Lecture 15: Testing (Cont.)

Class-based Testing in Matlab

- testCase.verifyEqual

```
%% Test Class Definition
classdef MyComponentTest < matlab.unittest.TestCase

    %% Test Method Block
    methods (Test)

        %% Test Function
        function testASolution(testCase)
            %% Exercise function under test
            % act = the value from the function under test

            %% Verify using test qualification
            % exp = your expected value
            % testCase.<qualification method>(act,exp);
        end
    end
end
```

Testing APP

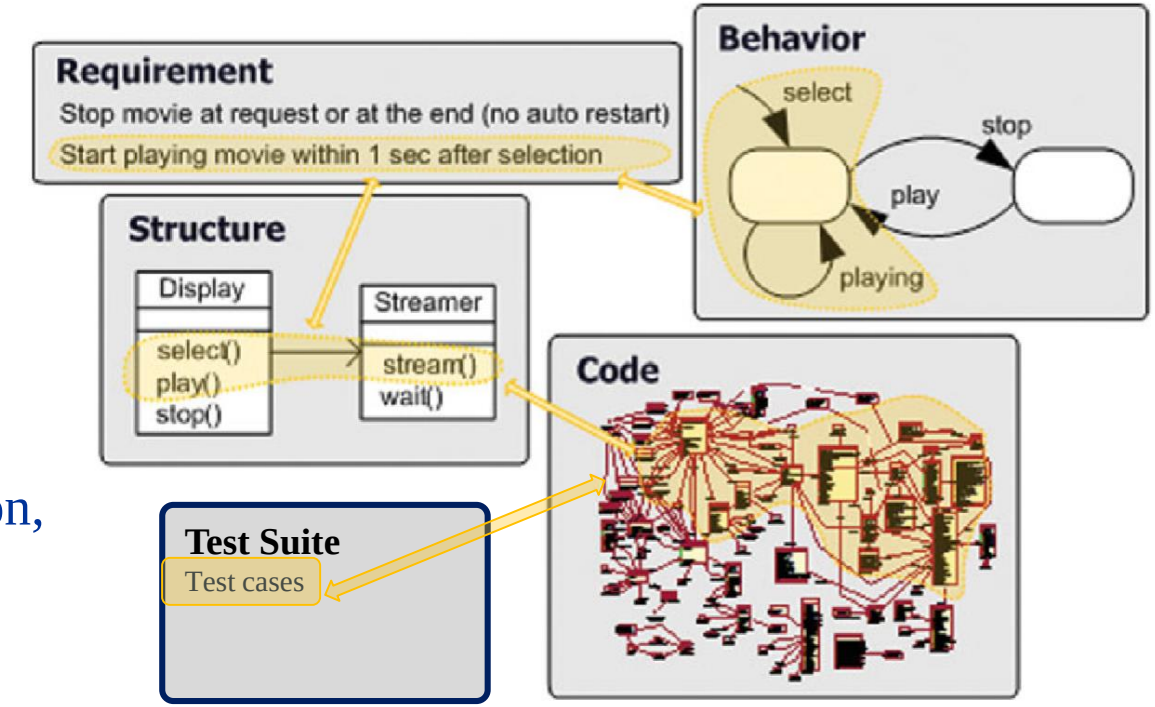
```
classdef TestPatientsDisplay < matlab.uitest.TestCase
    properties
        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete,testCase.App);
        end
    end

    methods (Test)
        function test_plottingOptions(testCase)
            % Press the histogram radio button
            testCase.press(testCase.App.HistogramButton)

            % Verify xlabel updated from 'Weight' to 'Systolic'
            testCase.verifyEqual(testCase.App.UIAxes.XLabel.String,'Systolic')

            % Change the Bin Width to 9
            testCase.choose(testCase.App.BinWidthSlider,9)

            % Verify the number of bins is now 4
            testCase.verifyEqual(testCase.App.UIAxes.Children.NumBins,4)
        end

        function test_tab(testCase) ...

    end
end
```

Component	`matlab.uitest.TestCase` Gesture Method				
	`press`	`choose`	`drag`	`type`	`hover`
Button	✓				
State button	✓	✓			
Check box	✓	✓			
Switch	✓	✓			
Discrete knob		✓			
Knob		✓	✓		
Drop-down		✓		✓	
Edit field				✓	
Text area				✓	
Spinner	✓			✓	
Slider		✓	✓		
List box		✓			
Button group		✓			
Tab group		✓			
Tab		✓			
Tree node		✓			
Menu	✓				
Date Picker				✓	
Axes	✓				✓
UI Axes	✓				✓
UI Figure	✓				✓

Traceability

What is traceability?

- We would like to make sure that
  - All requirements are implemented
  - All implementations are necessary
- Trace artifacts
  - Requirements, models, code, etc.
- Trace link
  - Association between two trace artifacts
  - Type: Refinement, Abstraction, Implementation, etc.
- Trace granularity: component level, statement level, etc.
- Trace quality: completeness, correctness, etc.

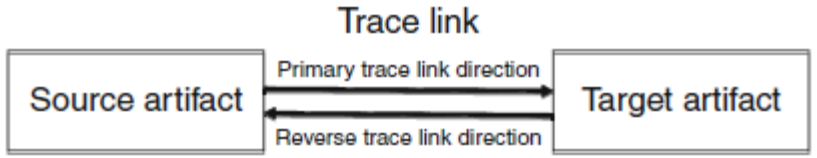


Objectives of Traceability

- Software lifecycle involves more than one person
- Within the team
  - Make sure the requirements are faithfully translated to code
- For the customers and regulation agencies
  - Part of validation evidence

Traceability Activities

- Trace Creation
  - Establish *trace link* between a *source artifact* and a *target artifact*
  - Traceability document



- Trace Validation
  - Between requirements and model: Model checking
  - Between concept model and implementation model: Model translation
  - Between model and code: Conformance testing
- Trace Maintenance
  - Update trace when modification happened

Naming Rules

- Requirements start with R
  - R1:
- Specifications start with S
  - S1
    - S1.1
- Test cases start with T
  - T1
- Model checking properties start with M
  - M1

Traceability Report

Requirement	Implemented by	Validated by
R1	S1	T1.1, T1.2…

Dealing with complexity

- Human can only deal with a limited amount of complexity at a particular time
  - Design with hierarchies
  - Object-Oriented design

Information Hiding

- Hiding complexity
  - You don’t need to deal with them at more abstract levels
- Hiding sources of change
  - Limit the effects of change within a scope
- Example: int id; id++;
  - id=NewID();
    - Hide implementation details
    - Hide potential changes to range and pattern of id
  - TypeID id; id=NewID();
    - Hide potential changes to the type of id

Loose Coupling

- Small, direct, visible and flexible relationships between modules (classes and routines)
- Easily reusable
- Low dependencies between modules

Coupling Criteria

- Size
  - Number of connections between modules
    - Public methods of a class
    - Input variables of a routine
- Visibility
  - Input variables of a routine are obvious, which is good.
  - Components that can edit global variables are not so obvious, which is bad
- Flexibility
  - How easily other component can use the connection?

Types of coupling

- data-parameter coupling
  - a=fun(b,c) and b,c are primitive data types
- object coupling
  - One component instantiate another object
- Object-parameter coupling
  - Object 1 requires Object 2 to pass to Object 3
  - Assumes Object 2 knows about Object 3

Other Considerations

- Strong Cohesion
  - How closely all the routines in a class support a central purpose
  - i.e. UI just for displaying and collect commands
- Contracts between classes
  - Formally specify what you would expect from other components
- Design for test

Defensive Programming

- A good program never puts out garbage, regardless of what it takes in.
  - Garbage in, nothing out
- Protecting your program from invalid inputs
  - Check the values of all data from external sources
  - Check the values of all routine input parameters
  - Decide how to handle bad inputs

Assertions

- Send error message when certain condition is false
- For conditions that should never occur
- Use assertions to document and verify preconditions and postconditions
- For already robust code, assert and handle the code anyway
  - A good way to document your assumptions

Example

- assert(cond,msg)
- tf = isa(A,dataType)

- assert(isa(table,'string'))
- assert(isa(order,'Order'))
- assert(~isempty(order.items))

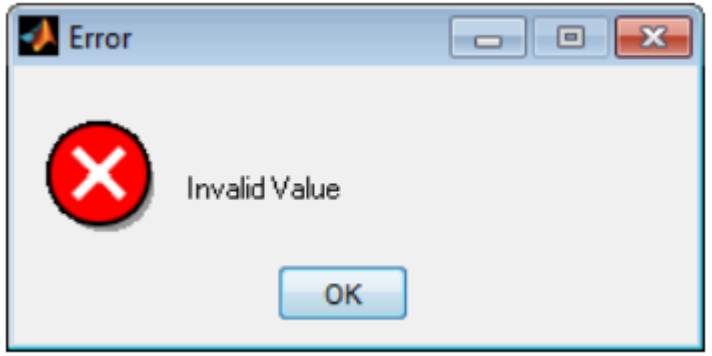
```
function succ=update(DB,table,order)
    succ=0;
    for i=1:length(DB.orderList)
        if strcmp(DB.orderList(i).table,table) && ~strcmp(DB.orderList(i).sta
            DB.orderList(i)=order;
            succ=1;
              DB.processor.displayMessage(sprintf('Order for %s Updated',orde
              DB.processor.displayOrder(order);
            break;
        end
    end
end
```

Error Handling Techniques

- Return a neutral (harmless) value
- Substitute the next piece of valid data (if data corrupted during video streaming)
- Return the same answer as the previous time (acted as a filter)
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error processing routine
- Handle the error in whatever way works best locally
- Shut down/ reboot
- A design decision that should be made early

Message Dialog

- f = msgbox(message,title,icon)
- f = msgbox('Invalid Value', 'Error','error');

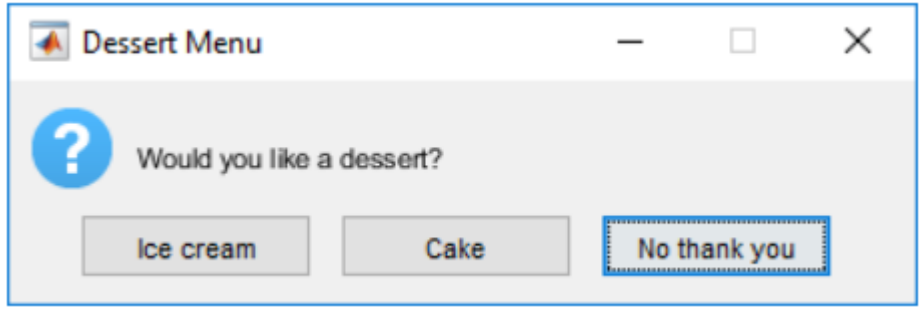


Value	Icon
'help'	
'warn'	
'error'	
'none'	No icon displays.

Quest dialog

- answer = questdlg(quest,dlgtitle,defbtn)

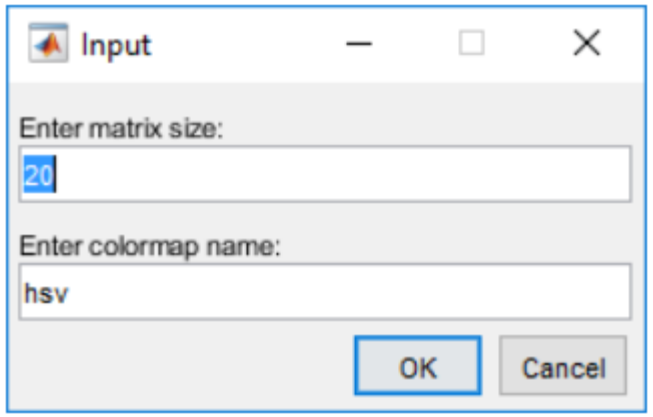
```
answer = questdlg('Would you like a dessert?', ...
        'Dessert Menu', ...
        'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch answer
    case 'Ice cream'
        disp([answer ' coming right up.'])
        dessert = 1;
    case 'Cake'
        disp([answer ' coming right up.'])
        dessert = 2;
    case 'No thank you'
        disp('I''ll bring you your check.')
        dessert = 0;
end
```



Input Dialog

- answer = inputdlg(prompt,dlgtitle,dims,definput)

```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlgtitle = 'Input';
dims = [1 35];
definput = {'20','hsv'};
answer = inputdlg(prompt,dlgtitle,dims,definput)
```



try and catch

```
try
   statements
catch exception
   statements
end
```

```
try
    a = notaFunction(5,6);
catch
    warning('Problem using function.  Assigning a value of 0.');
    a = 0;
end
```

```
try
    a = notaFunction(5,6);
catch ME
    switch ME.identifier
        case 'MATLAB:UndefinedFunction'
            warning('Function is undefined.  Assigning a value of NaN.');
            a = NaN;
        case 'MATLAB:scriptNotAFunction'
            warning(['Attempting to execute script as function. '...
                'Running script and assigning output a value of 0.']);
            notaFunction;
            a = 0;
        otherwise
            rethrow(ME)
    end
end
```

Reasons for Code Refactoring

- A routine is too long
- Inheritance hierarchies have to be modified in parallel
- Related data items used together are not in the same class
- A routine used more features of another class than of its own class
- A class does not do very much
- One class is overly intimate with another
- Data members are public
- A subclass only uses a small percentage of its parents' routines
- Global variables are used

Data-level Refactoring

- Replace a magic number with a named constant
- Replace an expression with a routine
- Introduce an intermediate variable with appropriate name
- Convert multiuse variables to multiple single-use variables (temp)
- Replace traditional records to data classes

Statement-level Refactoring

- Give a variable with useful name
- Use break or return to break a loop
- Return as soon as you know the answer

Routine-level Refactoring

- Turn inline code into routines
- Convert long routine to a class to improve readability
- Separate query operations from modification operations
- Combine similar routines by parameterization
- Pass whole object rather than specific fields

Class Refactoring

- Implementation
  - Change value objects to handle/reference objects, or vice versa
  - Extract specialized code into subclasses
  - Combine similar code into superclasses
- Interface
  - Move a routine to another class
  - Convert one class to two
  - Introducing an extension class
  - Modify class properties via routines instead of making it public
  - Hide routines that are not intended to be used outside the class

Safe Refactoring

- Version control
- Keep each refactoring small
- One at a time
- Plan the list of steps
- Add test cases