



# Capture the Essence of Your Test Cases

Designer Tools' Testing Strategy  
by Michael J. Hunter




The typical scripted test case is difficult to understand and even more difficult to maintain. Take this one, for example. Even if you could read it, you would be hard pressed to understand what it was doing without time-consuming scrutiny. I have written many test cases just like this. You probably have as well – and if not, I bet you have had to maintain some!

```
Logical.Projects.CreateNewProject();  
  
Logical.SceneElements.CreateRectangle(  
    new Point(100, 100), new Point(300, 300));  
  
Logical.SceneElements.SelectAll();  
  
Logical.Appearance.SetAppearanceProperty(  
    Logical.Fill);  
Logical.Appearance.SetColor(DataManager  
    .FullSpectrumProvider.GetNextValue());  
  
Logical.SceneElements.Cut();  
  
Logical.SceneElements.Paste();
```

This is exactly the same test case, but with a series of techniques applied that together let the essence of the test case come through. Not only does this version do everything the much longer version did, but this test case can take the place of many other test cases as well.



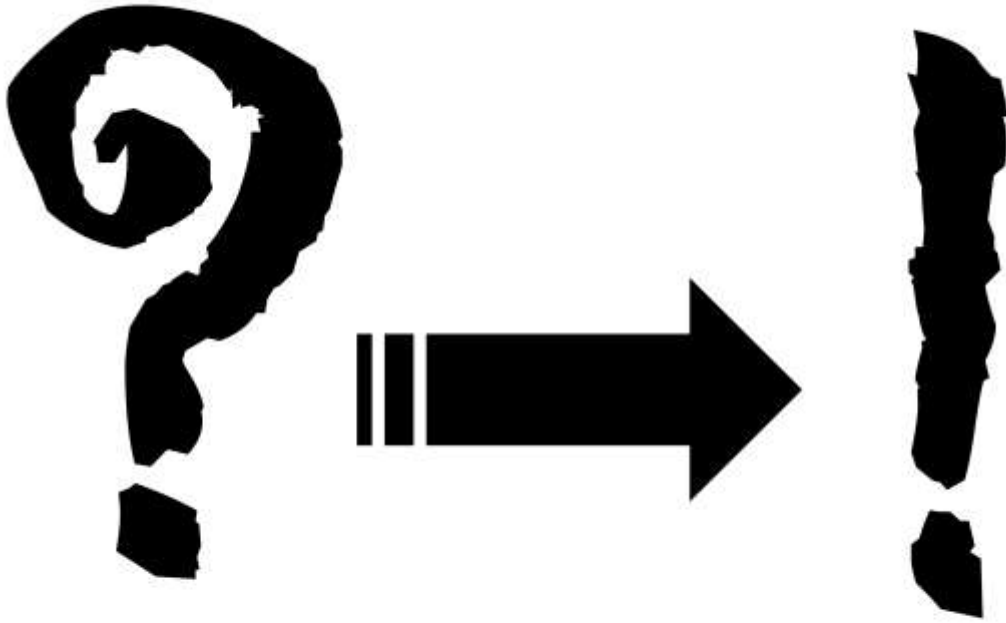
The plethora of complicated test cases just like this one that are typically necessary causes the development of test case automation to lag behind product development to such a degree that by the time we testers can start finding bugs our developers are likely to have moved onto other tasks and be reluctant to switch back to a “completed” task. Similarly, if we uncover fundamental flaws in the feature definition or implementation it may be too late in the product cycle to remedy them. In the race of us against everyone else we cannot even tie, let alone win!



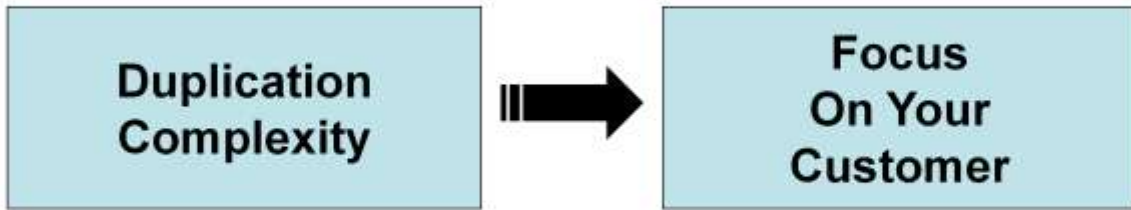
```
Logical.Projects.CreateNewProject();  
  
Logical.SceneElements.CreateRectangle(  
    new Point(100, 100), new Point(300, 300));  
  
Logical.SceneElements.SelectAll();  
  
Logical.Appearance.SetAppearanceProperty(  
    Logical.Fill);  
Logical.Appearance.SetColor(DataManager  
    .FullSpectrumProvider.GetNextValue());  
  
Logical.SceneElements.Cut();  
  
Logical.SceneElements.Paste();
```

Imagine if, rather than one tester poking our head up from among the throng, we could be out ahead of everyone else, scouting the way at all prudent speed.

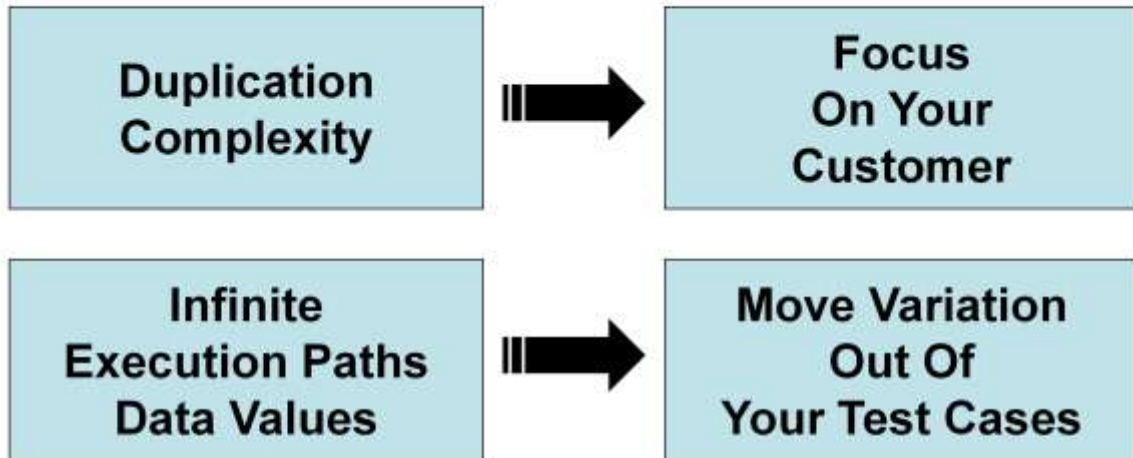
Writing fewer, simpler tests that rarely need to be maintained would let us not only keep up with the pack but get ahead. We would have time to look for those deep, gnarly bugs that are the most difficult to find. We could be testers rather than automators!



Move from large numbers of test cases that all look alike and are hard to understand to fewer more general test cases that cover more surface area of your product with less effort! The myriad problems you must overcome can be grouped into three areas, each with its own solution.

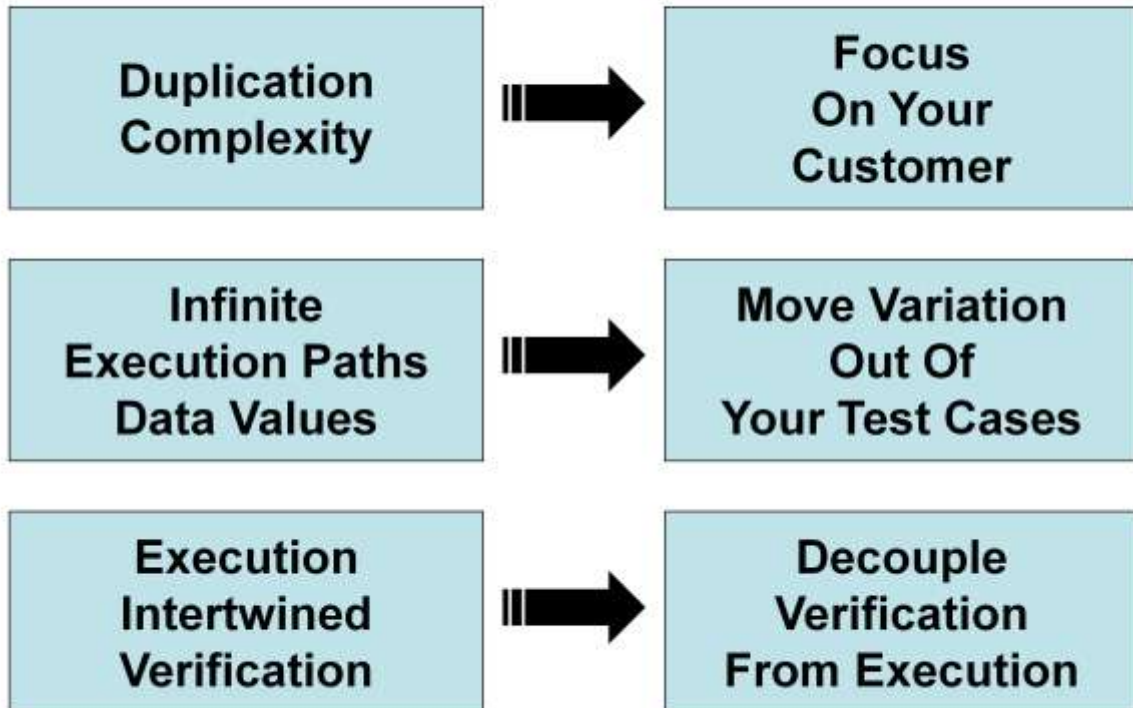


Test cases are full of duplication and complexity that is begging to be factored out, but doing so in a structured way is difficult. Make it simple by focusing on your customer.



The number of execution paths and data values that must be tested is immense and overwhelms your test cases. Free your test cases from this jail by moving variation from your test cases to your test libraries.





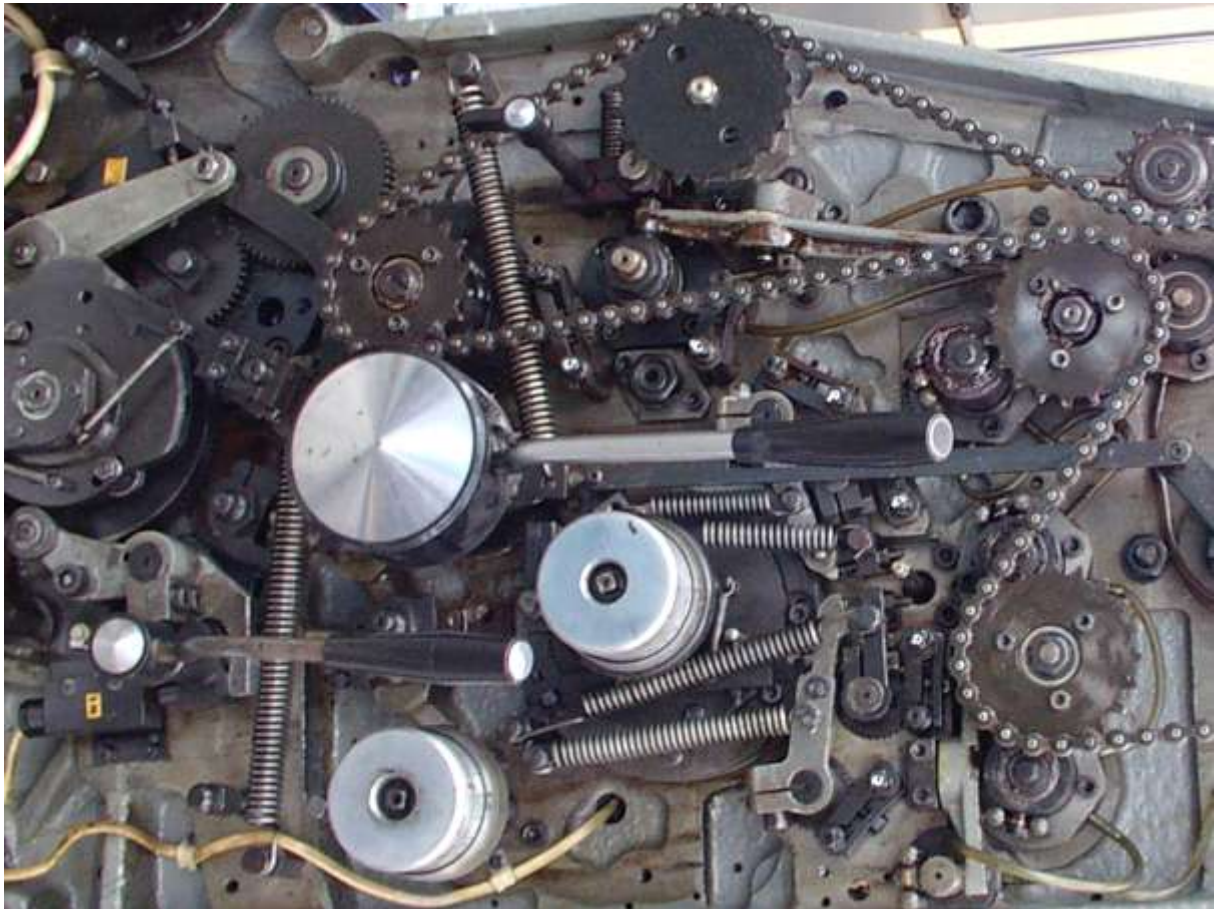
Execution and verification seem to be inextricably tied to each other and so your test cases are tied into knots while attempting to verify the small number of datapoints you manage to verify. Cut through this tangle by decoupling verification from execution.



Scripted test cases tend to be just like this crowd: each one is similar to each other one in many respects. The family of test cases for a particular feature look even more like each other, and sibling test cases may be virtually identical. Unfortunately, these similarities mean duplicated code. When duplicated code is a concern, a common solution is to factor said code out to a helper method. This eases the burden of maintaining this shared code, but as the number of helpers grows the code base starts to have another thing in common with this crowd: it becomes difficult to find the particular method you are looking for.



Data presents a similar problem. Most test cases need a pipe into internal application data structures, but the right pipe to use is not always clear. As the data structures change over time these pipes become convoluted and even harder to follow. Developers cannot always tell whether a particular pipe is being used, so when pipes are ripped out or replaced test cases can find themselves with connections that stop working with no warning or recourse. Testers must be plumbers as well, tracing undocumented pathways back to what appears to be the well of data they require, knowing full well that it may dry up tomorrow.



The point of many scripted test cases is to manipulate an application's user interface. The details of how this is done are often complicated and complex, involving even more chains, springs, pulleys, and levers than this machine uses. The user interface is generally the most changeable part of an application, with springs changing to levers and pulleys becoming chains – not to mention entire sections being removed or replaced - on a daily basis. Keeping test cases current with all this can easily form an overwhelming burden.

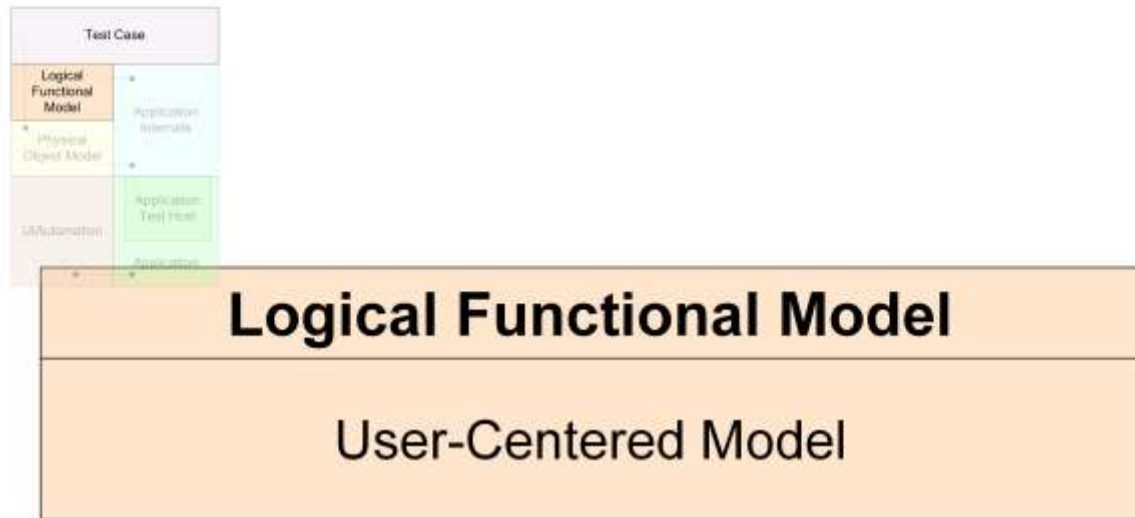
# **Focus On Your Customer**

Mitigate these problems by applying an organizing influence that will stay valid and not degrade over time: your product's customer. You know that the right way to approach building an application is to think about what the people who purchase your application want to do and how they want to do it. Using this as a basis for your development activities helps ensure you create an application your customer actually wants. Extending this principle into your test libraries focuses and provides structure for your Test libraries and test cases.

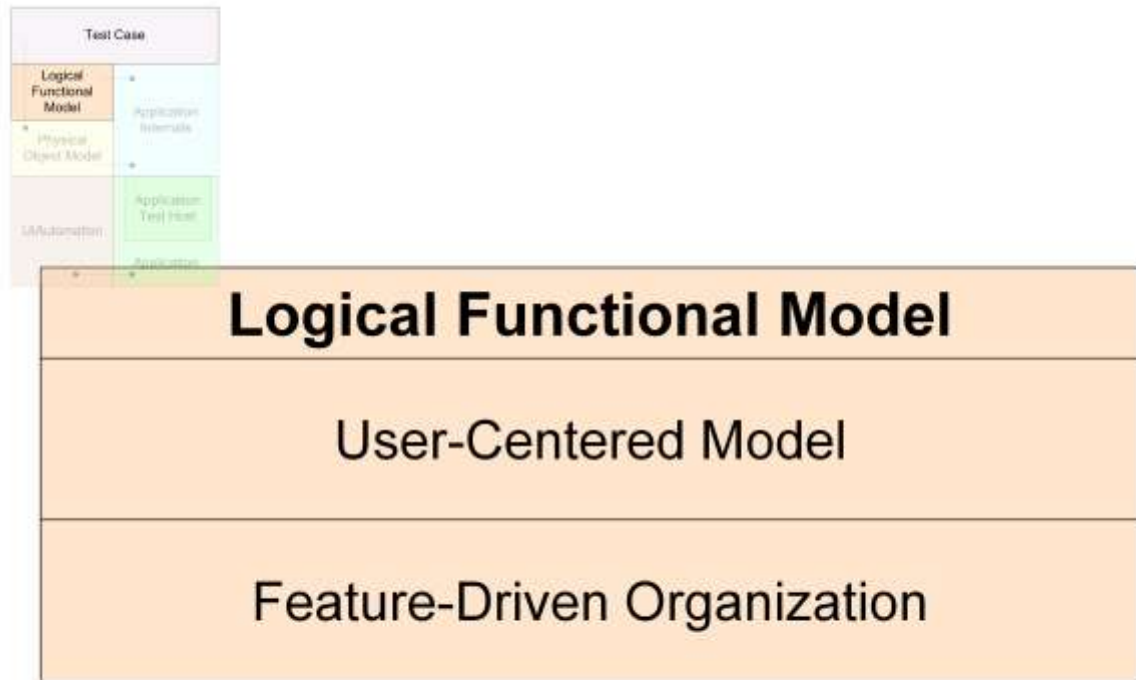
# **Semantics Not Specifics**

## **Logical Functional Model**

The starting point is the Logical Functional Model, or LFM. The LFM is very different from the object model point of view you are likely used to, and it is much more than your standard library of helper methods. The Logical Functional Model works from the user's point of view: what can your user do in your application? What features do they see? The LFM is all about the semantics of your answers to these questions, not the specifics.

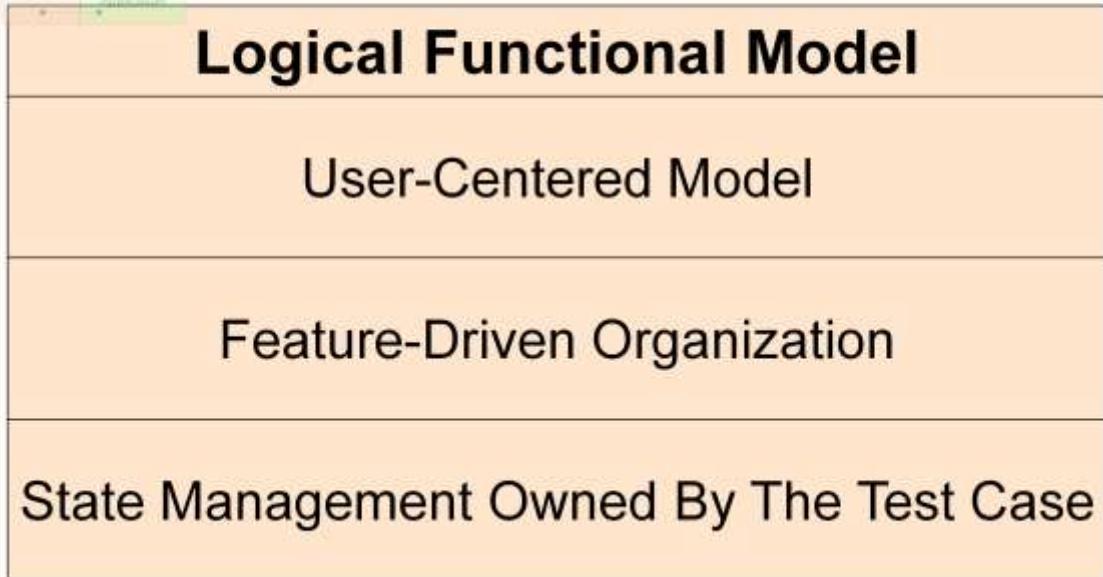
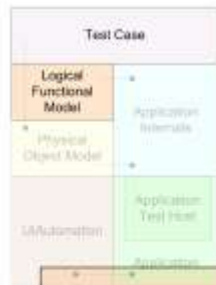


The LFM's user-centric view of an application is very different from the object model view most people are used to. Object models are created so the application can be automated, and so they look at their application the way a programmer would: menus and toolbars and palettes and views. Users, however, typically see the application in a different light: actions they can take. Keeping this user-centric view is harder than you might expect, for the scripting object model mindset is hard to break. Keeping it is however worth the trouble, for when your testing code is organized around user actions, writing a test case is nothing more than listing the actions a user would take. The test case becomes easy to read and easy to understand.



At its core, the LFM is nothing more than a library of common code that has been factored out to helper methods. A common problem with such libraries is finding an organization scheme that holds up over time. The LFM avoids this impending chaos by structuring its contents the same way as your customer groups the features in your application. Just as the users of a graphics application wouldn't see menus and toolbars and views but rather projects and drawings and shapes, the LFM is not organized around menus and toolbars and views but rather around projects and drawings and shapes. The steady influence of your customers' view of your features makes the LFM easy to understand and navigate, even over the course of years.





“Do what your name says you will do or throw an exception” is a rule of thumb object model methods usually live by. The LFM takes a slightly different tack: “Go through the motions necessary to do what your name says you will do”. This difference is the key that enables LFM methods to be used for both positive and negative testing. The LFM knows the actions required to delete the selected elements, but only the test cases can know what the result should be. Thus the responsibility for ensuring the application is in the necessary state must fall upon each test case, not the various LFM methods.

```
UIObject applicationWindow = UIObject.Root.Children;
UIObject mainMenu = applicationWindow.FindDescendant("Main Menu");
UIObject fileMenu = mainMenu.FindDescendant("File");
this.InvokeMenu(fileMenu);
UIObject newMenu = fileMenu.FindDescendant("New");
this.InvokeMenu(newMenu);
UIObject projectMenuItem = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItem);
```

```
UIObject applicationWindow = UIObject.Root.Children;
UIObject mainMenu = applicationWindow.FindDescendant("Main Menu");
UIObject fileMenu = mainMenu.FindDescendant("File");
this.InvokeMenu(fileMenu);
UIObject newMenu = fileMenu.FindDescendant("New");
this.InvokeMenu(newMenu);
UIObject projectMenuItem = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItem);
```

```
UIObject redValue = appearancePalette.FindDescendant("ColorEditor");
redValue.Text = "Red";
UIObject greenValue = appearancePalette.FindDescendant("ColorEditor");
greenValue.Text = "Green";
UIObject blueValue = appearancePalette.FindDescendant("ColorEditor");
blueValue.Text = "Blue";
foreach (FrameworkElement element in this.GetDescendants(rootElement))
{
    this.Log.VerifyValue(element.GetValue(typeof(TextBox))?.Text, "Color element's name should be 'Red'");
}
```

```
Logical.Projects.CreateNewProject();
```

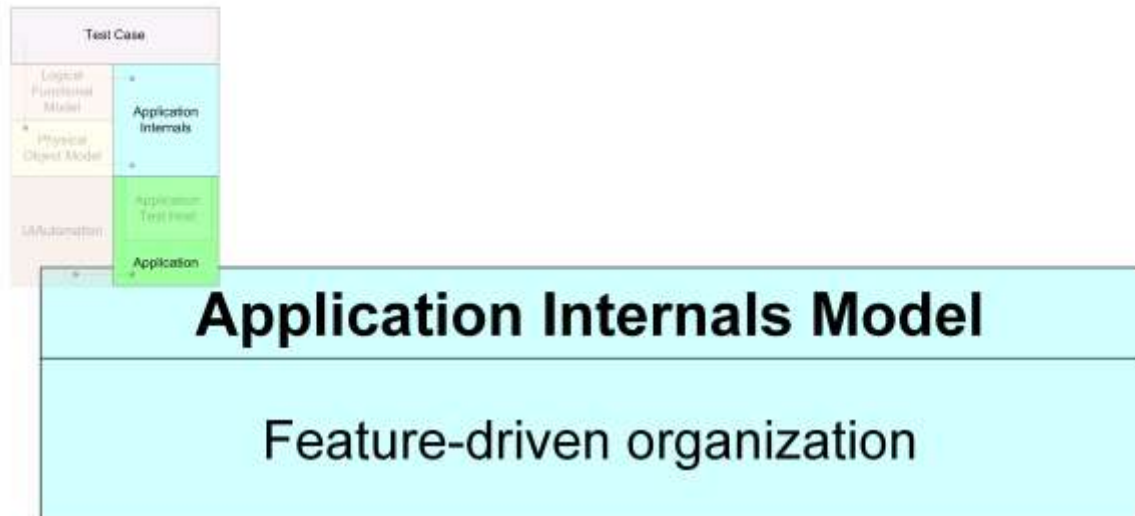
```
this.InvokeMenu(newMenu);
this.WaitForIdle();
UIObject projectMenuItem = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItem);
this.WaitForIdle();
this.Log.VerifyValue(this.GetDescendantCount(rootElement).Count, 2, "Some element count after paste");
```

Here is what happens when we apply a Logical Functional Model to our sample test case. The top example magnifies the first eight lines of code in the test case so that you can actually read them. Even now, however, decoding what exactly they do is difficult. Contrast this with the corresponding LFM call in the bottom – much smaller – rectangle. You can see that using an LFM will dramatically simplify your test cases, and you will be able to more easily discern the point of each. The essence of the test case starts to come through.

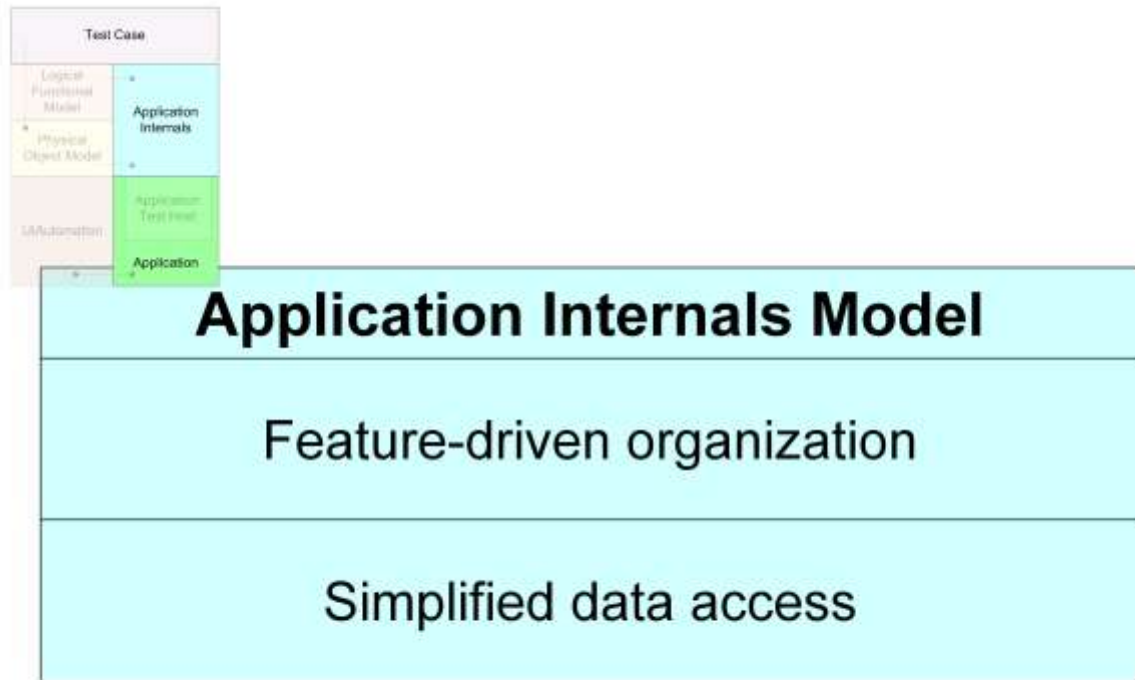
# **Semantics Not Specifics**

## **Application Internals Model**

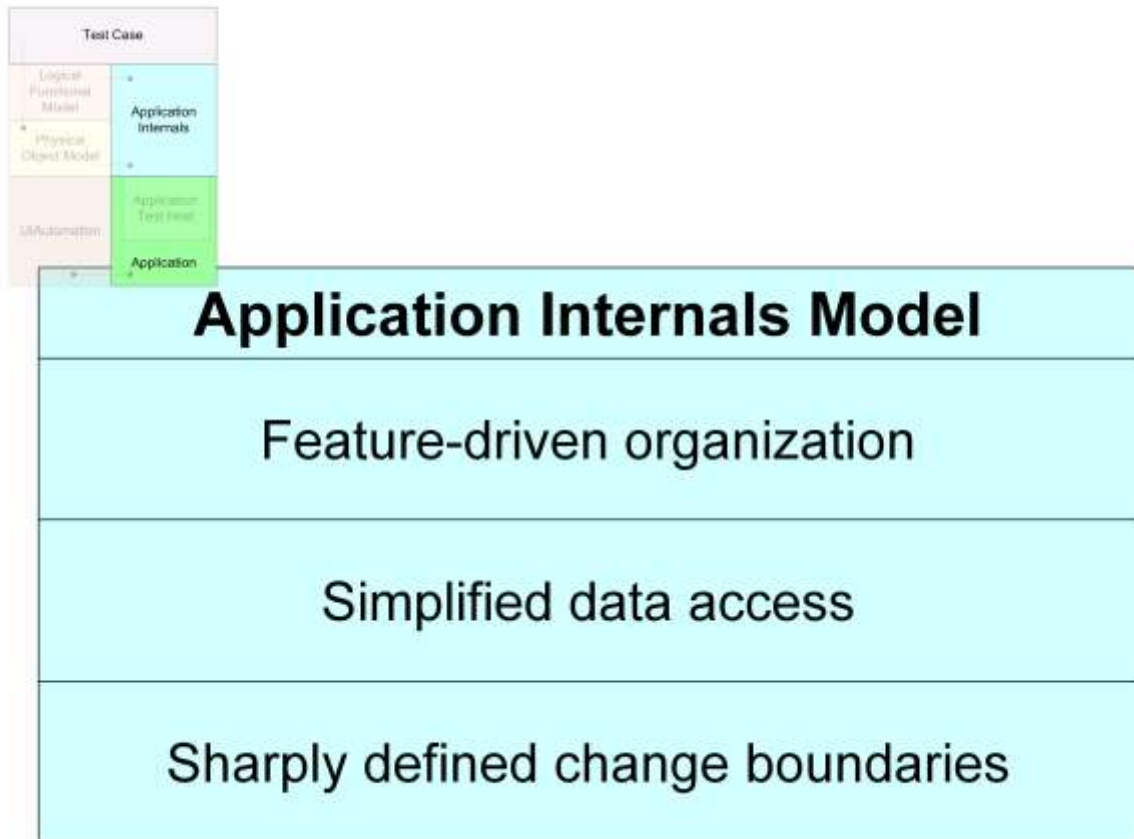
I said earlier that the LFM is very different from a scripting object model. One difference is that as it is focused on user actions it only contains setters. The getters move to a separate Application Internals model. This model follows the LFM's lead and organizes its data retrieval helpers around the same customer features as does the LFM. This feature driven-organization simplifies data access and sharply defines change boundaries. The Application Internals model talks to the semantics of the data you need, not the specifics.



The Logical Functional Model moves execution helpers out of your test cases in an orderly and easily understood fashion. The Application Internals Model does the same for your data retrieval needs. Application Internals maintains the same focus on user features as does the LFM, so once a tester understands the LFM's organization they can navigate the Internals model just as easily. As with the LFM, this approach provides an organizing influence that helps it maintain its integrity in the face of application changes and ongoing maintenance. This technique can be applied to data coming from other sources, such as the operating system, as well.



The Internals models put an easier-to-understand façade on the data they front. Most application's internal data structures are designed to make implementing the application easy, not to make it simple for anyone else to get at particular bits of information. Having data organized in a clear and consistent manner in the same buckets as the rest of the automation system makes working with that data easier and less error-prone. This organization also simplifies retrieving data from areas outside one's particular areas of knowledge. While testers may come to know the ins and outs of inspecting application state for their feature, they are unlikely to have the same knowledge about other areas. Once the Internals models are understood, however, anyone can find the information they require.



Limiting direct access to application and other data structures to the Internals models keeps most changes in those structures from affecting test cases and LFM methods. The value of isolating large changes to a few well-defined areas should be obvious. The Internals models provide value for smaller changes as well, as they allow you to make a few well-targeted changes rather than updating your entire code base. Even at the extremes, where Internals methods are nothing more than a single line of code, the organizing influence of focusing on customer features keeps the library clear and concise.

```
UObject applicationWindow = UObject.Root.Children;
UObject mainMenu = applicationWindow.FindDescendant("Main Menu");
UObject subMenu = mainMenu.FindDescendant("File");
this.InvokeMenu(subMenu);
UObject newMenu = FileMenu.FindDescendant("New");
this.InvokeMenu(newMenu);
UObject projectMenuItems = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItems);
```

```
Application.SceneDocument currentSceneDocument = this.ApplicationManager.DocumentManager.ActiveDocument as SceneDocument;
this.Log.VerifyNotNull(currentSceneDocument, "Have a scene document");
FrameworkElement rootElementForScene = currentSceneDocument.ViewModel.RootElement;
```

```
Application.SceneDocument currentSceneDocument =
    this.ApplicationManager.DocumentManager
        .ActiveDocument as SceneDocument;
this.Log.VerifyNotNull(currentSceneDocument
    , "Have a scene document");
FrameworkElement rootElementForScene =
    currentSceneDocument.ViewModel.RootElement;
```

```
int? sceneElementCount = null;
UObject sceneElements = ApplicationInternals.SceneElements;
this.Log.VerifyValue(sceneElements.Count, 0, "Initial scene element count");
```

```
this.Log.VerifyValue(ApplicationInternals
    .SceneElements.ElementCount
    , 0
    , "Initial scene element count");
```

Here again the top example magnifies just the start of the many lines of code required to find the number of elements on the active scene. All of that detailed knowledge of how to poke through the application's internal data structures is replaced with a single line of code that clearly states what data is being retrieved. When those details change, just one easily identified getter needs to be updated rather than a myriad of test cases. Any tester on your team could easily find this and any other data they might require. The essence of the test case starts to come through.

# **Semantics Not Specifics**

## **Physical Object Model**

The most visible aspect of your application is its user interface. Your UI is the start and end of your application to many of your customers. Test cases tend to start here as well. Focusing on the “how” of the test case – the specific order in which the various widgets are interacted with – rather than the “what” – the point of all that interaction – makes it difficult to pull the test case “forest” out of the UI manipulation “trees”. Building a Physical Object Model that references your UI in terms of the semantics of what you are trying to accomplish rather than the specifics of how it is done makes your test case more clear.





## Physical Object Model

Wrap application UI in an object model

There are two parts to the Physical Object Model. The outer layer is a straightforward object model for your application's user interface. This layer hides the details of finding and manipulating the various widgets and controls in your UI behind clearly and intuitively named methods and properties. Test cases and LFM implementation become simpler to write and easier to understand when they can dot their way around the application UI and ignore details such as the monitor's resolution and how UI controls are identified.



## Physical Object Model

Wrap application UI in an object model

Smudge UI details into similarity

The inner layer of the Physical Object Model is a set of controls abstractions. These abstractions hide the specifics of what type a control is behind a façade that merely exposes the behaviors of which it is capable. Test cases and LFM methods thus do not need to know whether a particular control is a button or a check box but rather only that it can be invoked or have its value set. Thus user interface changes do not affect test cases and LFM methods as long as the UI's semantics stay constant.



## Physical Object Model

Wrap application UI in an object model

Smudge UI details into similarity

Inject variation into execution methods

Generalizing control actions enables the control abstraction layer to completely camouflage all details regarding whether an action is executed via the mouse, the keyboard, or accessibility APIs. Every control has a corresponding Control Provider that knows how to execute its control's actions using its particular input method. With LFM methods and test cases ignorant of how each control is manipulated, a test case can exercise any or all of the execution methods by simply swapping in different Control Providers.

```

UObject applicationWindow = UObject.Root.Children;
UObject mainMenu = applicationWindow.FindDescendant("Main Menu");
UObject fileMenu = mainMenu.FindDescendant("File");
UObject newMenu = fileMenu.FindDescendant("New");
this.InvokeMenu(newMenu);
UObject projectMenuItem = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItem);

```

```

UObject applicationWindow = UObject.Root.Children;
UObject mainMenu = applicationWindow.FindDescendant("Main Menu");
UObject fileMenu = mainMenu.FindDescendant("File");
this.InvokeMenu(fileMenu);
UObject newMenu = fileMenu.FindDescendant("New");
this.InvokeMenu(newMenu);
UObject projectMenuItem = newMenu.FindDescendant("Project");
this.InvokeMenu(projectMenuItem);

```

```

UObject applicationPalette = UObject.Root.Children.FindDescendant("Design AppearancePalette");
UObject brushTypeSelector = applicationPalette.FindDescendant("AppearancePaletteBrush");
brushTypeSelector.FindDescendant("ColorEditor");
UObject redValue = applicationPalette.FindDescendant("ColorEditor");
redValue.Text = "255";
UObject greenValue = applicationPalette.FindDescendant("ColorEditor");
greenValue.Text = "0";
UObject blueValue = applicationPalette.FindDescendant("ColorEditor");
blueValue.Text = "0";
BrushTypeSelector.FindDescendant("ColorEditor");
this.Log.VerifyValue(brushTypeSelector);
}
this.InvokeMenu(editMenu);
this.WaitForIdle();
UObject editMenuItem = editMenu.FindDescendant("EditMenuItem");
this.WaitForIdle();
this.Log.VerifyValue(editMenuItem);
}

```

```

this.ApplicationWindow.MainMenu
.File.New.Project.Invoke();

```

As before, the top example highlights the several lines of code necessary to invoke the File, New, and Project menu items, while the bottom rectangle shows the single line of code required to do the same via a Physical Object Model. As with the LFM and Application Internals, all of the details regarding how that UI is found and invoked are removed from the test case. Wrapping a Physical Object Model around your user interface manipulation code makes working with UI intuitive and discoverable. Smudging the details of exactly how that manipulation is accomplished simplifies your test cases. The essence of the test case starts to come through.

```
Logical.Projects.CreateNewProject();
```

**Logical.Projects.CreateNewProject();**

```
, new Point(300, 300));  
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 1  
    , "Scene element count after add rectangle");  
this.Log.VerifyValue(ApplicationInternals.SceneElements.PrimarySelection  
    , "Rectangle1", "Name of added rectangle");  
  
Logical.SceneElements.SelectAll();  
this.Log.VerifyValue(ApplicationInternals.SceneElements.SelectedElementCount  
    , ApplicationInternals.SceneElements.ElementCount  
    , "Count of selected scene elements");  
  
Logical.Appearance.SetAppearanceProperty(Logical.Fill);  
Logical.Appearance.SetColor(Brushes.Red);
```

**Logical.Appearance.SetAppearanceProperty(Logical.Fill);  
Logical.Appearance.SetColor(Brushes.Red);**

```
Logical.SceneElements.Cut();  
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0  
    , "Scene element count after cut");  
  
Logical.SceneElements.Paste();  
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0  
    , "Scene element count after paste");
```

Here is the test case as it stands at this point. The combination of a Logical Functional Model, an Application Internals model, and a Physical Object Model move much of the complexity and maintenance from every test case to individual library methods where they belong. The test case is no longer bogged down in details of finding and manipulating UI and pulling data out of the application's internal data structures. The essence of the test case is becoming clear.



This monitor accepts input from a variety of sources, but the picture it displays in each case is the same. Similarly, most user actions in an application can be input via different methods, but the end result of each is identical. The infrastructure required for each of these execution variants is typically identical. The entire test case for each variant, in fact, is often completely identical but for how the action is executed. But every path has to be tested, so a test case is necessary for each one.



Every possible execution path must be taken. A representative sample of each data equivalency class must be used. The cross-product of these two axes needs to be tested within every possible context as well. This matrix this produces is usually large – impossibly so. It could be reduced by bringing setup and teardown into the mix, but the resultant management hassles would likely bring more pain than gain. When each test case must decide the direction it is going to take, the easiest path tends to be chosen. Rather than every path being taken one path is taken over and over and over.



An incredible amount of data ends up being locked up in every test case: Which execution path is taken. What data values are used. What parts of the user interface are touched. How that touching is performed. There is no good way to search on this information, so if you need to know which paths or data values have not been covered, or the UI details change, you have no recourse but to open each test case one by one and poke through it hoping to run into the answers for which you are looking.



# **Move Variation Out Of Your Test Cases**

The solution to each of these problems is to have your test libraries provide and direct variation of execution paths and data values. Execution Behavior Manager and Data Manager do just this. They allow you to replace all those look-alike test cases with many fewer tests that focus on what they are testing rather than how they are doing so. Test cases can ignore execution variations and data values they do not care about while still choosing specific paths and values when they do.

# Define Options

Execution Behavior Manager and Data Manager earn their keep by making choices for test cases and LFM methods. This work of course is predicated upon having sets of somethings from which to choose. As you might guess from its name, Execution Behavior Manager works with Composite Execution Behaviors to choose from amongst their child Execution Behaviors. Similarly, Data Manager serves as a clearinghouse from which callers can obtain data-generating Data Providers.

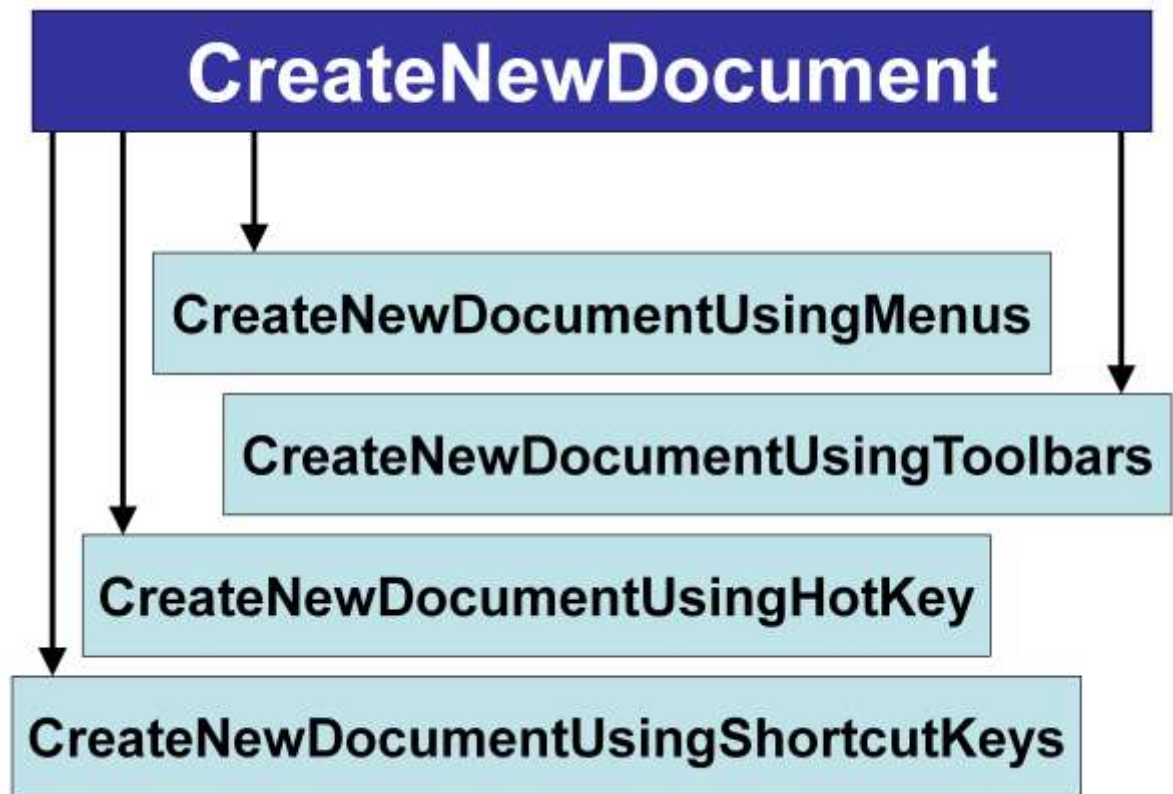
**CreateNewDocumentUsingMenus**

**CreateNewDocumentUsingToolbars**

**CreateNewDocumentUsingHotKey**

**CreateNewDocumentUsingShortcutKeys**

Every Logical Functional Model method is implicitly a child Execution Behavior. As such it requires no special implementation or instrumentation. It can pretty well do whatever it pleases. What it does is not important as far as this part of the system goes. Important here are two things: first, that the child Execution Behavior can be called directly by any test case or LFM method. Second, what doing so means: explicitly calling a child Execution Behavior means that the caller decidedly cares how that action is carried out. In cases like regression tests, you very much want to ensure that a specific execution path is followed. Child Execution Behaviors allow you to do just that.



Although every LFM method is implicitly a child Execution Behavior, certain LFM methods are also Composite Execution Behaviors. A Composite Execution Behavior is an LFM method that knows that it is implemented via one or more semantically equivalent child Execution Behaviors. Rather than actually carrying out an action, Composite Execution Behaviors ask the Execution Behavior Manager to choose one of their child Execution Behaviors, and then they execute the selected child. This is how you get variation over execution paths: each time a Composite Execution Behavior is called, a different child may be chosen. Test cases and LFM methods that call Composite Execution Behaviors make clear that they need something done but do not care how it happens.

```
Logical.Projects.CreateNewProject();
this.Log.VerifyValue(
    , "Initial scene el

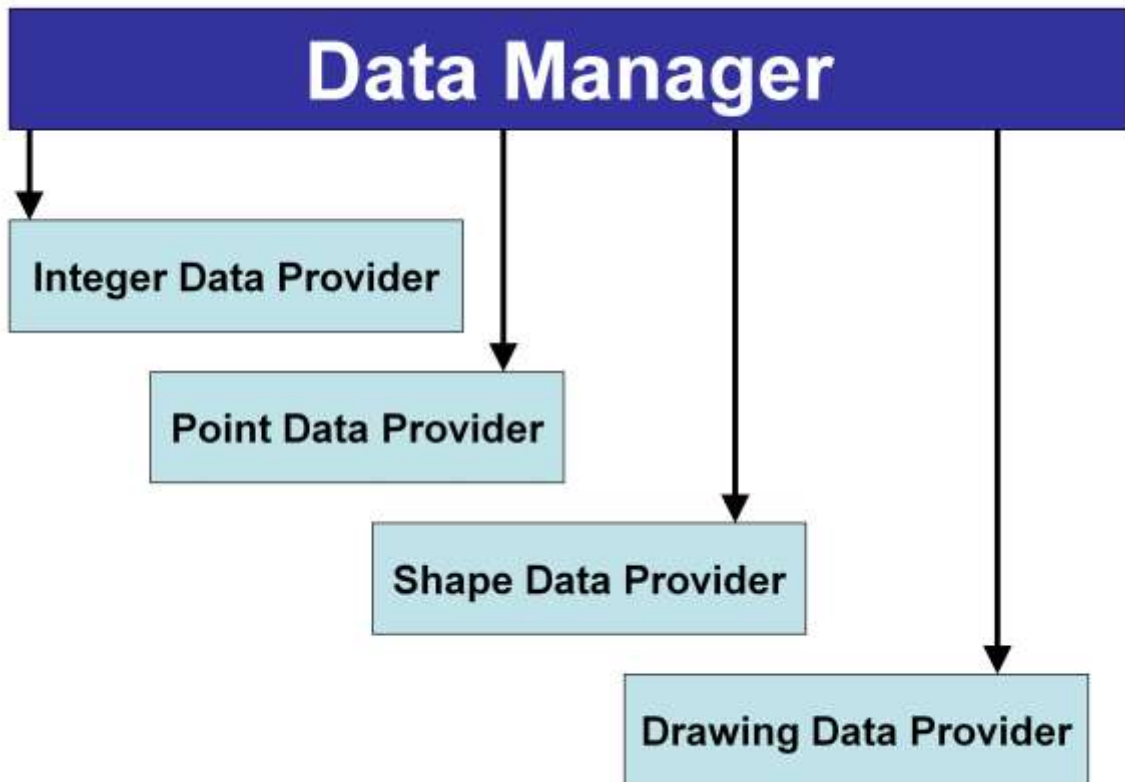
Logical.SceneElements.CreateRectangle(new Point(100, 100), new Point(300, 300));
this.L
this.l
Logi
this.l

Logical.Appearance.SetAppearanceProperty(Logical.Fill);
Logical.Appearance.SetColor(Brushes.Red);

Logical.SceneElements.Paste();
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0
    , "Scene element count after paste");
```

Logical.Projects.CreateNewProject();  
Logical.SceneElements.CreateRectangle(  
new Point(100, 100)  
, new Point(300, 300));  
Logical.Appearance.SetAppearanceProperty(  
Logical.Fill);  
Logical.Appearance.SetColor(Brushes.Red);

Each example here highlights a single line of code in our sample test case. You don't see the dramatic reduction in lines of code that you have seen previously because this test started out being written in terms of Composite Execution Behaviors. Although the code does not become simpler, understanding the test case does. It is now clear that the test case does not care how any of its actions are executed, simply that they are. You can see how adding another option for creating a new project would not require this test case to change, but that this test case would immediately take advantage of that new option. Not only is the essence of the test case coming through, but your testing is becoming more comprehensive as well!



Data Manager and its Data Providers apply these same principles to data generation, but they do so somewhat differently. Because each Data Provider will use different methods for generating the data it provides, Data Manager must delegate much of the selection process to the individual Providers. Also, Data Providers do not have the natural home Execution Behaviors have in the LFM, so Data Manager provides a well-known location from which test cases and LFM methods can look up Data Providers. A final difference is that while Composite Execution Behaviors only ever use a single child at any one time, Data Providers support generating a sequence of test values as well as creating a single value.

```

Logical.Projects.CreateNewProject();
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0
, "Initial scene element count");

Logical.SceneElements.CreateRectangle(new Point(100, 100), new Point(300, 300));
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 1
, "Scene element count after add rectangle");
this.Log.VerifyValue(ApplicationInternals.SceneElements.PrimarySelection
, "Rectangle1", "Name of added rectangle");

Logical.SceneElements.SelectAll();
this.Log.VerifyValue(ApplicationInternals.SceneElements.SelectedElementCount
, ApplicationInternals.SceneElements.ElementCount
, "Count of selected scene elements");

Logical.Appearance.SetAppearanceProperty(Logical.Fill);
Logical.Appearance.SetColor(Brushes.Red);

```

```
Logical.Appearance.SetColor(Brushes.Red);
```

```

this.Log.VerifyValue(ApplicationInternals.Appearance.GetFill(sceneElement), Brushes.Red
, "Scene element " + sceneElement.Name + " fill");
}

```

```

Logical.SceneElements.CreateNewProject();
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0
, "Initial scene element count");

Logical.SceneElements.CreateRectangle(new Point(100, 100), new Point(300, 300));
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 1
, "Scene element count after add rectangle");
this.Log.VerifyValue(ApplicationInternals.SceneElements.PrimarySelection
, "Rectangle1", "Name of added rectangle");

Logical.SceneElements.SelectAll();
this.Log.VerifyValue(ApplicationInternals.SceneElements.SelectedElementCount
, ApplicationInternals.SceneElements.ElementCount
, "Count of selected scene elements");

Logical.Appearance.SetAppearanceProperty(Logical.Fill);
Logical.Appearance.SetColor(DataManager.FullSpectrumProvider
.GetNextValue());

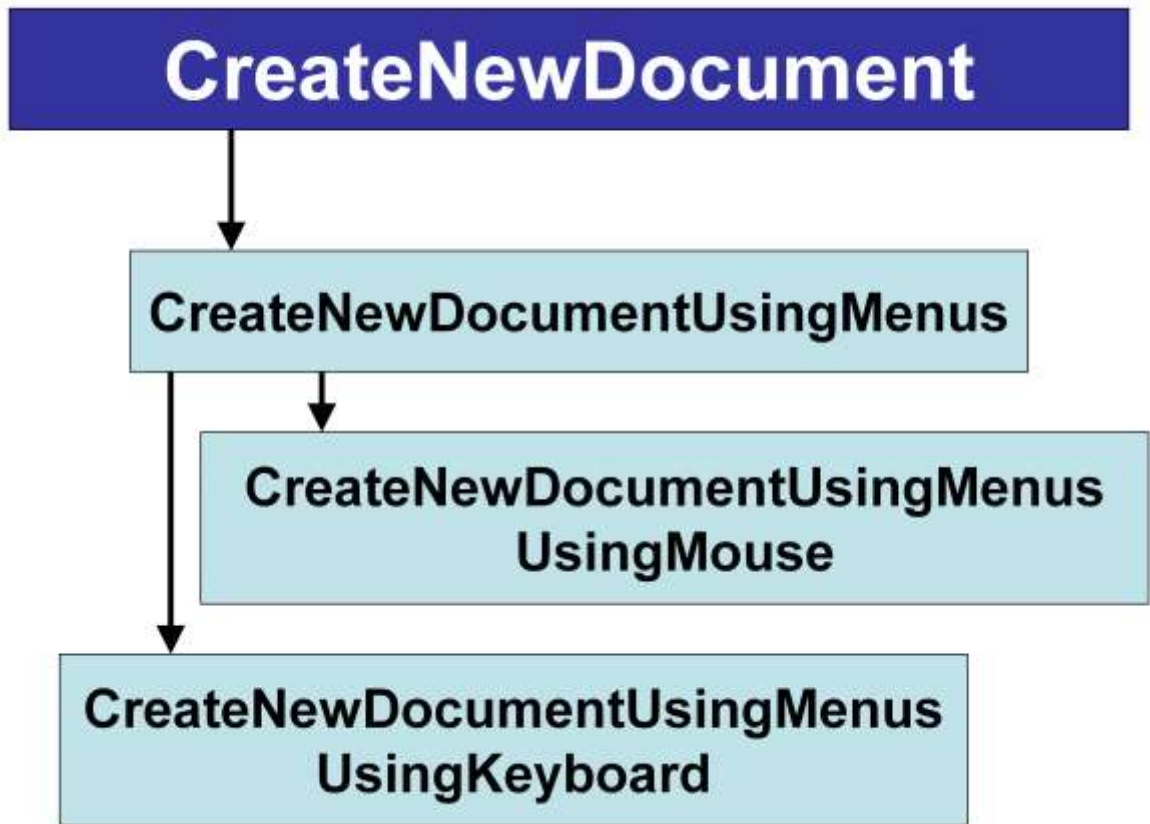
```

Converting the test case to use Data Providers initially seems to have made things worse, for the “before” code is much shorter than the “after” code is. Although the code is now longer it also clarifies the intent of the test case. For example, it is clearly important that a rectangle is created and not any other type of shape, but the exact location of that rectangle is clearly not important. You can modify equivalency classes or add specific values secure in the knowledge that most test cases will immediately start using the revised datapoints while those test cases that require specific values will be unaffected. Just as with Execution Behaviors, the essence of the test case is coming through and your testing is becoming more comprehensive.

# Define Options **Plan** **Choices**

Composite and Child Execution Behaviors work with Execution Behavior Manager to extend the reach of your test cases to cover the entire breadth and depth of your application. Data Manager and Data Providers allow you to free your test data from the shackles of individual test cases and move them to a central location where every test case can easily take advantage of them. Already you are ahead of the game, but both Execution Behaviors and Data Providers help you move even further by being composable, customizable, and disabling dynamically.





Recall that a Composite Execution Behavior is simply an LFM method that delegates its implementation to its set of child Execution Behaviors, and that every LFM method is automatically a child Execution Behavior. Thus it follows that every Composite Execution Behavior can also be a child Execution Behavior – that is, Composite Execution Behaviors can be composed within other Composite Execution Behaviors. This allows a complex tree of execution possibilities to be easily built yet still be easy to understand.

Similarly, because a Data Provider is simply an object that generates one or more data values, any Data Provider can be used by any other Data Provider. Thus variation can be injected throughout the entire depth of the data generation process with ease.

# [ Test Case ]

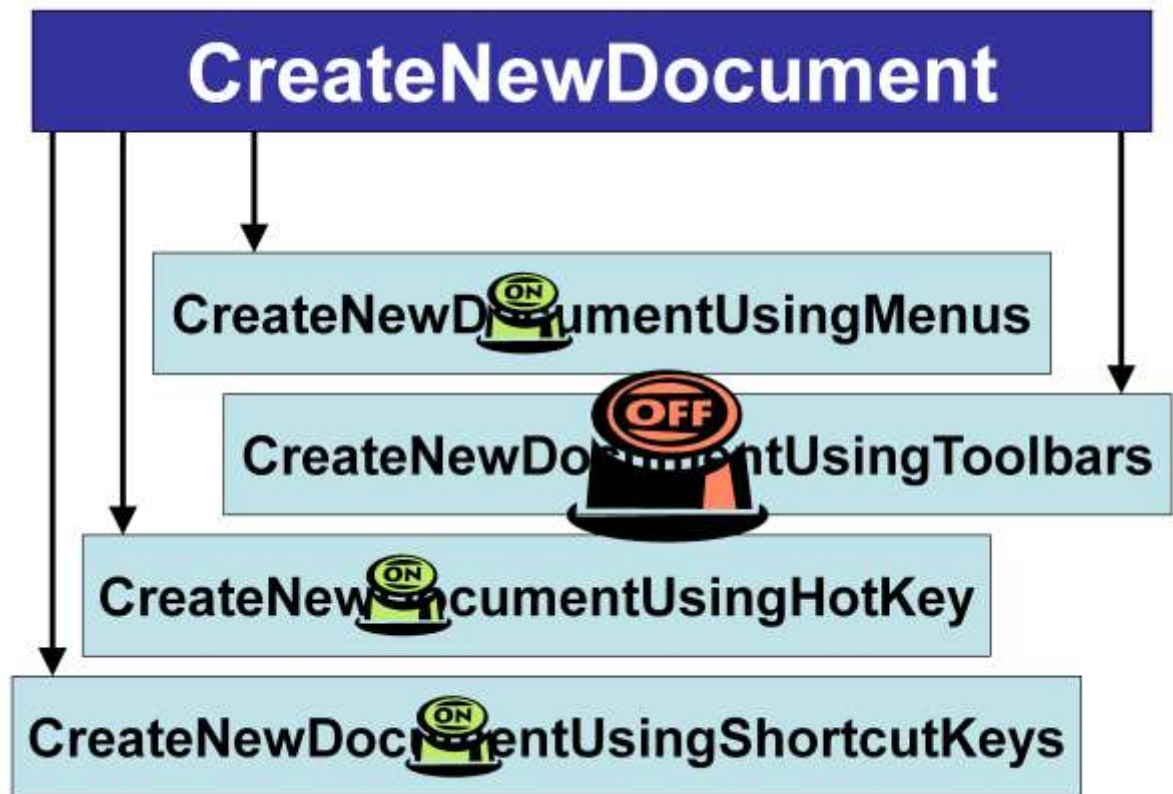
**CreateNewDocumentUsingMenus**

**CreateNewDocumentUsingToolbars**

**CreateNewDocumentUsingHotKey**

**CreateNewDocumentUsingShortcutKeys**

Execution Behaviors help isolate test cases from the details of how actions are executed; Data Providers remove all knowledge of how data is generated and the set of possible values. However, an individual test case may require a specific execution method or data value. Test cases can customize Execution Behaviors on a simple level by directly executing a child Execution Behavior, forcing that specific path to be taken. Test cases can customize Data Providers similarly, by directly utilizing specific Data Providers or even by hard-coding their test data. Both approaches can be intermingled within a single test case, allowing the best strategy to be used at all times.



Further extending the power of Execution Behaviors is their ability to be dynamically disabled. An individual Execution Behavior may only be executable when the application is in a specific state, and it obviously will not have the desired effect if the application is not in that required state. On the other hand, many Execution Behaviors are always available for use. Each Execution Behavior can flag its availability as being dynamic. When the Execution Behavior Manager is asked to choose from a Composite Execution Behavior's set of child Execution Behaviors, it makes its selection only from that subset of children that are available at that exact moment.

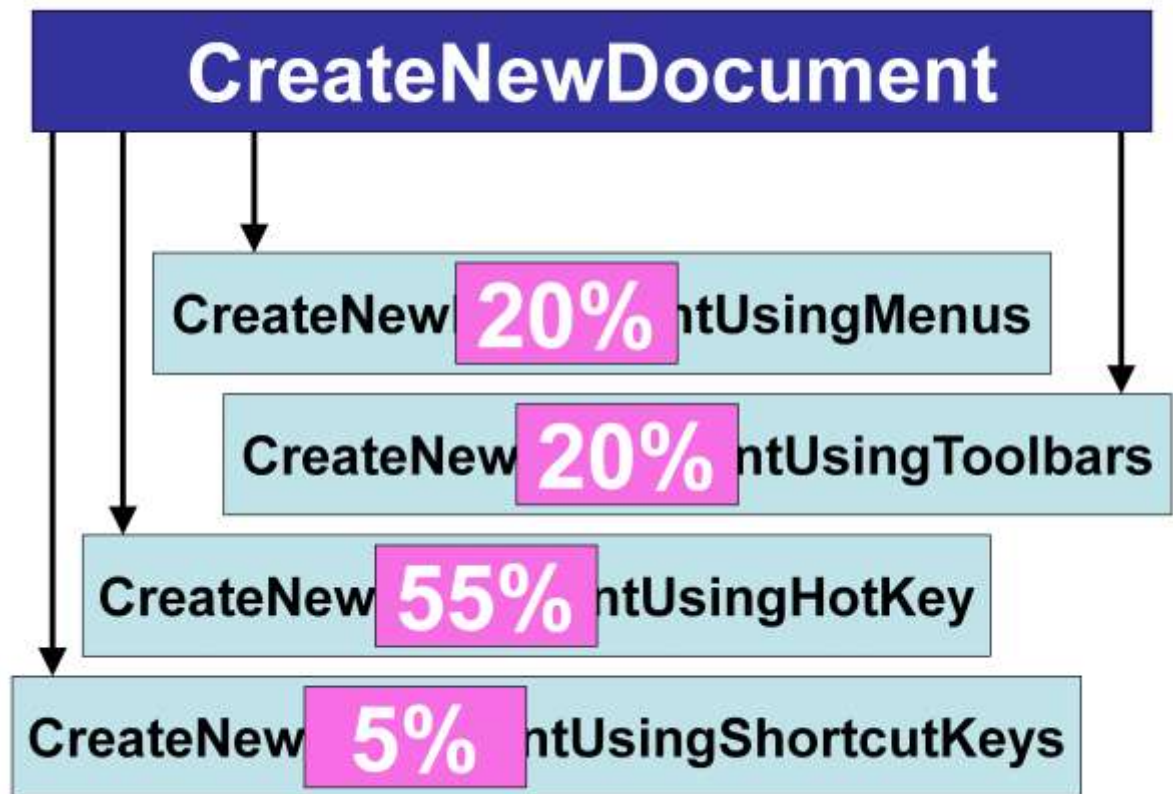
**Define**  
**Options**  
**Plan**  
**Choices**  
**Manage**  
**Runtime**  
**Selection**

I mentioned earlier that Execution Behaviors and Data Providers support a simple level of compile-time customizability by allowing individual child Execution Behaviors and Data Providers to be called directly. This allows individual test cases to manage their environment, but some choice selection management must take place at higher scopes. Both Execution Behaviors and Data Providers support more sophisticated levels of customization at runtime, where choices can be replayed, choices can be biased, and guarantees that every choice that can be selected is.

<b><u>Test Run</u></b>
<b>Test Case 1</b> CreateNewDocument ...UsingMenus CreateShape ... UsingMouse ... UsingKeyboard ... UsingKeyboard ... <b>Test Case 2</b> CreateNewDocument ...UsingShortcutKeys ...UsingToolbars ...UsingToolbars

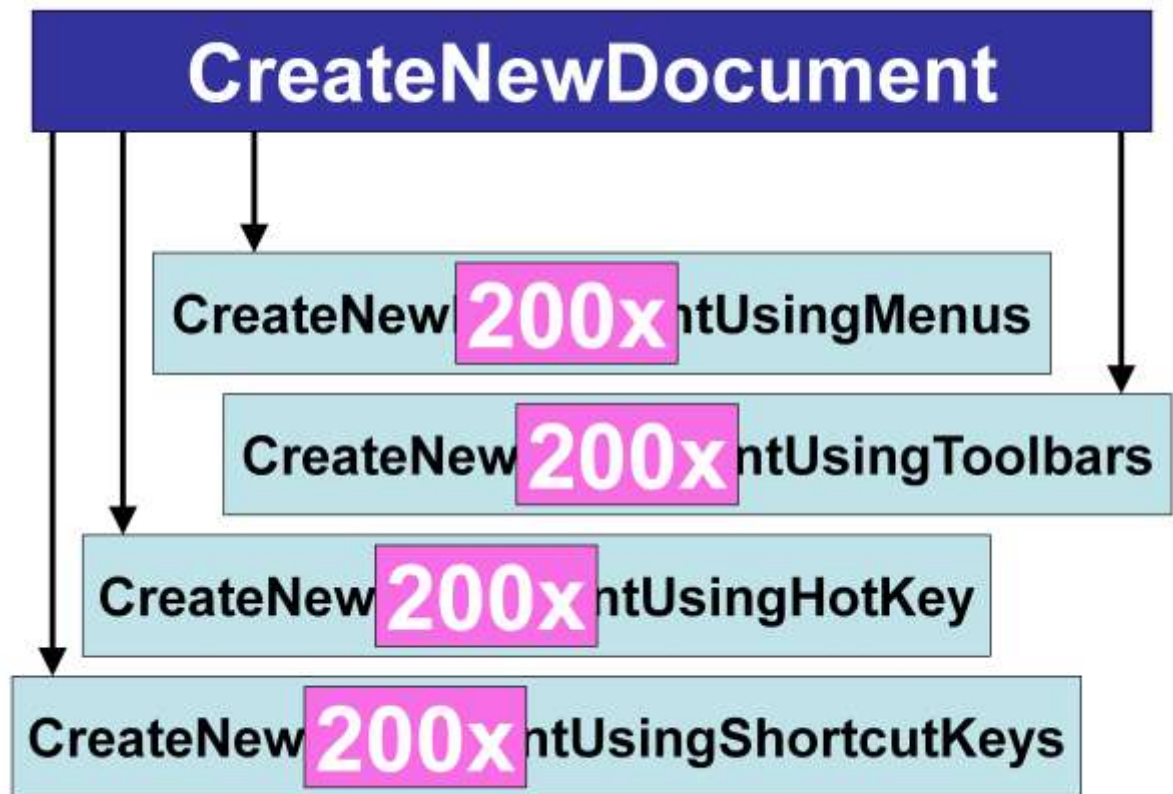
<b><u>Replay</u></b>
<b>Test Case 2</b> CreateNewDocument ...UsingShortcutKeys ...UsingToolbars ...UsingToolbars
<b>Test Case 2</b> CreateNewDocument ...UsingShortcutKeys ...UsingToolbars ...UsingToolbars
<b>Test Case 2</b> CreateNewDocument ...UsingShortcutKeys ...UsingToolbars ...UsingToolbars

In a world where most test cases delegate choosing execution paths and generating test data to Execution Behavior Manager and Data Manager, every time a test case is run it is likely to take a different path using different data than the last time it was run. This automatic variation provides breadth of coverage, but it also makes Replayability essential. Each time a test case is executed, a replay script is generated in which every choice Execution Behavior Manager and Data Manager make is recorded. Running this script executes the test case using exactly the same paths it took and values it used originally. Instant regression tests!



Earlier I explained how each child Execution Behavior can dynamically disable itself based on application state. Additionally, each Execution Behavior can declare metadata about itself, such as whether it uses the mouse or keyboard. Global weighting factors can then be applied across an entire test run to skew the Execution Behavior selection process in specific directions along these axes. Test cases and LFM methods can do the same on a smaller scale by defining local weighting factors.

Data Providers can be biased as well. Many Data Providers define equivalency classes for the data they produce; global and local weighting factors can be used to cause certain equivalency classes to be selected more or less often than the others.



Even with Execution Behavior Manager and Data Manager directing the action, over the course of any particular test run it is unlikely that every possible execution path and data value will be selected. Each subsequent test run, however, increases the chance that each execution path and data value – and combinations thereof – will be selected not just once but many times over. Execution Behavior Manager and Data Manager ensure full coverage by remembering the choices they make each run and using that historical data as the basis for their decisions in later runs. Even the far edges of your application and most esoteric cross-feature interactions are sure to be well-exercised after months of test runs.

```

Logical.Projects.CreateNewProject();
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0
, "Initial scene element count");

Logical.SceneElements.CreateRectangle(DataManager.ScenePointProvider.GetNextValue()
, DataManager.ScenePointProvider.GetNextValue());
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 1
, "Scene element count after add rectangle");
this.Log.VerifyValue(ApplicationInternals.SceneElements.PrimarySelection
, "Rectangle1", "Name of added rectangle");

Logical.SceneElements.SelectAll();
this.Log.VerifyValue(ApplicationInternals.SceneElements.SelectedElementCount
, ApplicationInternals.SceneElements.ElementCount
, "Count of selected scene elements");

Logical.Appearance.SetAppearanceProperty(Logical.Fill);
Color fill = DataManager.FullSpectrumProvider.GetNextValue();
Logical.Appearance.SetColor(fill);

```

```

Logical.Appearance.SetAppearanceProperty(Logical.Fill);
Color fill = DataManager.FullSpectrumProvider
.GetNextValue();
Logical.Appearance.SetColor(fill);

```

```

, "Scene element count after cut");

```

```

Logical.SceneElements.Paste();
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0
, "Scene element count after paste");

```

Here is the test case as it stands at this point. Applying Execution Behaviors and Data Providers makes obvious where specific execution methods and data values matter and where they do not. Information about which execution paths are taken and which data values are used has been freed from the jail of test cases and released into Execution Behavior Manager and Data Manager, where every test case can take advantage of that information. The essence of the test case is becoming clear.





Although execution usually gets top billing, the verification a test case performs is arguably more important: it doesn't matter how many hoops a test case sends the application through if the results are not accurately and thoroughly verified. Knowledge of what test actions are to be executed is paramount to effective verification, so execution and verification each become as tightly coupled to the other as are these train cars. Hard-coding verification within itself gives the test case everything it needs to execute, but it also makes reuse of either the execution steps or the verification nearly impossible.



Regardless of whether verification is done inline to the test case or in helper methods, verification code becomes tangled up with execution code throughout the test case. Initial state is gathered and expected state calculated before each operation; verification of what actually happened must necessarily take place immediately after each operation. Separating verification code from execution code – be that to factor common code out to helpers or just to make a change – is just as difficult as would be separating one thread out of the hairball of ropes above.



Ideally test cases would verify every state value after every operation. Doing so would be akin to attempting to purchase every one of the widgets above individually: a time-consuming and frustrating task. Instead, each test case verifies only a small number of properties that it thinks are important and ignores the rest, just as you would purchase only those widgets that you think you will need, because the cost of doing otherwise is too high. Experienced testers, though, know that those ignored properties are exactly where the most insidious bugs manifest themselves, just like the widget you don't buy is always the one you need.

# Decouple Verification From Execution

Imagine for a moment that you could

- get all the benefits of checking every property after every action while avoiding the tedium of explicitly acting to do so,
- verify everything all the time without requiring every test case be visited each time an expected result changed, and
- modify the definition of "everything" without having to update every test case.

If you lived in such a world, you could catch bugs you currently miss. Far from being an alternate reality, that world can be reached by decoupling verification from execution.

```
Logical.Projects.CreateNewProject();  
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0  
    , "Initial scene element count");
```

```
Logical.SceneElements.CreateRectangle(DataManager.ScenePointProvider.GetNextValue()  
    , DataManager.ScenePointProvider.GetNextValue());  
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 1  
    , "Scene element count after add rectangle");  
this.Log.VerifyValue(ApplicationInternals.SceneElements.PrimarySelection  
    , "Rectangle1", "Name of added rectangle");
```

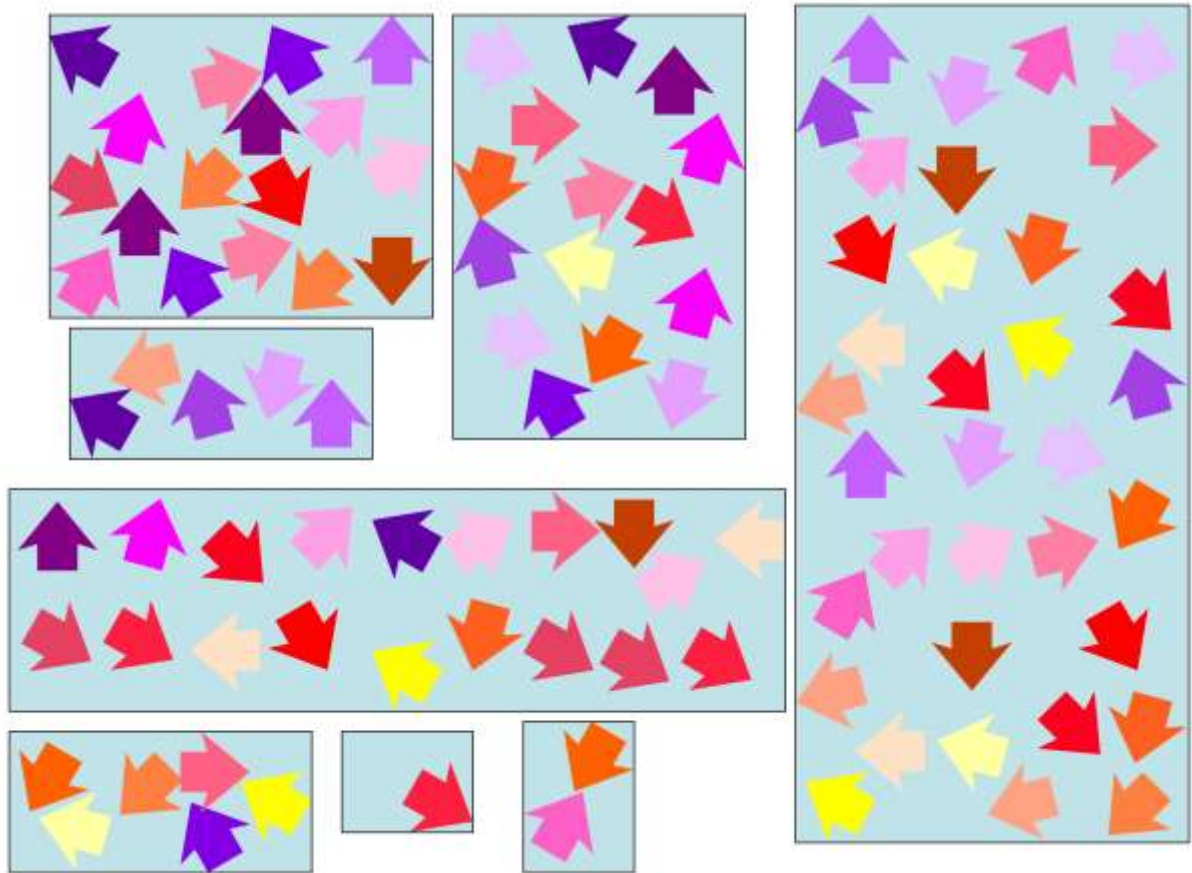
```
Logical.SceneElements.CreateRectangle(  
    DataManager.ScenePointProvider.GetNextValue()  
    , DataManager.ScenePointProvider.GetNextValue());  
this.Log.VerifyValue(  
    ApplicationInternals.SceneElements.ElementCount  
    , 1, "Scene element count after add rectangle");  
this.Log.VerifyValue(  
    ApplicationInternals.SceneElements.PrimarySelection  
    , "Rectangle1", "Name of added rectangle");
```

```
this.Log.VerifyValue(ApplicationInternals.SceneElements.ElementCount, 0  
    , "Scene element count after paste");
```

The change we are looking for here is simple: the removal of all verification code from the test case. With verification, execution path decisions, and test data generation moved into your test libraries, and with action execution, application data retrieval, and UI manipulation details moved there as well, not much would be left in the test case. Your tests would finally be reduced to their essence.

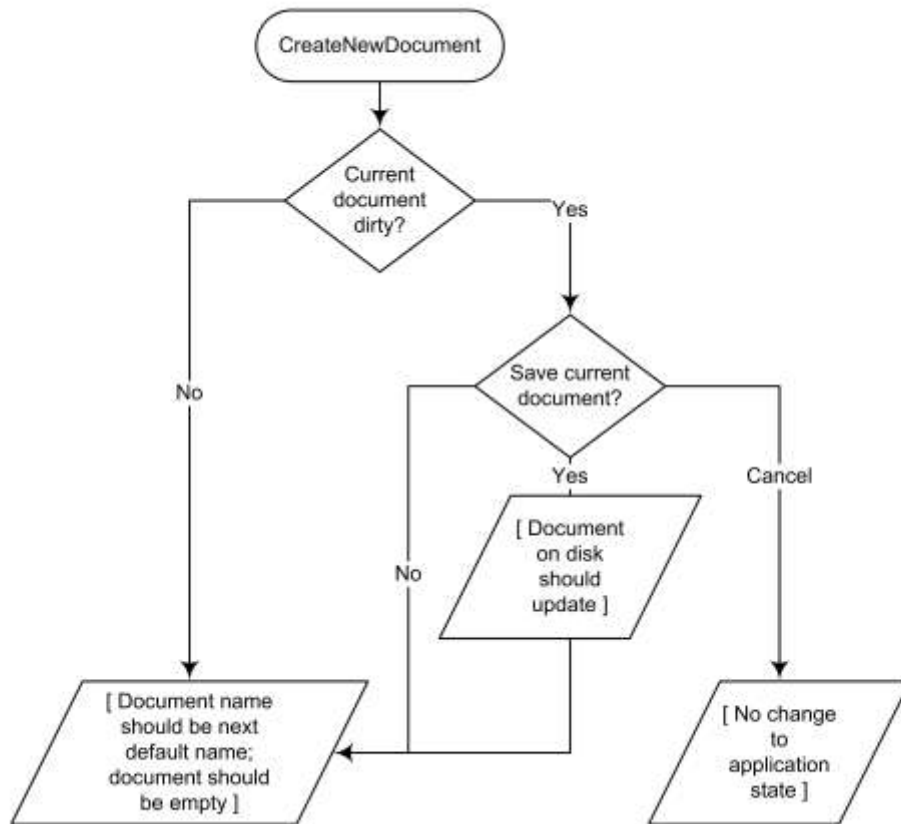
# Who

Decoupling verification from execution may seem difficult. Extracting the tangles of verification from your test cases may seem likely to be a chore. Both can indeed be the case, and without a doubt you have some work ahead of you. The Beatles got by with a little help from their friends, and so will you. Three friends in particular will be of use: clearly defined verification scope, models of what should happen within that scope, and state to track it all.



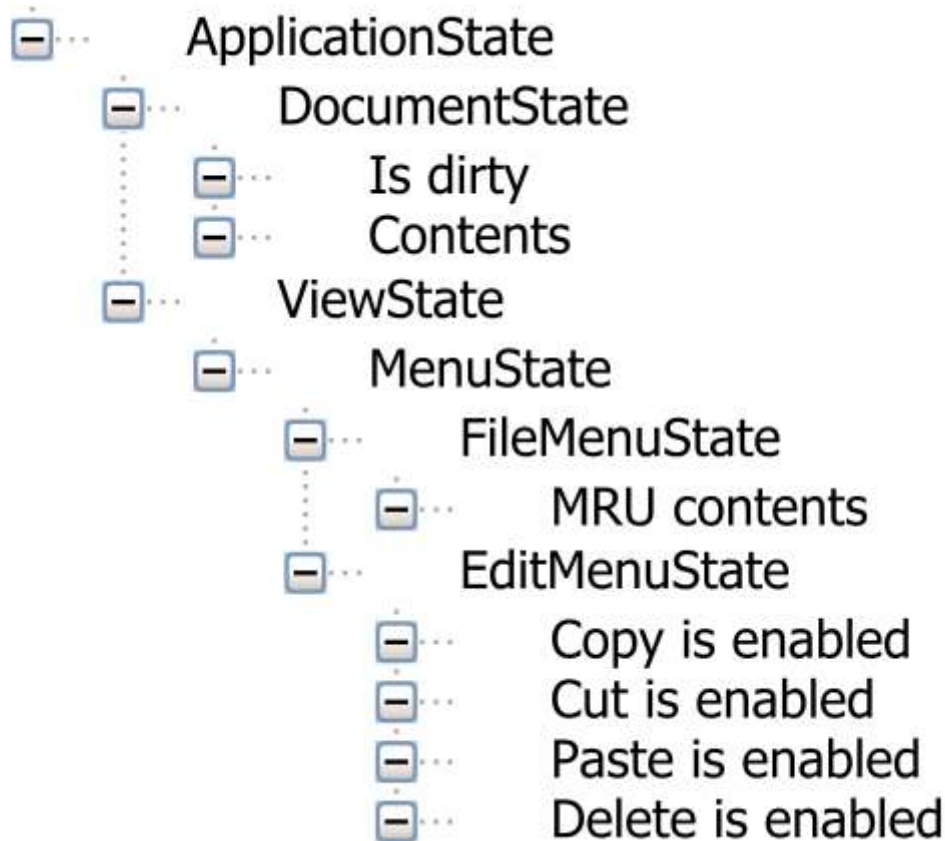
Think back to our sample test case as it was at the beginning. One of the reasons scripted test cases become so complicated and convoluted is that they do not form boundaries amongst their various actions. Just as dye quickly makes its way into every nook and cranny when it is added to a bottle of water, verification code naturally surrounds every bit of execution code. Verification you need to do here also needs to be done there, and soon the tangled mess we love to hate appears.

Start to corral this into some sense of order by drawing boundaries around areas with deterministic behavior. These scoped areas are vital to Comprehensive Verification because they define regions for which expected results can be consistently determined.



Scoped areas of code with clearly defined boundaries are the first set of friends with whom you must acquaint yourself. The next set of friends to gather are models of what should happen within each of those limits. Because each Expected State Generator is focused on a very specific part of your application, they can be small, easily written, easily maintained, and easily understood. Clearly defining their areas of responsibility allows each to be isolated from the others, which further simplifies their creation and maintenance and improves their understandability.





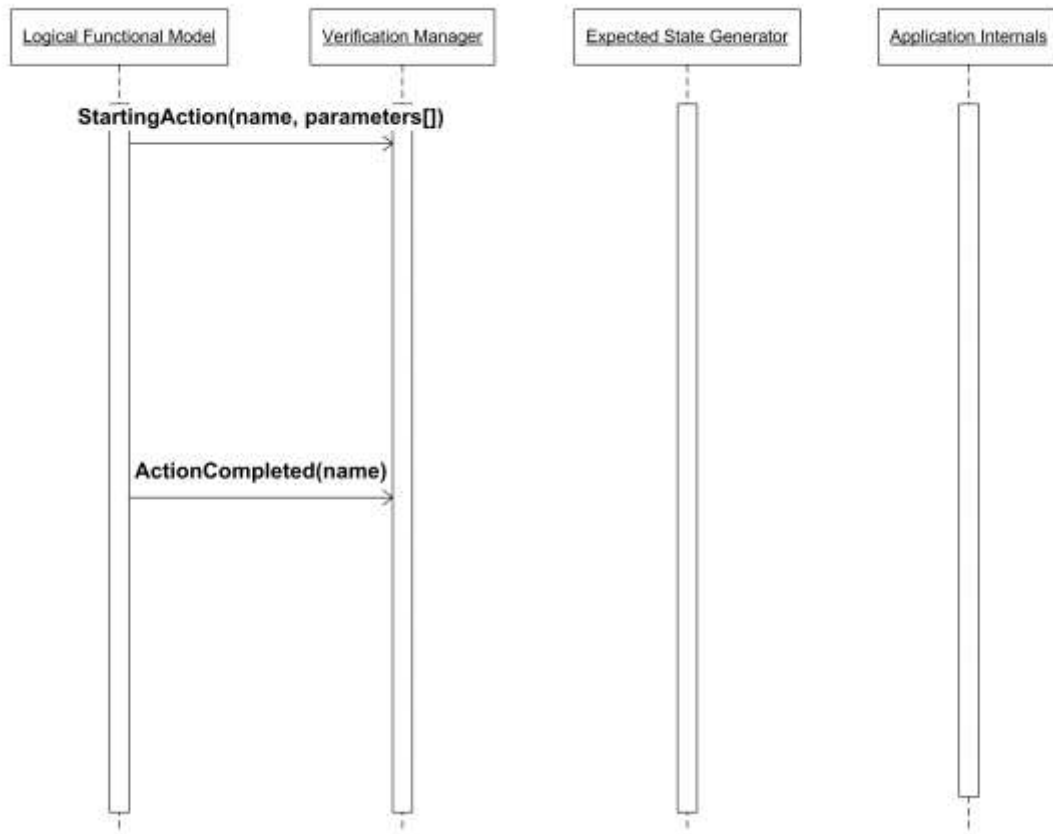
The first set of friends you gathered was clearly defined scopes of execution. The next set of friends was models of what should occur within each scope. From the combination of these two groups comes the last set of friends you need: state models in which to record the expected results of each action within each scope.

Comprehensive Verification enables you to verify everything all the time, and this state model is how you define what “everything” is. It can take on whatever structure you like, but simpler is better.

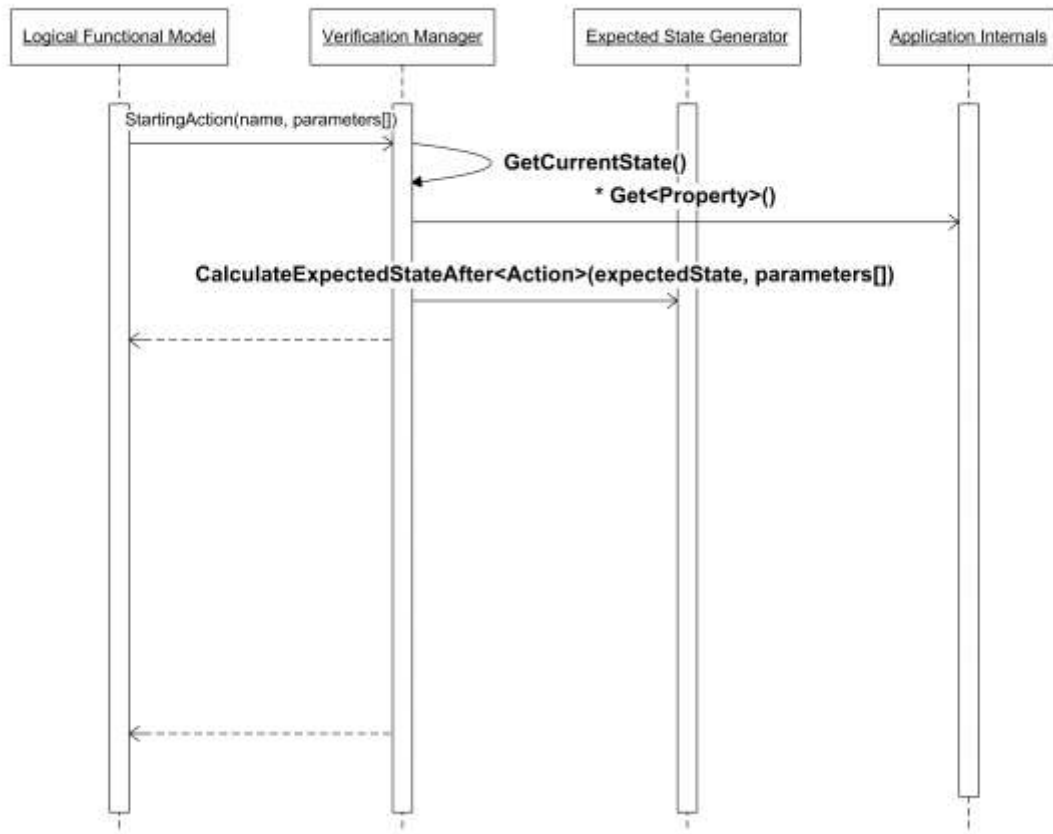
# Who

# How

Heretofore I have purposely not gone into very many details regarding implementation, for those details are mostly irrelevant to the concepts I have been presenting. Loosely Coupled Comprehensive Verification is slightly different in this regard. I promise not to go too deep – so even you managers can follow along, but understanding the basic mechanisms underlying Comprehensive Verification is important to understanding the benefits it brings. As you will see, those mechanisms are straightforward: events to loosely couple verification with the rest of the system, continuous baselining of application state, and the obvious comparing of expected and actual state.

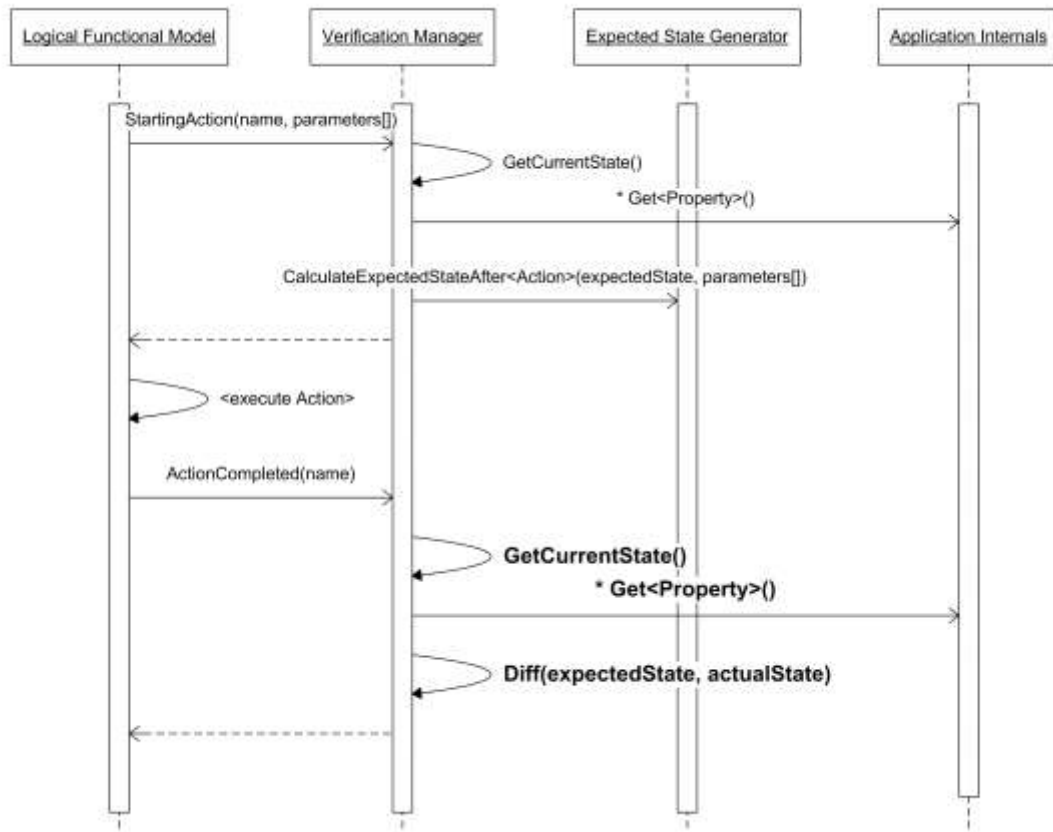


Verification in scripted test cases is typically tightly coupled to each test case. Loosely Coupled Comprehensive Verification eliminates this coupling almost completely, reducing it to just two points: the beginning and end of each scope. Each time a verification scope is entered or exited, that code sends an event to verification announcing the fact. These two events are the only point of coupling to the Verification subsystem – nothing else happens in the test case or LFM as far as verification is concerned.



When Verification receives an `OperationStarting` event it initializes an instance of the state model with the application's current state. This constant baselining ensures that the verification models always base their calculations on the most up-to-date data.

That freshly-built baseline is next passed around to every verification model, whereupon each model goes to work determining what effect the in-progress operation should have on the application's state. These calculations tend to be short and simple as most actions affect only a small part of an application. Complicated calculations, in fact, are often signs that a model's focus or an operation's scope needs to be reduced.



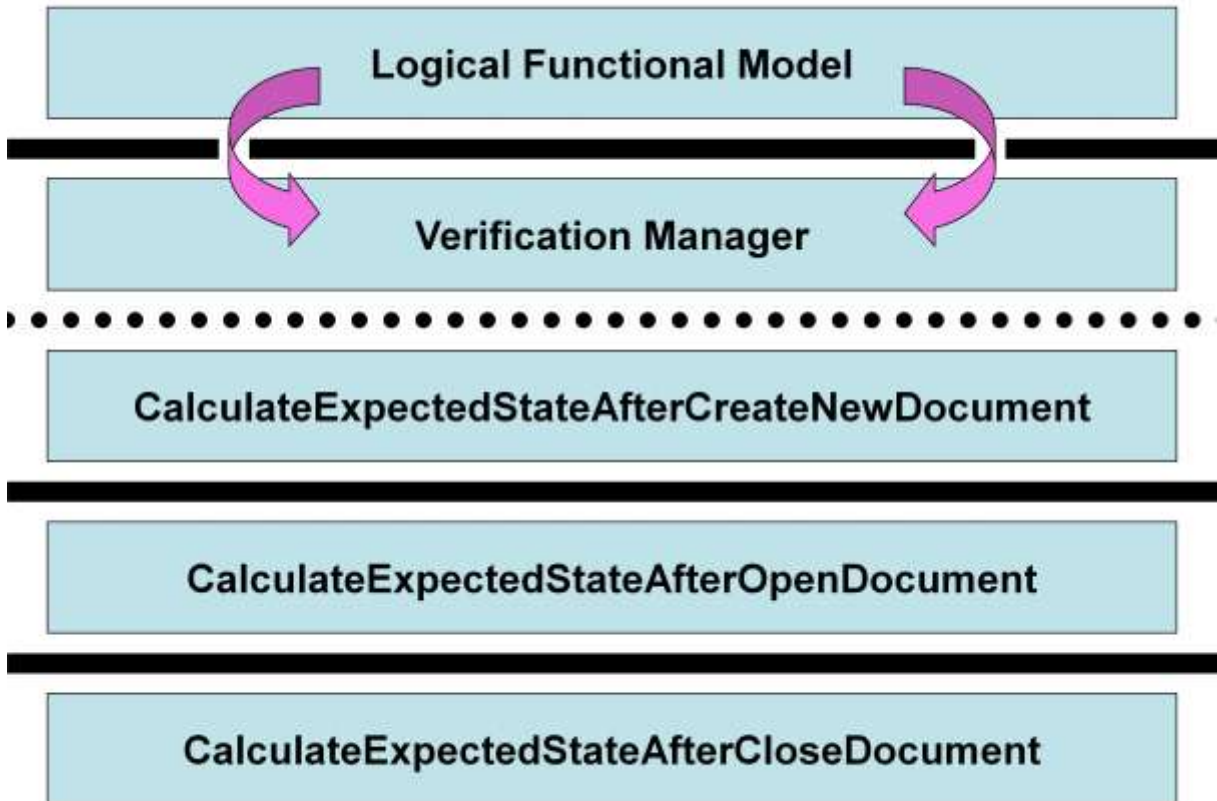
Once all verification models have completed their expected state calculations Verification sits quiescent until the OperationCompleted event comes in. What happens at this point is probably obvious: Verification grabs a copy of the current – which is to say, actual – application state, compares it against the expected state generated earlier, and logs any differences as failures. The only part of this process that even approaches a “wrinkle” is that parts of the expected state can be set to “I don’t know what this should be”, which cases need to be handled specially.

# Who

# How

# **Why**

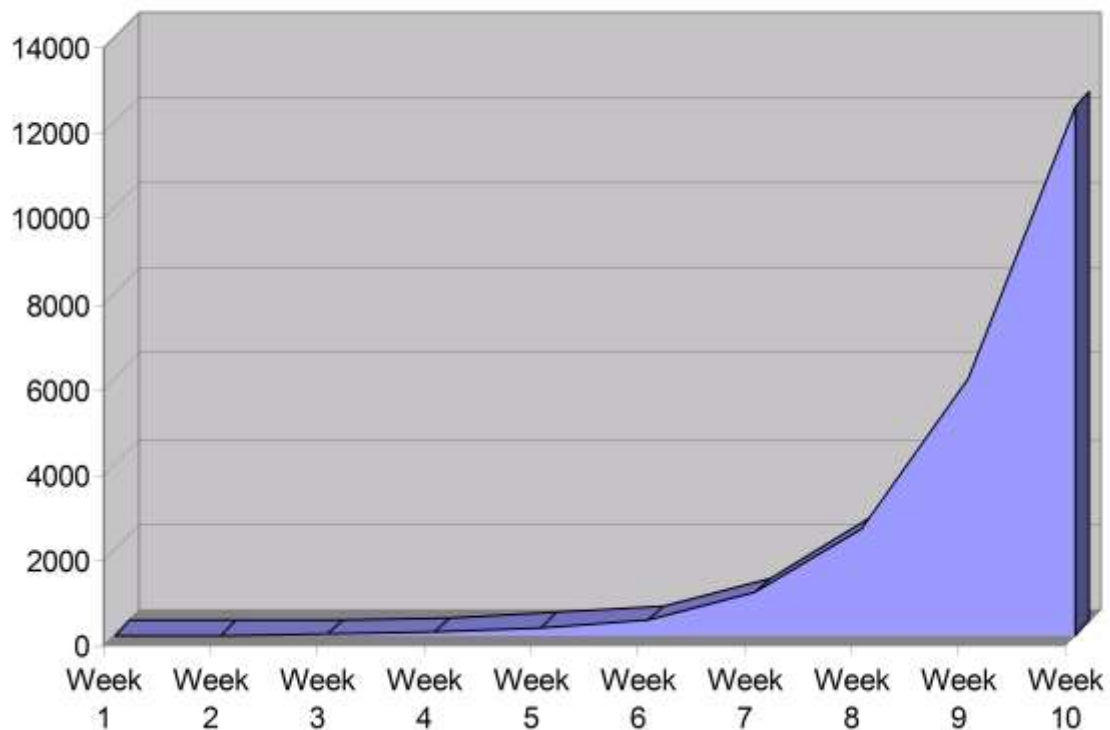
Scoped areas of execution, expected state generating models, and a model of application state. Events that loosely couple verification to execution. Constant baselining of application state. You may be finding it hard to believe that all this trouble can really be worth the effort. Oh, but it is! Loosely Coupled Comprehensive Verification isolates your verifications from the rest of your system, allows your verifications to start small and grow over time, and eliminates the follow-on failures that can be the curse of scripted test cases.



Loosely coupling Verification to the rest of the system isolates it off in its own corner. This separation makes Verification very flexible. The source of the events it receives is irrelevant, so a test case can call directly into verification just as easily as the LFM.

Similarly, the details of how each Expected State Generator performs its calculations have no bearing on the other Expected State Generators nor on the rest of the system, so these details can be changed at will – and as we all know, verification details change all the time.

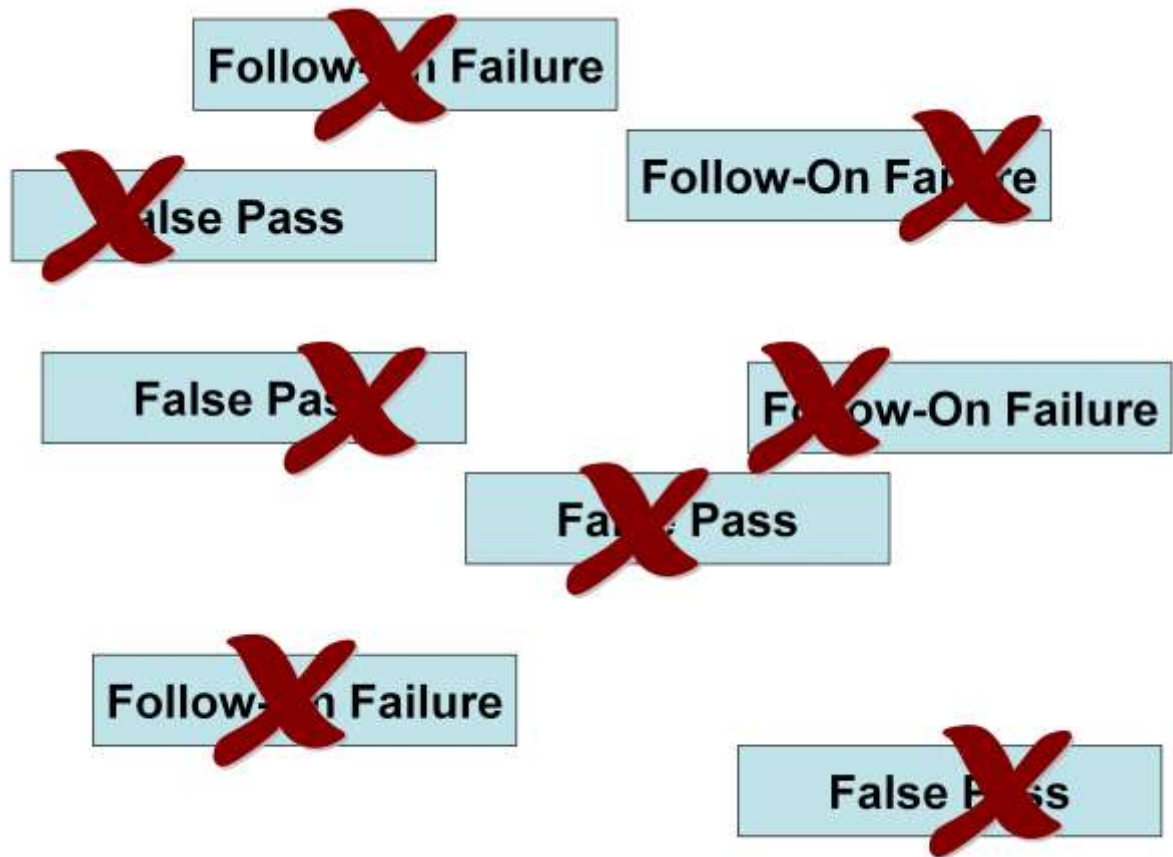
## Verified Datapoints



Decoupling verification from execution means the set of properties being verified can start small and simple. As application code comes online, as feature teams decide what should happen in particular cases, and as testers have time to implement the necessary calculations, these details can expand.

The ability to say "I don't care" what happens to a particular property as a result of a particular operation helps here as well. This allows decisions regarding what data to verify to be made early on whereas details regarding what is expected to happen can be decided over time.





One more benefit of Loosely Coupled Comprehensive Verification is the near elimination of follow-on failures. These occur when a test case hardcodes expected results; when a given step has an unexpected effect every verification of that datapoint fails. Since Comprehensive Verification always bases its calculations on the current application state, however, it automatically adapts to this situation and prevents follow-on failures.

Baselining also takes care of false passes – the evil cousin to follow-on failures. If a previous incorrect result is unexpectedly remedied by a subsequent step, Comprehensive Verification will note that something changed even though no Expected State Generator expected it to. By contrast, a standard scripted test case would miss this.

# Do you want to automate? Or test?



If you are writing scripted test cases the same way I used to, you are spending so much of your time writing and maintaining them that you have little time left over to actually do some testing. Is this really where you want to be? Do you want to be an automator? Or do you want to be a tester?

```
Logical.Projects.CreateNewProject();

Logical.SceneElements.CreateRectangle(
    new Point(100, 100), new Point(300, 300));

Logical.SceneElements.SelectAll();

Logical.Appearance.SetAppearanceProperty(
    Logical.Fill);
Logical.Appearance.SetColor(DataManager
    .FullSpectrumProvider.GetNextValue());

Logical.SceneElements.Cut();

Logical.SceneElements.Paste();
```

When the essence of your test case comes through,

- It says nothing about how its actions are carried out. Running it a number of times causes the full set of possibilities to be executed.
- It contains no verification and so will not need to change to make the verification more complete or if the expected results of any of its actions change.
- It contains no references to any UI and will not need to change regardless of how drastically the UI changes.
- It will only need to change if the functionality it is testing changes.
- It is very simple.
- It is focused on actions a user might take - it looks quite similar to the steps in a help topic, in fact.
- Writing it tests the spec.
- It can be written before the code it exercises exists.
- It can just "light up" once the code it exercises does exist.



These models provide a solid framework for your test cases to build upon. By focusing on user features in the Logical Functional Model, Application Internals Model, and Physical Object Model you produce a solid core infrastructure that will survive the ravages of time. Your test cases can focus on what they are doing rather than how they are doing it, and they can be understood at a glance. The patterns organizing your infrastructure are grounded in the stable base of customer features and so provide a consistent influence even as your application changes. Your test cases only need to change in reaction to major changes in your application's features.



Execution Behavior Manager and Data Manager are the track switches of your test libraries, directing decisions about which execution paths and test data values to use when. Moving path decisions to a centralized location allows you to optimize path and data coverage to cover those portions of your application you deem most important. As new execution paths or test data come online and existing ones become obsolete test cases pick up the changes automatically. Every part of every test case helps test the entire breadth and depth of your application and so you can get more testing out of fewer test cases. You can spend less time automating and more time testing.



Although these houses now form a seamless whole, they were constructed separately and continue to be modified piece by piece. Similarly, Loosely Coupled Comprehensive Verification lets you build execution and verification independently of each other. Implementation of your Logical Functional Model and Physical Object Model can proceed apace regardless of how much is unknown about how they will be verified. Individual Expected State Generators can come online, and existing ones become more robust, as their details become known. The elimination of follow-on failures simplifies test case failure analysis. False passes now can be caught. All of this occurs external to your test cases and so they become clearer and easier to understand.



These techniques do not just benefit your test cases. They also allow your feature definition, development, and testing process to be modified such that:

- Specifications can be tested more thoroughly, helping your feature definitions be more solid sooner.
- Test cases can be written as soon as (before, even) the specification is complete, providing a clear statement to your developers as to what the code they write should do.
- Test cases can be isolated from changes to the UI, giving your project managers and designers the freedom to explore different ideas through the development cycle and allowing the UI to be locked down much later than would otherwise be possible.
- Less time can be spent writing each test case, giving you time to write larger quantities of test cases as well as more detailed and more comprehensive test cases.

Your test cases can help your product win the race.

More details at

<http://www.thebraidytester.com>

