Software Quality Assurance and Testing

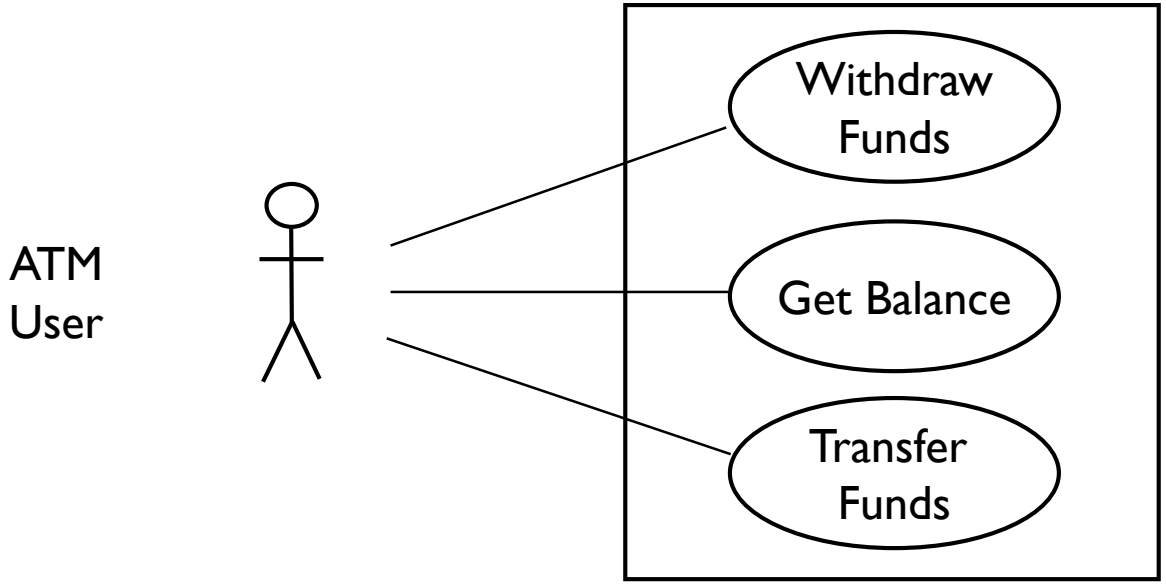
Amr Kamel

Associate Professor,
Department of Computer Science,
Faculty of Computers & Information,
Cairo University.

Graph Coverage for Use Cases

CS496 – Software Testing

Simple Use Case Example



- UML use cases are often used to express software requirements
- Node Coverage!!: Try each use case once …

Use case graphs, by themselves, are not useful for testing

Use Case Documentation

- Use cases are commonly documented further.
- Documentation is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

ATM "Withdraw Fund" Use Case (1/4)

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.

ATM "Withdraw Fund" Use Case (2/4)

- Description (continued) :
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

ATM “Withdraw Fund” Use Case (3/4)

- Alternatives :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.

ATM "Withdraw Fund" Use Case (4/4)

- Alternatives (continued):
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- Post-condition :
  - Funds have been withdrawn from the customer's account.

How to test this?

Our approach in Graph Modeling

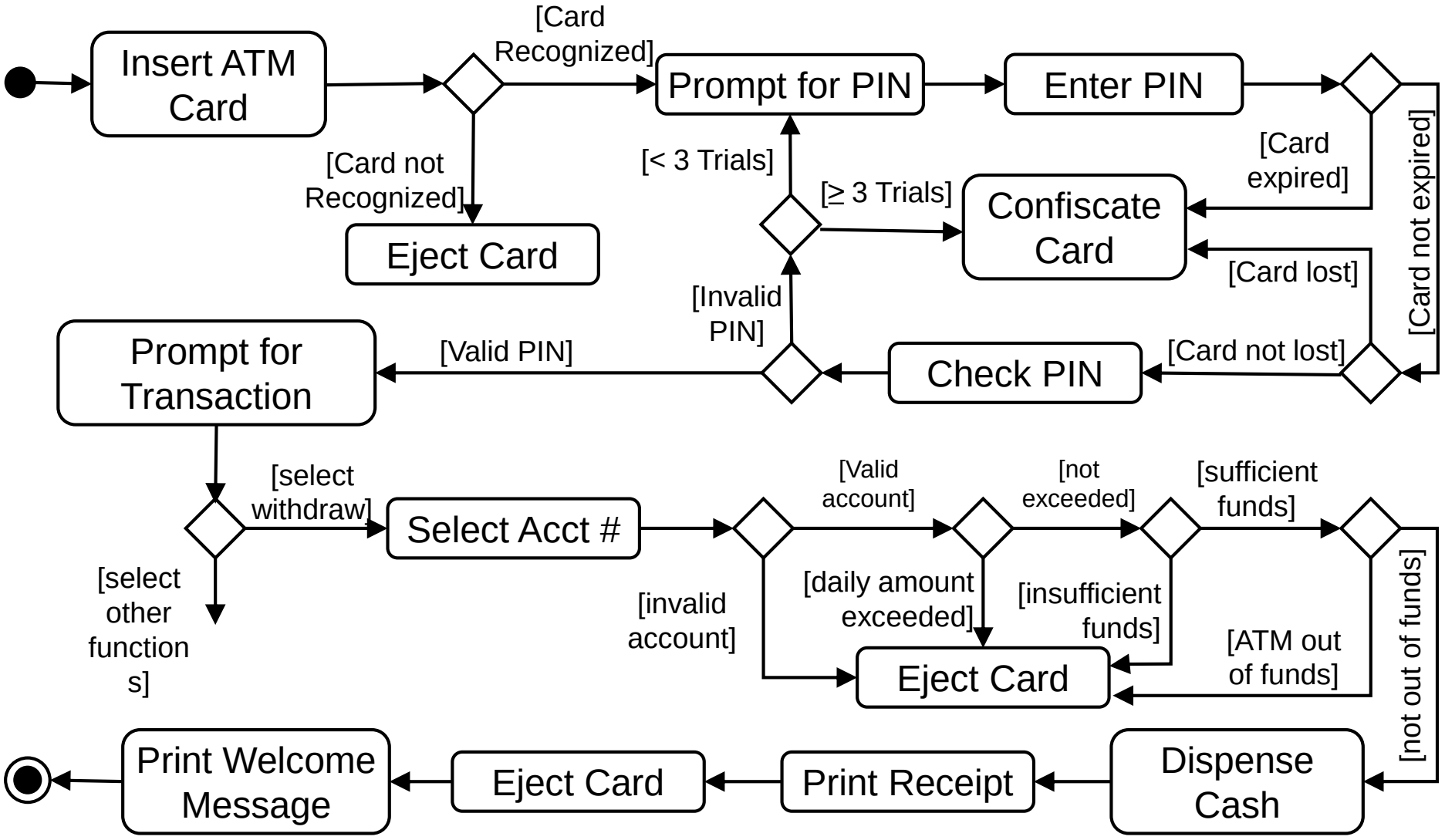
Find a graph then cover it

- Graph to use

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates
  - No obvious DU pairs

"Withdraw Fund" Activity Diagram



Covering Activity Graphs

- Node Coverage & Edge Coverage
  - Inputs to the software are derived from labels on nodes and predicates
  - Used to form test case values
- Data flow techniques do not apply

Covering Activity Graphs

- Scenario Testing
  - Scenario : A complete path through a use case activity graph
  - Should make semantic sense to the users
  - Number of paths often finite
  - If not, scenarios defined based on domain knowledge
  - Use “specified path coverage,” where the set S of paths is the set of scenarios
  - Note that specified path coverage does not necessarily subsume edge coverage, but scenarios should be defined so that it does

Summary of Use Case Testing

- Use cases are defined at the requirements level
- Can be very high level
- UML Activity Diagrams encode use cases in graphs
  - Graphs usually have a fairly simple structure
- Requirements-based testing can use graph coverage
  - Straightforward to do by hand
  - Specified path coverage makes sense for these graphs

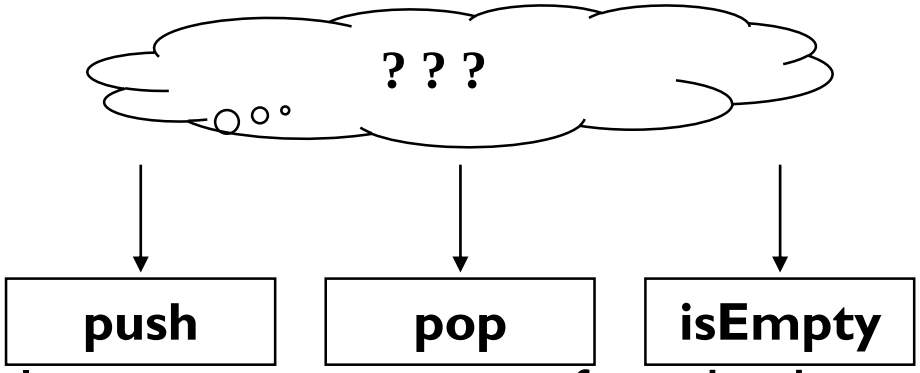
Design Specifications

- A design specification describes aspects of what behavior software should exhibit
- Implementation may not exactly reflect the specifications
  - Design specifications are often called models of the software
- Two types of Specification descriptions
  1. Sequencing constraints on class methods
  2. State behavior descriptions of software

Sequencing Constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called
- They can be encoded as preconditions or other specifications

Class stack
public void push (Object o)
public Object pop ( )
public boolean isEmpty ( )



- Tests can be created for these classes as sequences of method calls
- Sequencing constraints give an easy and effective way to choose which sequences to use

Sequencing Constraints Overview

- Sequencing constraints might be
  - Expressed explicitly
  - Expressed implicitly
  - Not expressed at all
- Testers should derive them if they do not exist
  - Look at existing design documents
  - Look at requirements documents
  - Ask the developers
  - Last choice : Look at the implementation
- If they don’t exist, expect to find more faults !
- Share with designers before designing tests
- Sequencing constraints do not capture all behavior

Queue Example

```
public int deQueue()
{
  // Pre: At least one element must be on the queue.
  … …
public enQueue (int e)
{
  // Post: e is on the end of the queue.
```

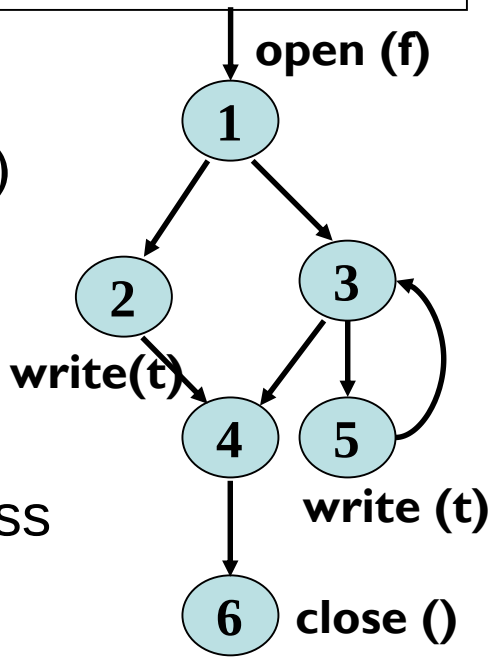
- Sequencing constraints are implicitly embedded in the pre and post-conditions
  - enQueue () must be called before deQueue ()
- Does not include the requirement that we must have at least as many enQueue () calls as deQueue () calls
  - Can be handled by state behavior techniques

File ADT Example

- **open (String fName)** // Opens the file with the name fName
- **write (String textLine)** // Writes a line of text to the file
- **close (String fName)** // Closes the file and makes it
  // unavailable for use

Valid sequencing constraints on FileADT:

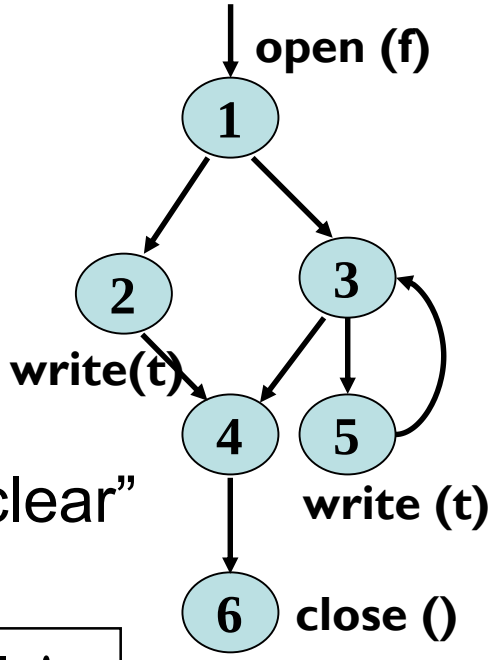
1. An open(F) must be executed before every write(t)
2. An open(F) must be executed before every close()
3. A write(t) must not be executed after a close() unless an open(F) appears in between
4. A write(t) should be executed before every close()
5. A close() must not be executed after a close() unless an open(F) appears in between
6. An open(F) must not be executed after an open(F) unless a close() appears in between



Static Checking

Is there a path that violates any of the sequencing constraints ?

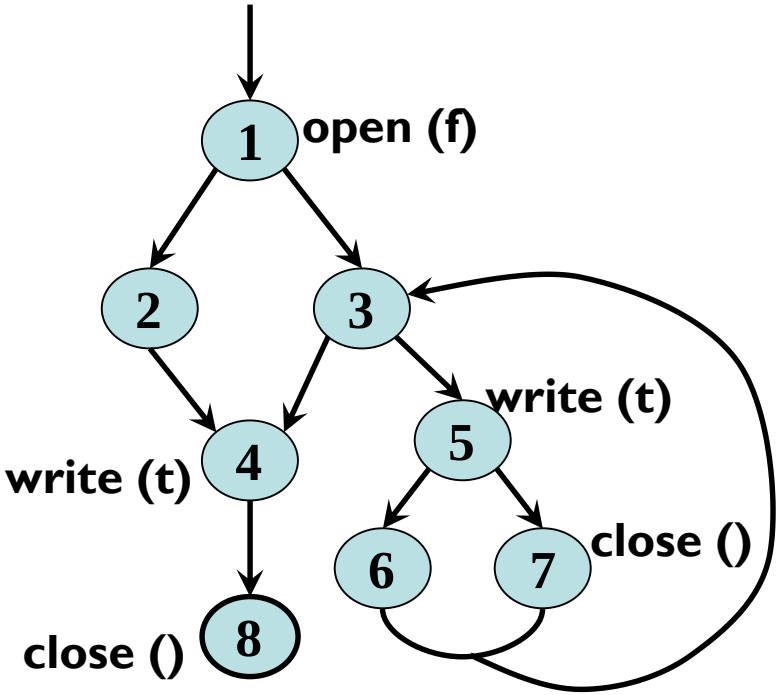
- Is there a path to a write() that does not go through an open()?
- Is there a path to a close() that does not go through an open()?
- Is there a path from a close() to a write()?
- Is there a path from an open() to a close() that does not go through a write()? ("write-clear" path)



[1, 3, 4, 6] – ADT use anomaly!

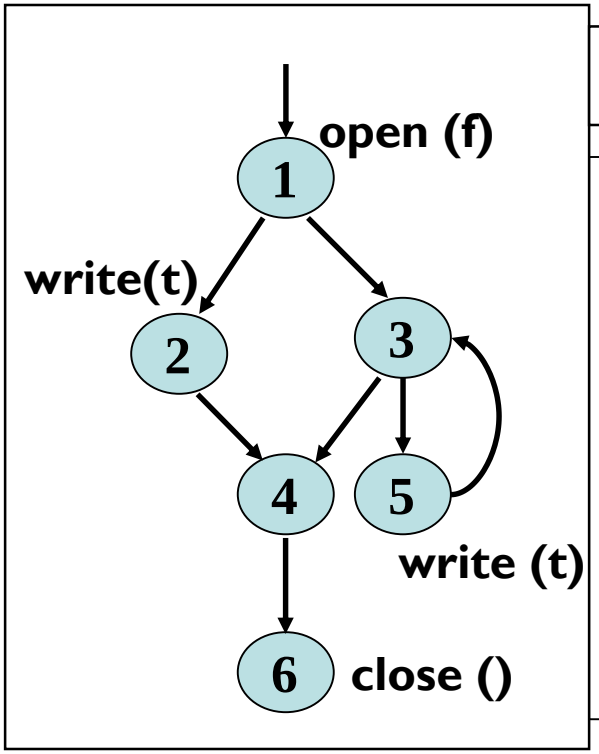
Static Checking

- Consider the following graph :



[7, 3, 4] – close () before write () !

Generating Test Requirements



[1, 3, 4, 6] - ADT use anomaly!

- But it is possible that the logic of the program does not allow the pair of edges [1, 3, 4]
- That is – the loop body must be taken at least once
- Determining this is undecidable – so static methods are not enough

- Use the sequencing constraints to generate test requirements
- The goal is to violate every sequencing constraint

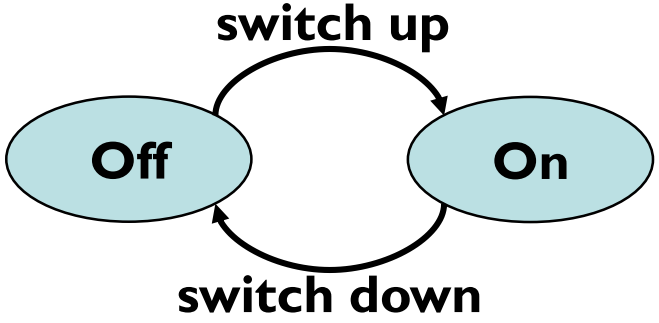
Test Requirements for FileADT

Apply to all programs that use FileADT

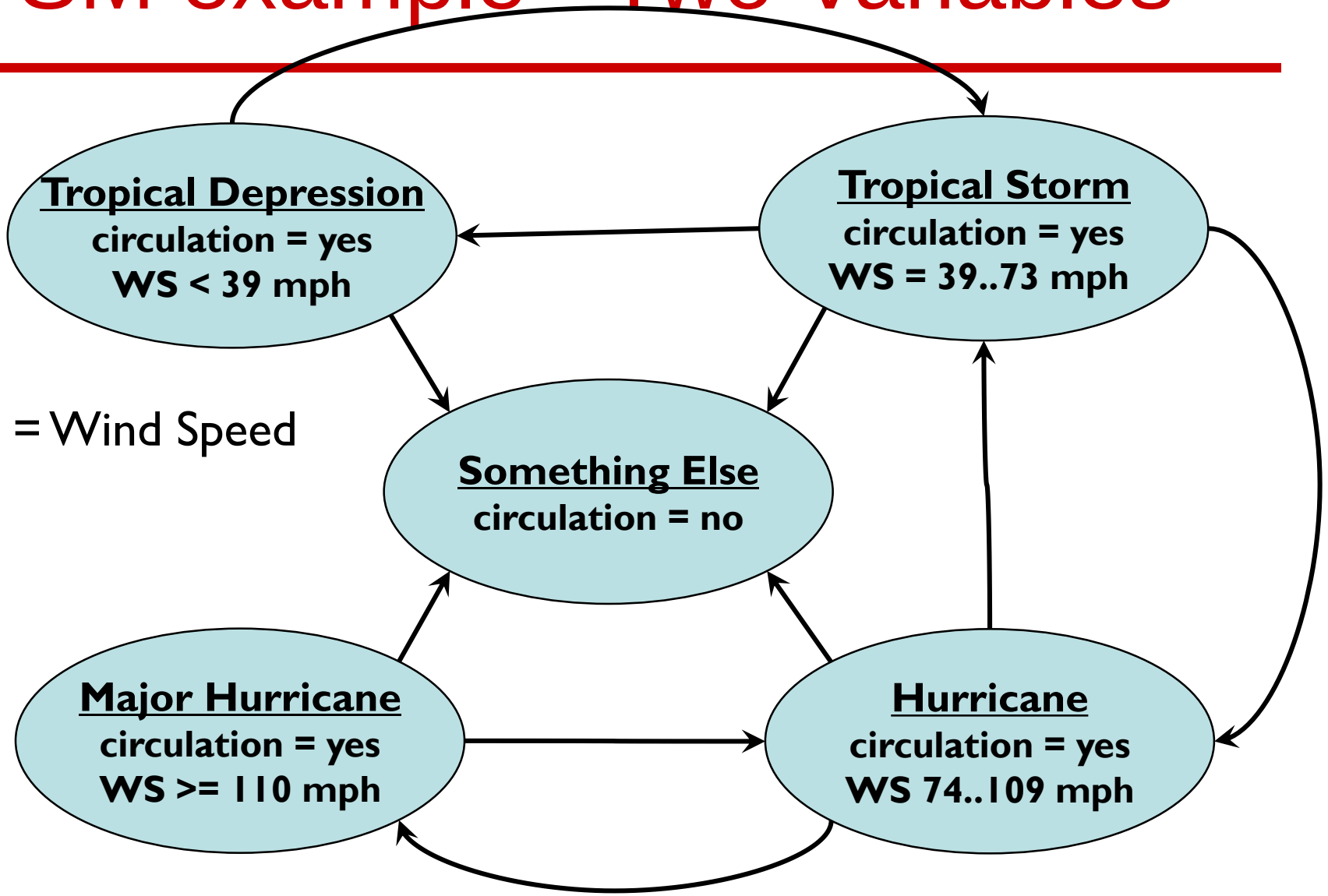
1. Cover every path from the start node to every node that contains a write(t) such that the path does not go through a node containing an open(F).
2. Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open(F).
3. Cover every path from every node that contains a close() to every node that contains a write(t) such that the path does not contain an open(F).
4. Cover every path from every node that contains an open(F) to every node that contains a close() such that the path does not go through a node containing a write(t).
5. Cover every path from every node that contains an open(F) to every node that contains an open(F).

Testing State Behavior

- A Finite State Machine (FSM) is a graph that describes how software variables are modified during execution
- Nodes: States, representing sets of values for key variables
- Edges: Transitions, possible changes in the state



FSM example - Two Variables



Other variables may exist but not be part of state

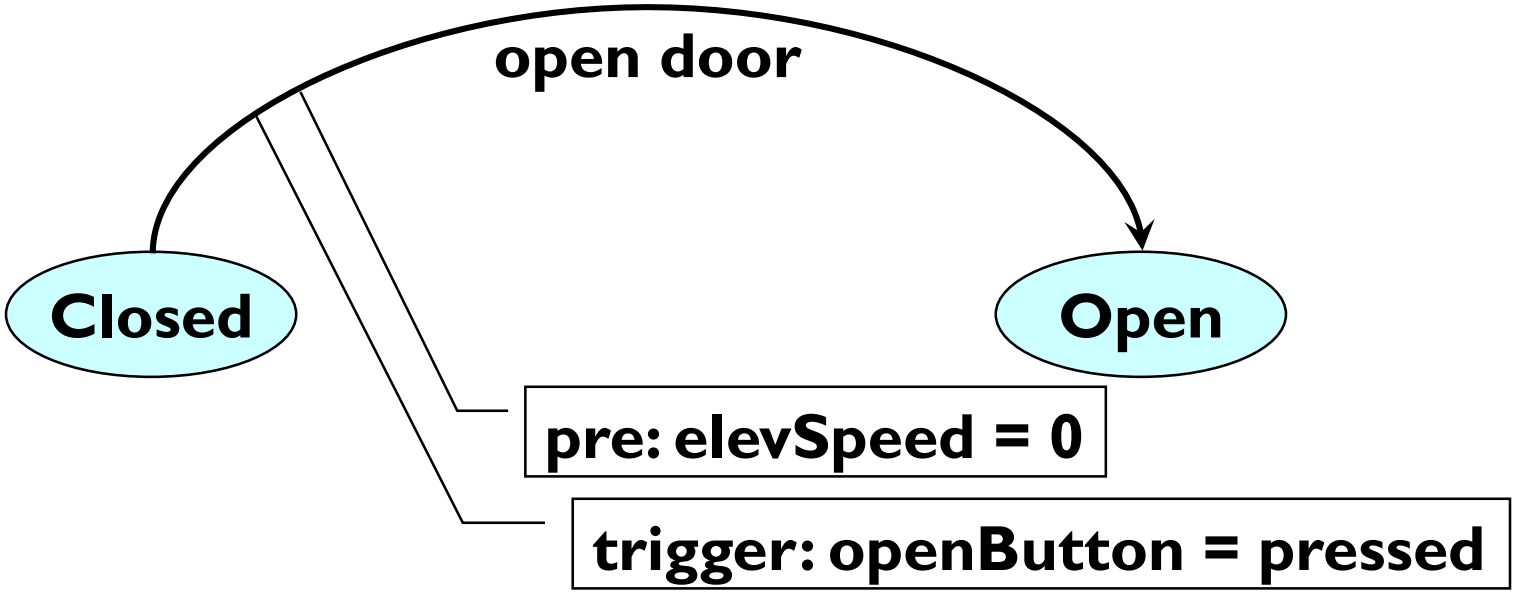
FSM are Common

- FSMs can accurately model many kinds of software
  - Embedded and control software (think electronic gadgets)
  - Abstract data types
  - Compilers and operating systems
  - Web applications
- Creating FSMs can help find software problems
- Numerous languages for expressing FSMs
  - UML state-charts
  - Automata
  - State tables (SCR)
  - Petri nets
- Limitation: FSMs are not always practical for programs that have lots of states (for example, GUIs)

Annotations on FSMs

- FSMs can be annotated with different types of actions
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables

Annotations Basics



- Preconditions (guards): conditions that must be true for transitions to be taken
- Triggering events: changes to variables that cause transitions to be taken

Covering FSMs

- Node coverage : execute every state (*state coverage*)
- Edge coverage : execute every transition (*transition coverage*)
- Edge-pair coverage : execute every pair of transitions (*transition-pair*)
- Data flow:
  - Nodes often do not include defs or uses of variables
  - defs of variables in triggers are used immediately (the next state)
  - defs and uses are usually computed for guards, or states are extended
  - FSMs typically only model a subset of the variables
- Generating FSMs is often harder than covering them …

Deriving FSMs

- With some projects, an FSM (such as a state-chart) is created during design
  - Tester should check to see if the FSM is still current with respect to the implementation
- If not, it is very helpful for the tester to derive the FSM
- Strategies for deriving FSMs from a program:
  1. Combining control flow graphs (*wrong*)
  2. Using the software structure (*wrong*)
  3. Modeling state variables
- Example based on a digital watch …
  - Class Watch uses class Time

Class Watch

```
public class Watch {
// Constant values for the button (inputs)
private static final int NEXT    = 0;
private static final int UP      = 1;
private static final int DOWN = 2;
// Constant values for the state
private static final int TIME          = 5;
private static final int STOPWATCH = 6;
private static final int ALARM         = 7;
// Primary state variable
private int mode = TIME;
// Three separate times, one for each state
private Time watch, stopwatch, alarm;
public Watch () // Constructor
public void doTransition (int button) // Handles inputs
public String toString()  // Converts values
```

```
// Inner class keeps track of
// hours and minutes
public class Time
private int hour   = 0;
private int minute = 0;

public void changeTime(int button)
public String toString()
```

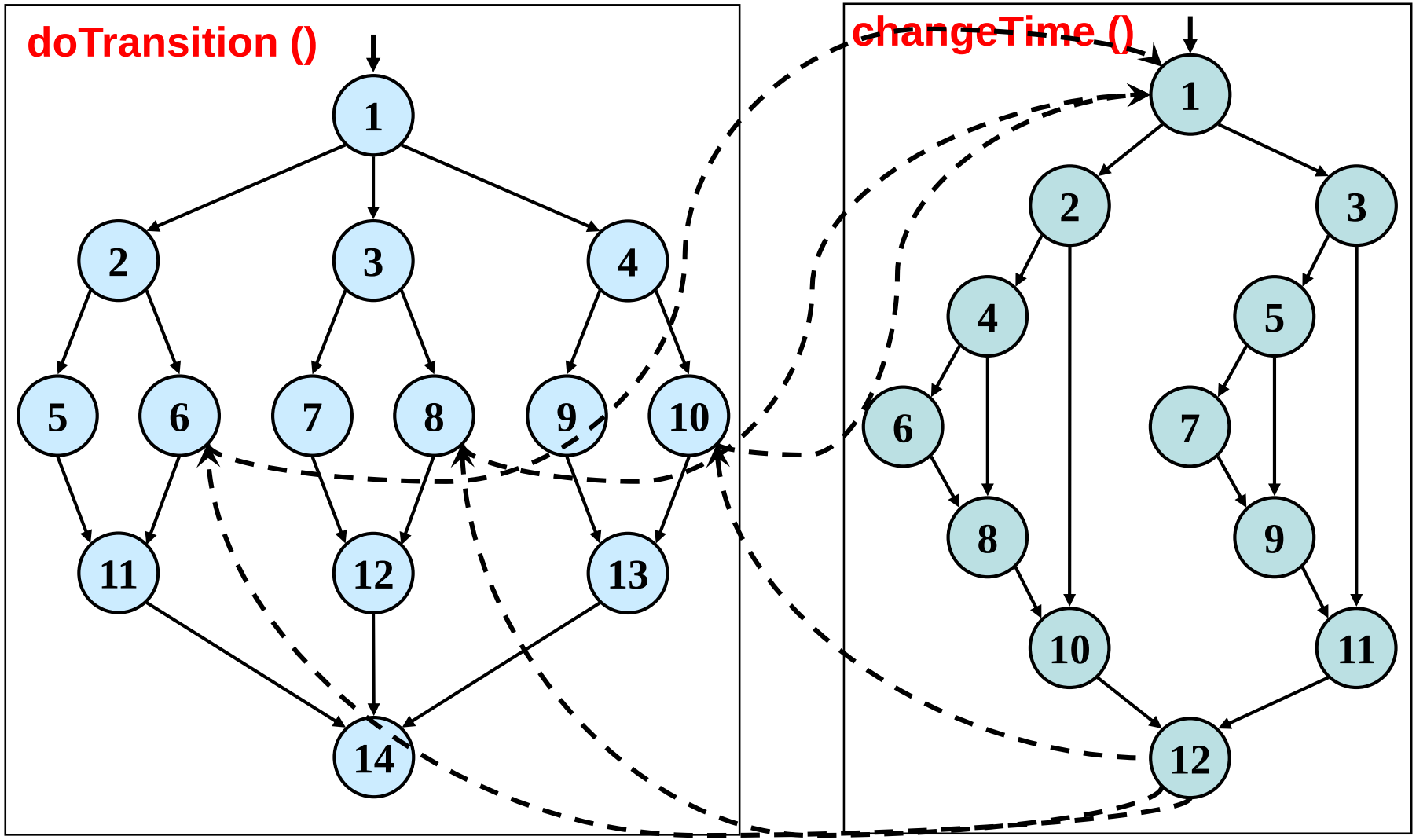
```
// Takes the appropriate transition when a button is pushed.
public void doTransition (int button)
{
   switch ( mode )
   {
      case TIME:
         if (button == NEXT)
            mode = STOPWATCH;
         else
            watch.changeTime (button);
         break;
      case STOPWATCH:
         if (button == NEXT)
            mode = ALARM;
         else
            stopwatch.changeTime (button);
         break;
      case ALARM:
         if (button == NEXT)
            mode = TIME;
         else
            alarm.changeTime (button);
         break;
      default:
         break;
   }
}  // end doTransition()
```

```
// Increases or decreases the time.
// Rolls around when necessary.
public void changeTime (int button)
{
   if (button == UP)
   {
      minute += 1;
      if (minute >= 60)
      {
         minute = 0;
         hour += 1;
         if (hour > 12)
            hour = 1;
      }
   }
   else if (button == DOWN)
   {
      minute -= 1;
      if (minute < 0)
      {
         minute = 59;
         hour -= 1;
         if (hour <= 0)
            hour = 12;
      }
   }
}  // end changeTime()
```

1. Combining Control Flow Graphs

- The first instinct for inexperienced developers is to draw CFGs and link them together
- This is really not an FSM
- Several problems
  - Methods must return to correct call-sites—implicit nondeterminism
  - Implementation must be available before graph can be built
  - This graph does not scale up

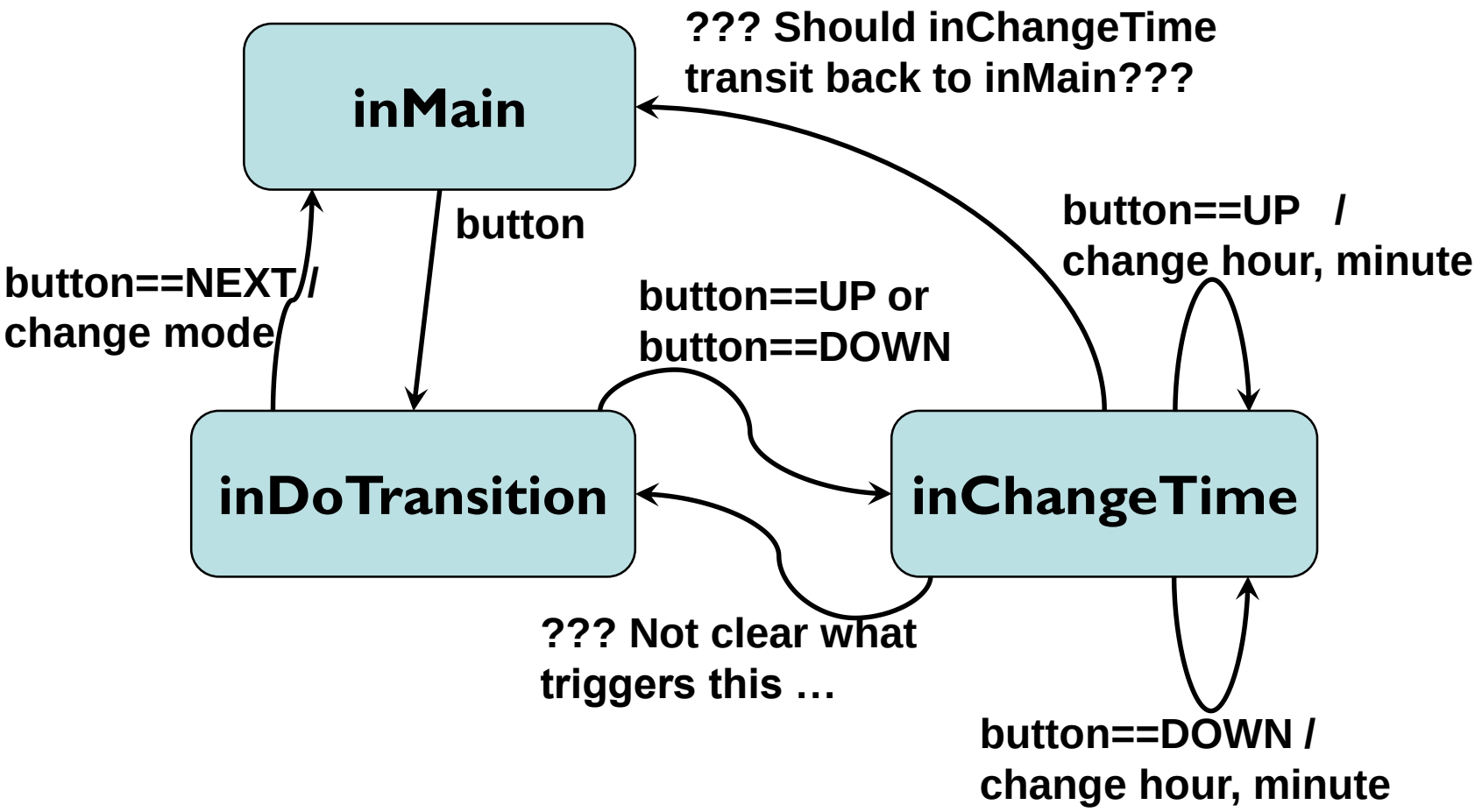
CFGs for Watch Example



2. Using the Software Structure

- A more experienced programmer may map methods to states
- These are really not states
- Problems
  - Subjective—different testers get different graphs
  - Requires in-depth knowledge of implementation
  - Detailed design must be present

SW Structure for Watch Example



3. Modeling State Variables

- More mechanical
- State variables are usually defined early
- First identify all state variables, then choose which are relevant
- In theory, every combination of values for the state variables defines a different state
- In practice, we must identify ranges, or sets of values, that are all in one state
- Some states may not be feasible

State Variables in Watch

Constants

- ~~NEXT, UP, DOWN~~
- ~~TIME, STOPWATCH, ALARM~~

Not relevant, really just values

Non-constant variables in class Watch

- int mode (values: TIME, STOPWATCH, ALARM)
- Time watch, stopwatch, alarm

State Variables in Time

Non-constant variables in class Time

- int hour (values: 1..12)
- int minute (values: 0 .. 59)

12 X 60 values is 720 states (too many)

Combine values into ranges of similar values :

- hour : 1..11, 12
- minute : 0, 1..59

Clumsy ...
Not sequential …

Four states: (1..11, 0); (12, 0); (1..11, 1.. 59); (12, 1 .. 59)

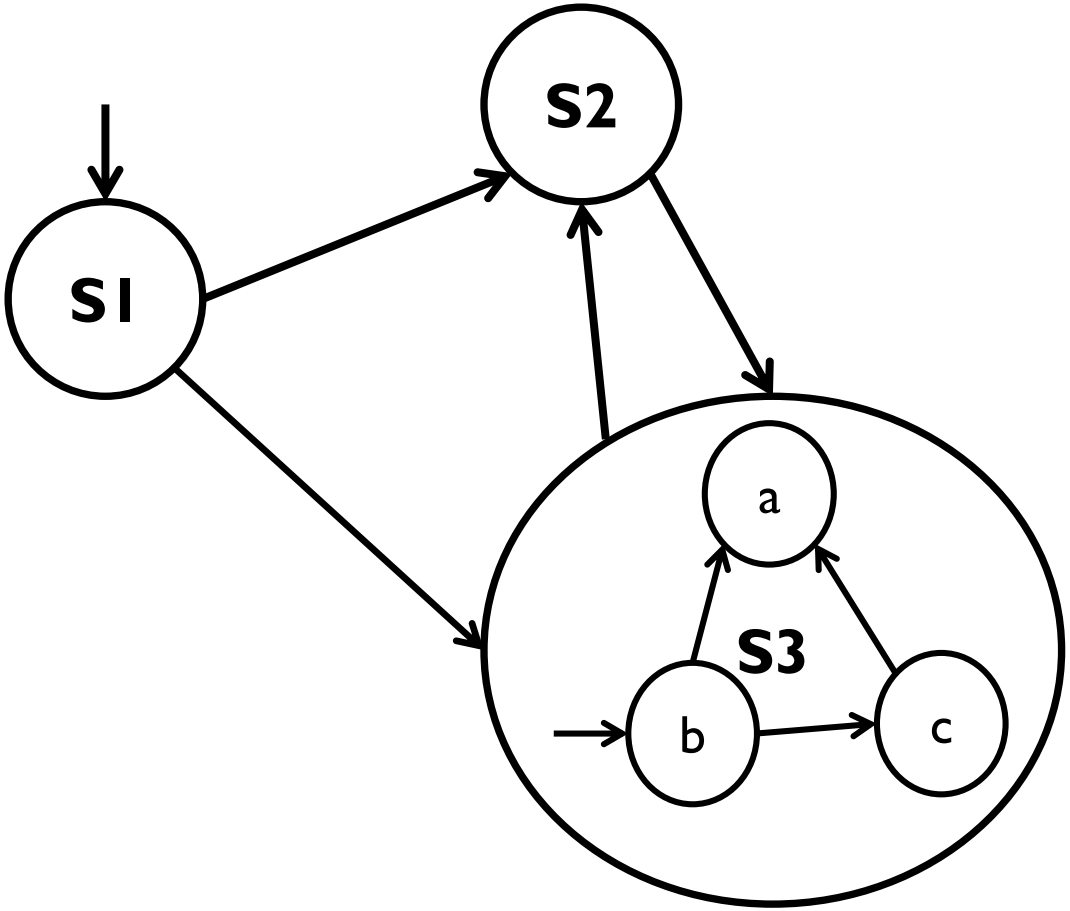
Time : 12:00, 12:01..12:59, 01:00 .. 11:59

These require lots of thought and semantic domain knowledge of the program

Hierarchical FSMs

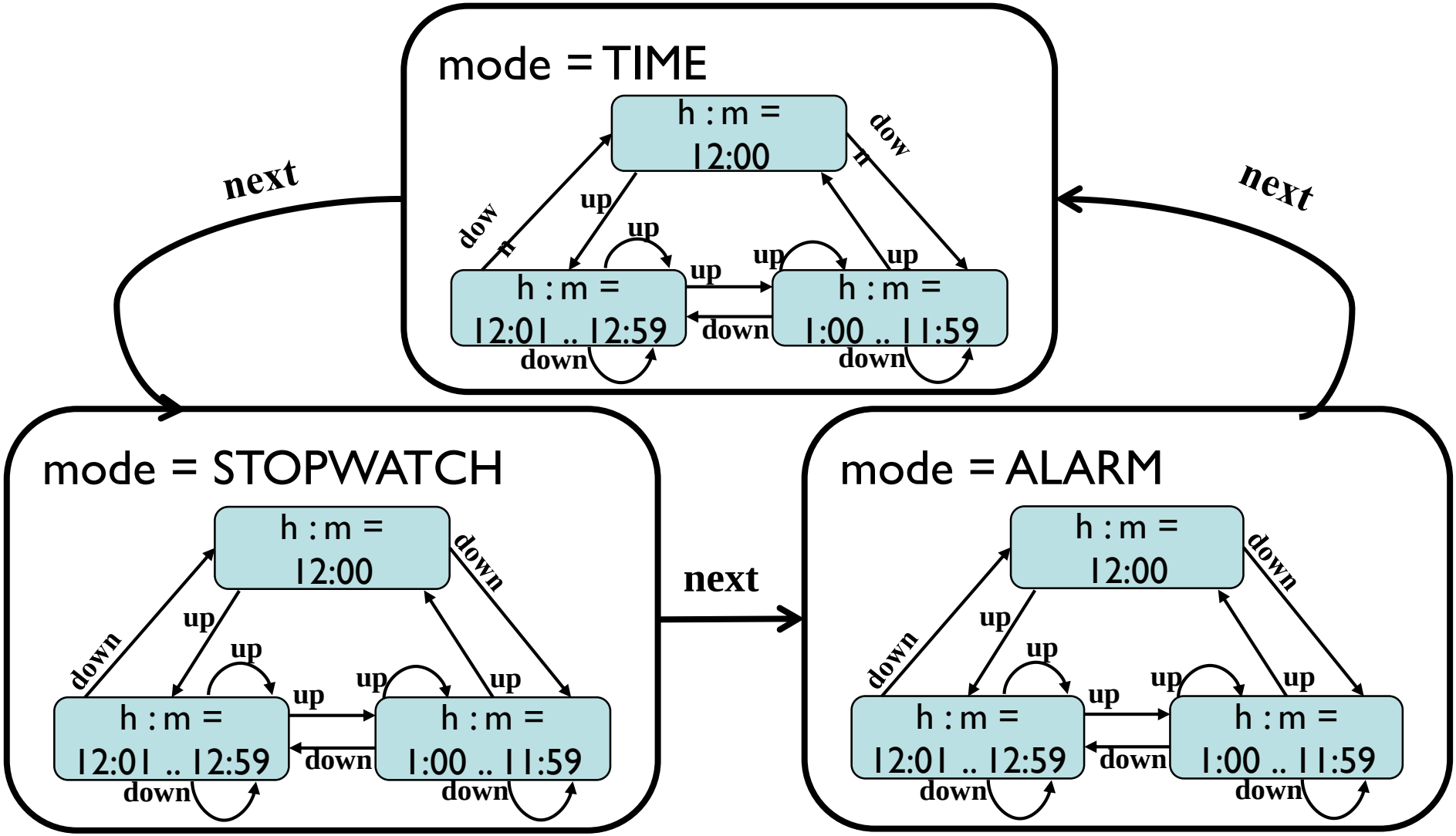
Class Watch uses class Time

How can we model two classes—one that uses another?



One FSM is contained within the other

Watch / Time Hierarchical FSM



Applying GC Criteria to FSMs

- Advantages
  1. Tests can be designed before implementation
  2. Analyzing FSMs is much easier than analyzing source
- Disadvantages
  1. Some implementation decisions are not modeled in the FSM
  2. There is some variation in the results because of the subjective nature of deriving FSMs
  3. Tests have to be “mapped” to actual inputs to the program – the names that appear in the FSM may not be the same as the names in the program