Applying Design Patterns to Automated Testing

Brian Le Suer

- Description of a commonly occurring problem
- Template for a reusable solution

What is a Design Pattern ?

- Solution is already tested
- Makes tests more modular
- Reduces the amount of test maintenance

Why Apply Design Patterns ?

- Pattern Name
- Problem Description
- Solution Description
- Consequences of Applying the Pattern

*According to authors of 'Design Patterns: Elements of Reusable Object-Oriented Software'

*4 Basic Elements

- A few words that describe the pattern

Pattern Name

- Describes when to apply the pattern

Problem

- An abstract description of the resolution

Solution

- The costs and benefits of applying the pattern

Consequences

- Problem: Application under test may respond to a stimulus by invoking none, one or more windows

- Example: Adding an item to a shopping cart might results in an up-sell, cross-sell, back-order, out of stock or delivery delay page

Pattern: MultiWaitForWindows

- Solution: Must be able to handle three possible conditions:
  - No window will be displayed
  - One window will be displayed
  - A sequence of windows will be displayed

```
AppObject MultiWaitForWindows (List<AppObject> lwWindow, Integer iTimeout, NotExistAction ReportAction)
  Integer iLoop
  AppObject wActiveWindow
  Boolean bExists

  iLoop = 0
  bExists = false
  while !bExists && iLoop < iTimeout
    for (wActiveWindow in lwWindow)
      if wActiveWindow.WaitUntilExists (0)
        bExists = true
        break
    iLoop++
    Sleep (1)

  if !bExists
    UA.SaveSnapshot (DateTime.Now ().Format ("mmddyyhhnnss") + ".ss")
    switch ReportAction
      case NotExistAction.None
        Print ("None of the  windows in the list {lwWindow} was active")
      case NotExistAction.Warning
        LogWarning("None of the  windows in the list {lwWindow} was active")
      case NotExistAction.Error
        LogError ("None of the  windows in the list {lwWindow} was active")
      case NotExistAction.Exception
        throw TestException ("None of the  windows in the list {lwWindow} was active")
  return (wActiveWindow)
```

- Consequences:
  - Pattern does not anticipate sequence of windows
    - Requires the user to call the solution in a loop for a sequence
  - Requires the user to take action to handle the active window

- Problem: a portion of the text to be verified needs to be masked so that the predictable part of the data can be compared to the expected result

- Example: date time stamps and system generated IDs are frequently included in character sequences that also contain data that needs to be validated

Pattern: FuzzyVerify

- Solution: must be able to mask out unpredictable parts of the target string in any position in which they might occur

Pattern: FuzzyVerify

```
VerifyStringsWithMasking(String ExpectedResult, String ActualResult)
    List<String> lsPartialStrings = ExpectedResult.Split("*")
    Boolean bMatch = true
    for (String sString in lsPartialStrings)
        if !ActualResult.Contains (sString)
            bMatch = false
            LogError ("the expected partial string {sString} did not match an part of the actual string {ActualResult}")
    if bMatch
        Print ("the actual string matches the masked expected string")
```

- Consequences: in order for this pattern to be used successfully, the caller needs to properly delimit the expected result. If the mask is incorrectly applied, then the comparison will fail.

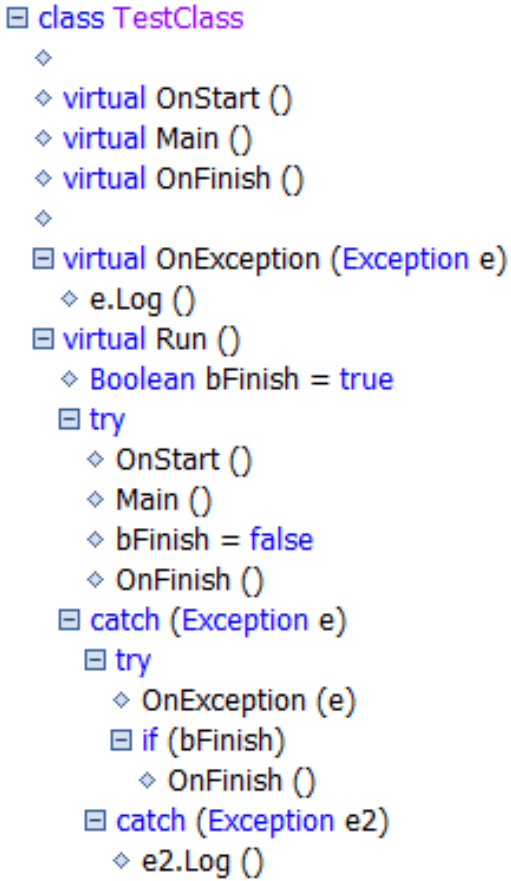
- Problem: An automated test is typically built upon the assumption that it will start from some known state, often referred to as the 'base' state. This must be so, because when the action of a test begins, it will manipulate objects that it expects to exist in an enabled state.

- Example: a test might start by invoking some window from the AUT's main menu. If, because of the current application context, the menu item does not exist or is disabled, the test will fail.

Pattern: Test Class

- Solution: must provide a way to ensure that the AUT is at the ‘base’ state at the start of each test. It must provide a structure where control will be passed from the current test to a set of methods capable of carrying out the necessary actions should a test fail or lose context.

Pattern: Test Class


class TestClass
virtual OnStart ()
virtual Main ()
virtual OnFinish ()
virtual OnException (Exception e)
e.Log ()
virtual Run ()
Boolean bFinish = true
try
OnStart ()
Main ()
bFinish = false
OnFinish ()
catch (Exception e)
try
OnException (e)
if (bFinish)
OnFinish ()
catch (Exception e2)
e2.Log ()

- Consequences:
  - It may be argued that such a pattern is inefficient in that steps are taken between each test to return the AUT to the 'base' state, where fewer actions would be required to start from the state achieved by a previous test.
  - Another argument might pose that a group of tests start and stop from some other application state that requires many steps to achieve.

- Problem: When a link or button is clicked in a web application during an automated test, there is an expectation that a specific page will be displayed. But there are conditions that may exist, most often related to security, that cause some other page or dialog to interrupt the normal processing flow.

- The problem is further complicated when tests are intended to run against different browser types because the triggers and the pages that are invoked vary.

Pattern: NavigateHandler

- Solution: must provide a way to handle any number of pages or dialogs that may or may not appear. It must also provide a mechanism for customizing the responses to for each browser.

Pattern: NavigateHandler

```
class WebNavigationHandler
    WebPage PageFrom
    
    virtual Boolean OnDone ()
        return (true)
    virtual Boolean OnSecurityDialog ()
        return (true)
    virtual Boolean OnLoginDialog ()
        return (true)
    virtual Boolean OnOtherDialog ()
        return (true)
    virtual Boolean OnNavigationBlocked ()
        return (true)
    virtual Boolean OnInfoBarSecurity ()
        return (true)
    virtual Boolean OnInfoBarPopupBlocked ()
        return (true)
    virtual Boolean OnInfoBarOther ()
        return (true)
    
    virtual Boolean Process (WebPage? _PageFrom, WebViewState State)
        PageFrom = _PageFrom
        switch (State)
            case WebViewState.Done
                return (OnDone())
            case WebViewState.SecurityDialog
                return (OnSecurityDialog())
            case WebViewState.LoginDialog
                return (OnLoginDialog())
            case WebViewState.OtherDialog
                return (OnOtherDialog())
            case WebViewState.NavigationBlocked
                return (OnNavigationBlocked())
            case WebViewState.InfoBarSecurity
                return (OnInfoBarSecurity ())
            case WebViewState.InfoBarPopupBlocked
                return (OnInfoBarPopupBlocked ())
            case WebViewState.InfoBarOther
                return (OnInfoBarOther ())
            default
                return (true)
```

```
class IE_DefaultNavigateHandler : WebNavigateHandler
    Boolean OnSecurityDialog ()
        IE_DialogBox_Security.Continue ()
        return (false)
    Boolean OnNavigationBlocked ()
        IE_Page_NavigationBlocked.Continue ()
        return (false)
```

```
void Navigate (Integer x = null, Integer y = null)
    WebPage PageFrom = GetPage()
    AppObject Owner = PageFrom.GetOwner()
    AppObject View = Owner.View
    
    Click (x, y)
    
    Type NavigateHanderType = Web.GetNavigateHandler()
    WebNavigateHandler NavigateHandler = NavigateHanderType.Instantiate()
    
    Timer timer
    timer.Start ()
    
    for (;;)
        timer.Stop ()
        WebViewState State = View.WaitWhileLoading ()
        timer.Start ()
        
        if (NavigateHandler.Process (PageFrom, State))
            break
        
        if (timer.GetElapsed() > Web.GetNavigateTimeout())
            throw Exception ("Navigate operation exceeded timeout of {Web.GetNavigateTimeout()} seconds")
```

- Consequences: While the design of this pattern is not complex, its successful implementation requires a mechanism for getting the state of the browser.
- In this example, a method called *WaitWhileLoading* is assumed to exist. It not only synchronizes with the browser, but also returns the current state, which is passed to the navigation handler for processing.

- Problem: There are many types of tests that are built to verify different aspects of the AUT's GUI, including but not limited to field data types, field length, required/optional fields, tab order and adherence to standards.

- While the expected results for each of these tests will vary by field or by dialog, the requirements are common and reusable solutions can be easily developed

Pattern: Universal Tests

- Solution: A universal test must be reusable for any page, window or dialog. It must not include any application specific references. Instead, it only encodes ‘rules’ that are applied to whatever is passed into the test.

The following is an example of a universal test:

```
class VerifyFieldLength : Step
    parameter AppObject FieldName
    parameter Integer FieldLength
    Main ()
        String TestString = ""

        TestString = String.Replicate("a",FieldLength)

        FieldName.SetValue(TestString + "x")

        FieldName.VerifyValue(TestString)

        return
```

- Consequences: Implementing a universal test requires an understanding of abstraction

- Applying design patterns
  - Saves time
  - Reduces maintenance
  - Provides assurance that the solution has already been tested

Summary of Benefits

Thank you!
Questions?