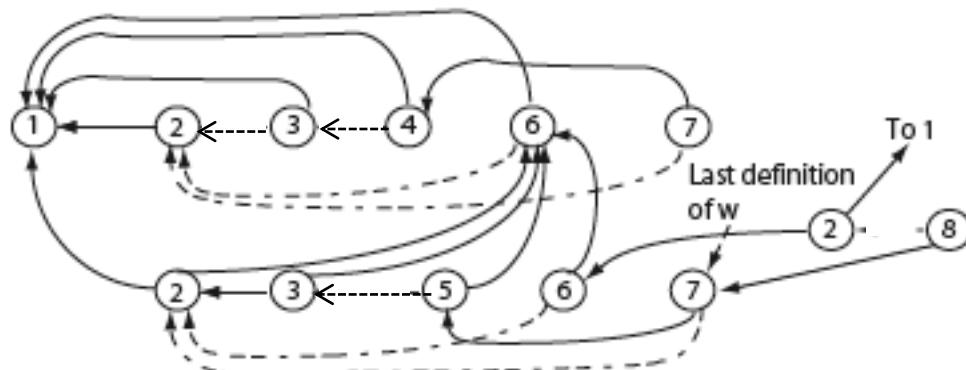


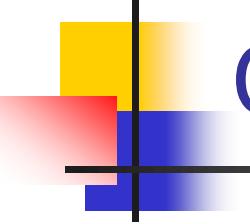
# Construction of a DDG: Example [3]

```
1 input (x, y);
2 while (x < y){
3   if (f1(x)== 0)
4     z=f2(x);
      else
5     z=f3(x);
6   x=f4(x);
7   w=f5(z);
}
8 output (w)
end
```

$\text{trace}(t)=\{1, 2, 3, 4, 6, 7, 2, 3, 5, 6, 7, 2, 8\}$

Continuing this way we obtain the following DDG for program P and  $\text{trace}(t)$ .





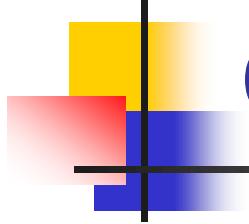
# Obtaining dynamic slice (DS)

Step 1: Execute P against test t and obtain  $\text{trace}(t)$ .

Step 2: Build the dynamic dependence graph  $G$  from  $P$  and  $\text{trace}(t)$ .

Step 3: Identify in  $G$  *node n* labeled L that contains the *last assignment to v*. If no such node exists then the dynamic slice is empty, otherwise execute Step 4.

Step 4: Find in  $G$  the set  $\text{DS}(t, v, n)$  of all *nodes reachable from n, including n itself*.  $\text{DS}(t, v, n)$  is the dynamic slice of  $P$  with respect to  $v$  at location L and test  $t$ .



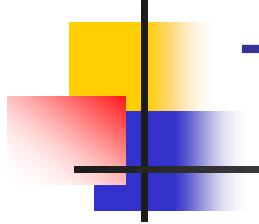
# Obtaining dynamic slice: Example

Suppose we want to compute the dynamic slice of P with respect to variable w at line 8 and test t shown earlier.

We already have the DDG of P for t.

First, identify the last definition of w in the DDG.  
This occurs at line 7 as marked.

Traverse the DDG backwards from node 7 and collect all nodes reachable from 7.  
This gives us the following dynamic slice: {1, 2, 3, 5, 6, 7}.



# Test selection using dynamic slice

Let  $T$  be the test set used to test  $P$ . Let  $P'$  be the modified program.

Let  $n_1, n_2, \dots, n_k$  be the nodes in the CFG of  $P$  modified to obtain  $P'$ .

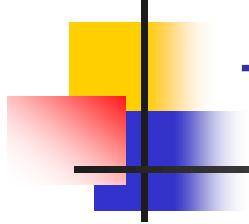
*Which tests from  $T$  should be used to obtain a regression test  $T'$  for  $P'$ ?*

Find  $DS(t)$  for  $P$ .

If any of the modified nodes is in  $DS(t)$ , then add  $t$  to  $T'$ .

Test selection using test minimization





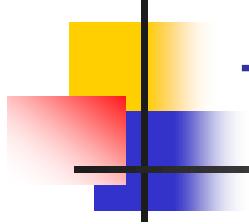
## Test minimization

Test minimization is yet another method for selecting tests for regression testing.

To illustrate test minimization, suppose that P contains two functions: main and f.

Now suppose that P is tested using test cases t1 and t2.

During testing it was observed that t1 causes the execution of main but not of f while t2 causes the execution of both main and f.



## Test minimization

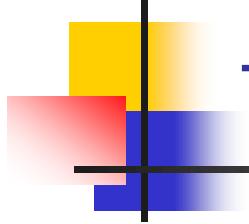
Now suppose that  $P'$  is obtained from  $P$  by making some modification to  $f$ .

*Which of the two test cases should be included in the regression test suite?*

Obviously there is no need to execute  $P'$  against  $t_1$  as it does not cause the execution of  $f$ .

Thus the regression test suite consists of only  $t_2$ .

In this example we have used function coverage to **minimize** a test suite  $\{t_1, t_2\}$  to obtain the regression test suite  $\{t_2\}$ .



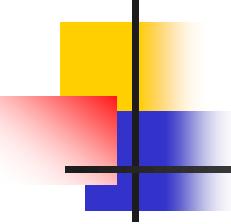
## Test minimization

Test minimization is based on the coverage of testable entities in P.

Testable entities include, for example:

- program statements
- decisions
- def-use chains
- mutants.

One uses the following procedure to minimize a test set based on a selected testable entity.



## A procedure for test minimization

Step 1: Identify the type of testable entity to be used for test minimization.

Let  $e_1, e_2, \dots, e_k$  be the  $k$  testable entities of type TE present in P.

In our previous example, TE is “function”.

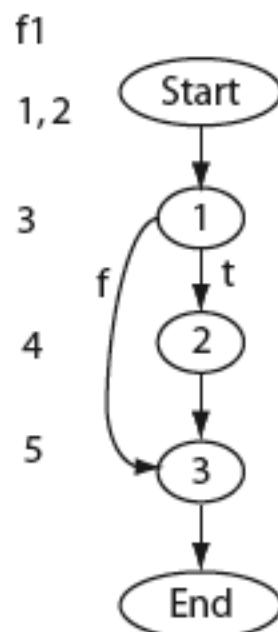
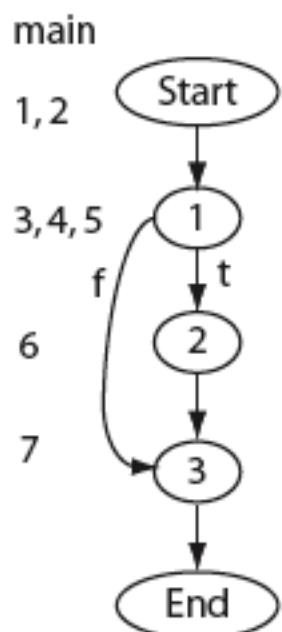
Step 2: Execute P against all elements of test set T and, for each test t in T,  
determine which of the  $k$  testable entities is covered.

Step 3: Find a minimal subset  $T'$  of T such that each testable entity is  
covered by at least one test in  $T'$ .

# Test minimization: Example

Step 1: Let the **basic block** be the testable entity of interest.

The basic blocks for a sample program are shown here for both main and function f1.



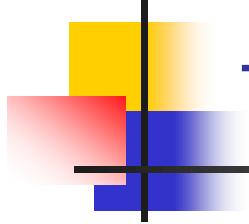
Step 2: Suppose the coverage of the basic blocks, when executed against three tests, is as follows:

t1: main: 1, 2, 3.    f1: 1, 3

t2: main: 1, 3.    f1: 1, 3

t3: main: 1, 3.    f1: 1, 2, 3

Step3: A minimal test set for regression testing is {t1, t3}.



## Test minimization

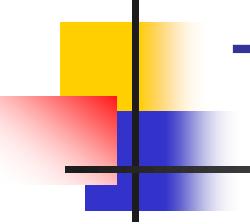
Minimal test set is not unique.

Test minimization is NP hard. Hitting set problem.

Various testable entities can be used for minimization – blocks, branch outcomes, uses etc.

Test selection using test prioritization





# Test prioritization

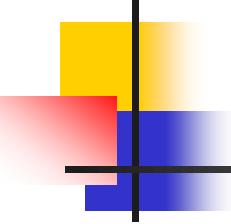
Test minimization discards test cases.

There is a chance that, if  $P'$  were executed against a discarded test case, it would reveal an error in the modification made.

When very high quality software is desired, it might not be wise to discard test cases as in test minimization.

In such cases one uses **test prioritization**.

Tests are prioritized based on some criteria. For example, tests that cover the maximum number of a selected testable entity could be given the highest priority, the one with the next highest coverage the next higher priority, and so on.



## A procedure for test prioritization

Step 1: Identify *the type of testable entity* to be used for test minimization.

Let  $e_1, e_2, \dots, e_k$  be the  $k$  testable entities of type TE present in P.

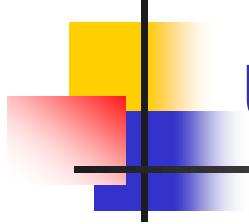
In our previous example TE is “function”.

Step 2: Execute P against all elements of test set T and for each test t in T.

For each  $t$  in T, *compute* the number of distinct testable entities covered.

Step 3: *Arrange the tests* in T in order of their respective coverage.

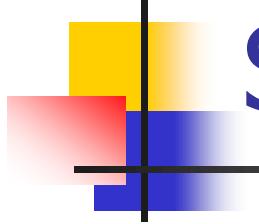
Test with the maximum coverage gets the highest priority and so on.



## Using test prioritization

Once the tests are prioritized, one has the option of using all tests for regression testing or a subset of them.

The choice is guided by several factors, such as the **resources available** for regression testing and the desired product quality.



# Summary

Regression testing is an essential phase of software product development.

In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.

In such situations, one can use sophisticated technique for selecting a subset of all tests and reduce the time for regression testing.