UNIT-V

Object-Oriented Unit testing
(or)
Class Testing

MADHESWARI.K
AP/CSE
SSNCE

Topics to be covered?

- Unit in traditional software system
- Methods as units
- Class as units

class testing

- The main question for class testing is whether a class or a method is a unit.
- In traditional s/w system the common guidelines are that a unit is

  1. the smallest chunk that can be compiled by itself
  2. a single procedure/function (standalone)
  3. Something so small it would be developed by one person

Result

In o-o testing methods and classes are considered as units.

what is class testing?

- Making class as a unit in unit testing solves the intra-class integration problem
- Conventional testing focuses on inter-process-output, whereas class testing focuses on each method , then designing sequences of methods to exercise states of a class

1. Test all features of a class object
2. Units should be tested in isolation
3. Test sequences of methods

Class vs. Procedure Testing

Procedure Testing	Class Testing
basic component: function (procedure)	basic component: class = data members (state) + set of operations
testing method: based on input/output relation	objects (instances of classes) are tested
	correctness cannot simply be defined as an input/output relation, but must also include the object state.
	The state may not be directly accessible, but can normally be accessed using public class operations

Example

```
class Watcher {
 private:
   …
   int status;
   …
 public:
   void checkPressure() {
     …
     if (status == 1)
       …
     else if (status …)
       …
   }
   …
}
```

- Testing method checkPressure() in isolation is Meaningless.
  - Generating test data
  - Measuring coverage
- Creating oracles is more difficult
  - the value produced by method *check_pressure* depends on the state of class *Watcher's* instances (variable *Status*)
  - failures due to incorrect values of variable *Status* can be revealed only with tests that have control and visibility on that variable

Methods as Units

- Pseudo code for O-O Calendar

  Class:CalendarUnit

  Class:testit

  class:Date

  class:Day

  Class:month

  Class:year

- Unit testing for Date. Increment

Methods as Units

- Superficially, this choice reduces object-oriented unit testing to traditional (procedural) unit testing.
- A method is nearly equivalent to a procedure, so all the traditional specification-based(functional)(black box) and code-based testing(Structural) (whitebox) techniques apply.
- Unit testing of procedural code requires stubs and a driver test program to supply test cases and record results.
- If we consider method as o-o units, we must provide stub classes that can be instantiated and a main program class that acts as a driver to provide and analyze test cases.
- Since instances of the nUnit framework are available for most object-oriented languages, the assert mechanism in those frameworks is the most convenient choice.

Methods as Units

- Example

```
//Check that two objects are equal
assertEquals(str1, str2);
//Check that a condition is true
assertTrue (val1 < val2);
//Check that a condition is false
assertFalse(val1 > val2);
```

Note: Equivalence testing can be used for testing methods as units

Class as Units

- Treating a class as a unit solves the intraclass integration problem, but it creates other problems.
- One has to do with various views of a class.
    1. Static view
    2. Compile Time View

Static View

- In the static view, a class exists as source code.
- This is fine if all we do is code reading.
- The problem with the static view is that inheritance is ignored, but we can fix this by using fully flattened classes.

Class as Units

Compile Time View

- We might call the second view the compile-time view because this is when the inheritance actually "occurs."

Execution-time view

- The third view is the execution-time view, when objects of classes are instantiated.
- Testing really occurs with the third view, but we still have some problems.
- For example, we cannot test abstract classes because they cannot be instantiated.
- Also, if we are using fully flattened classes, we will need to "unflatten" them to their original form when our unit testing is complete.
- If we do not use fully flattened classes, in order to compile a class, we will need all the other classes above it in the inheritance tree.

Class as Units

- The class-as-unit choice makes the most sense when little inheritance occurs, and classes have what we might call internal control complexity

Example For Class as unit testing

Pseudocode for Windshield Wiper Class

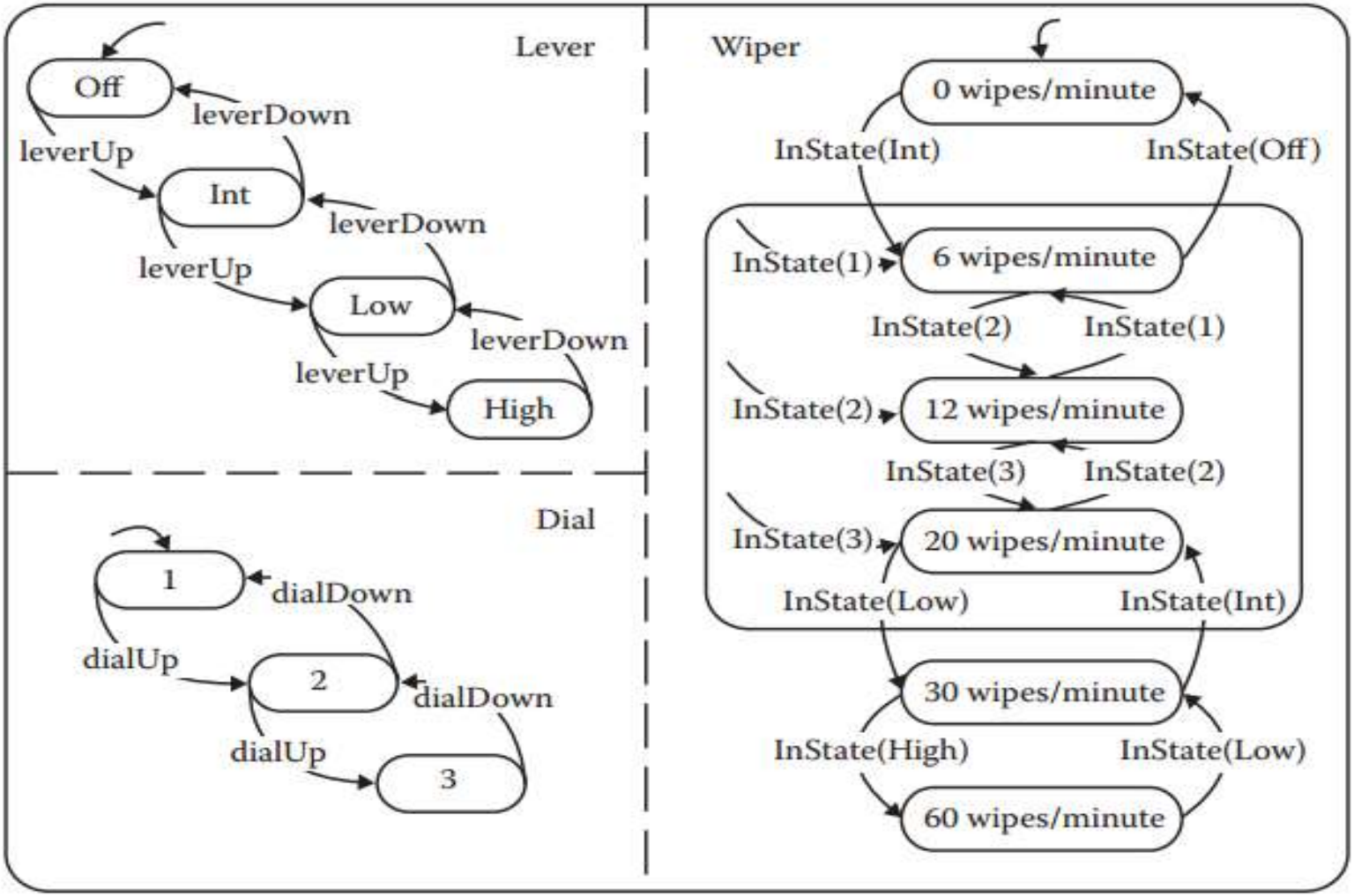
With this formulation, operations sense lever and dial events and maintain the state of the lever and dial in the leverPosition and dialPosition state variables. When a dial or lever event occurs, the corresponding sense method sends an (internal) message to the setWiperSpeed method, which, in turn, sets its corresponding state variable wiperSpeed. Our revised windshieldWiper class has three attributes, get and set operations for each variable, and methods that sense the four physical events on the lever and dial devices.

Class as Units

```
class windshieldWiper
        private wiperSpeed
        private leverPosition
        private dialPosition
        windshieldWiper(wiperSpeed, leverPosition, dialPosition)
        getWiperSpeed()
        setWiperSpeed()
        getLeverPosition()
        setLeverPosition()
        getDialPosition()
        setDialPosition()
        senseLeverUp()
        senseLeverDown()
        senseDialUp(),
        senseDialDown()
End class windshieldWiper
```

Class as Units

Windshield wiper StateChart.



Class as Units

Unit Testing for Windshield Wiper Class

In our example, it makes sense to proceed in a bottom–up order beginning with the get/set methods for the state variables (these are only present in case another class needs them). The dial and lever sense methods are all quite similar;

Class as Units

pseudocode for the senseLeverUp

```
senseLeverUp()
      Case leverPosition Of
            Case 1: Off
                  leverPosition = Int
                  Case dialPosition Of
                        Case 1:1
                              wiperSpeed = 6
                        Case 2:2
                              wiperSpeed = 12
                        Case 3:3
                              wiperSpeed = 20
                  EndCase 'dialPosition
            Case 2:Int
                  leverPosition = Low
                  wiperSpeed = 30
            Case 3: Low
                  leverPosition = High
                  wiperSpeed = 60
            Case 4: High
                  (impossible; error condition)
      EndCase 'leverPosition
End enseLeverUp
```

Class as Units

- Testing the senseLeverUp method will require checking each of the alternatives in the Case and nested Case statements.
- The tests for the "outer" Case statement cover the corresponding leverUp transitions in the StateChart.
- In a similar way, we must test the leverDown, dialUp, and dialDown methods. Once we know that the Dial and Lever components are correct, we can test the wiper component
  Pseudocode for the test driver class will look something like this:

```
class testSenseLeverUp
        wiperSpeed
        leverPos
        dialPos
        testResult 'boolean
main()
        testCase = instantiate windshieldWiper(0, Off, 1)
        windshieldWiper.senseLeverUp()
        leverPos = windshieldWiper.getLeverPosition()
        If leverPos = Int
                Then testResult = Pass
                Else testResult = Fail
        EndIf
End 'main
```

Class as Units

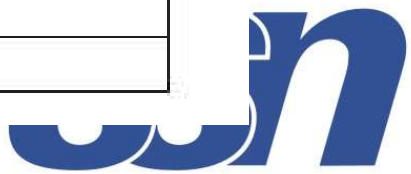
- There would be two other test cases, testing the transitions from INT to LOW, and LOW to HIGH. Next, we test the rest of the windshieldWiper class with the following pseudocode.

```
class test WindshieldWiper
        wiperSpeed
        leverPos
        dialPos
        testResult 'boolean
main()
        testCase = instantiate windshieldWiper(0, Off, 1)
        windshieldWiper.senseLeverUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 6
                Then testResult = Pass
                Else testResult = Fail
        EndIf
End 'main
```

Class as Units

Table 15.1 Lever and Dial Use Case

Use case name	Normal usage
Use case ID	UC-1
Description	The windshield wiper is in the OFF position, and the Dial is at the 1 position; the user moves the lever to INT, and then moves the dial first to 2 and then to 3; the user then moves the lever to LOW; the user moves the lever to INT, and then to OFF.
Preconditions	1. The Lever is in the OFF position.
	2. The Dial is at the 1 position.
	3. The wiper speed is 0.
Event Sequence	
Input Events	Output Events
1. Move lever to INT	2. Wiper speed is 6
3. Move dial to 2	4. Wiper speed is 12
5. Move dial to 3	6. Wiper speed is 20
7. Move lever to LOW	8. Wiper speed is 30
9. Move lever to INT	10. Wiper speed is 20
11. Move lever to OFF	12. Wiper speed is 0
Postconditions	1. The Lever is in the OFF position.
	2. The Dial is at the 3 position.
	3. The wiper speed is 0.



Class as Units

Table 15.2 Test Cases for Lever Component

Test Case	*Preconditions (Instantiate Statement)*	*windshieldWiper Event*	*Expected leverPos*
1	windshieldWiper(0,Off,1)	senseLeverUp()	INT
2	windshieldWiper(0,Int,1)	senseLeverUp()	LOW
3	windshieldWiper(0,Low,1)	senseLeverUp()	HIGH
4	windshieldWiper(0,High,1)	senseLeverDown()	LOW
5	windshieldWiper(0,Low,1)	senseLeverDown()	INT
6	windshieldWiper(0,Int,1)	senseLeverDown()	OFF