Automatically Add Variation Into Test Cases

Michael J. Hunter, Designer Tools Test Technical Lead

Removing Execution Path Decisions From The Test Case: Execution Behavior Manager

Problem: Multiple Possible Implementations Of An Action Often Don't Matter

Most user actions in an application can be executed in different ways. For example, creating a new document can typically be done via the following methods:

- Clicking the File menu, clicking the New submenu, then clicking the New Document menu item
- Typing Alt+F to invoke the File menu, typing N to invoke the New submenu, then typing N to invoke the New Document menu item
- Typing Alt to invoke the main menu repeatedly pressing the left arrow key until the File menu is selected, repeatedly pressing the down arrow key until the New submenu item is selected, pressing the left arrow key a single time to expand the New submenu, repeatedly pressing the down arrow key until the New Document menu item is selected, then pressing Enter to invoke the New Document menu item
- Invoking the New Document menu item via accessibility APIs
- Clicking the New Document toolbar button
- Invoking the New Document toolbar button via accessibility APIs
- Typing Ctl+N
- Executing the scripting object model method that creates a new document

An execution variant consists of the actions required to put the application into the starting state, gather initial state before execution, gather actual state after execution and compare it to the expected state, and to clean up after the test. The infrastructure required for each of these execution variants is typically identical. The entire test case for each variant, in fact, is completely identical except for the implementation details. Anyone who has written such cookie-cutter test cases has surely chafed at the necessity of doing so and wished for a better way.

Solution: Don't Make The Test Author Choose

Execution Behaviors are that better way. Execution Behaviors let test cases ignore execution variations they do not care about while still allowing test cases that do care about specific execution paths to choose those specific paths.

An Execution Behavior is nothing more than an LFM method. To use the example from the previous section, consider the Execution Behavior for Logical.ProjectItems.CreateNewScene (or CreateNewScene for short). CreateNewScene is a Composite Execution Behavior, which knows that it is implemented via one or more child Execution Behaviors.

Composite Execution Behaviors appear on the surface to be just another LFM method, but under their covers, they look very different. The first difference is that Composite Execution Behaviors

are decorated with an attribute that lists its child Execution Behaviors; this attribute is what makes a method a Composite Execution Behavior.

The second difference is that a Choose Any Composite Execution Behavior's implementation does exactly two things:

1. Ask the Execution Behavior Manager to select and return one of its child Execution Behaviors.
2. Execute the selected child Execution Behavior.

LFM methods that are child Execution Behaviors are not implemented any differently than LFM methods that are not child Execution Behaviors. The only visible evidence, in fact, is that a child Execution Behavior is tagged with various attributes indicating interesting properties about it: whether it uses the mouse or keyboard to manipulate UI, for example.

Composable

Any type of Execution Behavior can be composed into any other type of Execution Behavior, so a Choose Any Composite Execution Behavior may itself be a child to one or more other Composite Execution Behavior. Thus invoking a Choose Any Composite Execution Behavior may cause a variety of other Execution Behaviors to be invoked as execution travels down the tree of children. This allows a complex tree to be easily built.

Customizable

Execution Behaviors help isolate test cases from the details of executing actions. However, individual test cases may at times require a specific execution method. For example, a bug may only reproduce when an operation is executed via the mouse. The LFM exposes both Composite Execution Behaviors and child Execution Behaviors as first class citizens, allowing both types to be directly accessed and executed.

Replayable

When an automated test finds a bug, a regression test for the bug that reproduces exactly the steps that were taken (i.e., child Execution Behaviors that were executed) in that instance is needed. As the test case executes, the Execution Behavior Manager records each choice it makes. When the test case completes this data is folded into a copy of the original test suite; running this suite causes the Execution Behavior Manager to make the same choices in the same order as it did previously.

Replay suites may become invalid for several reasons, especially due to changes to the test case or any of the Execution Behaviors it executes. When the Execution Behavior Manager runs out of items to replay, it simply falls back to its standard selection process. While replayability could be made robust against test case changes, doing so would be complicated and quite hard to make robust. For now, the work required has been judged to not be worth the potential gains.

Comprehensive

Any particular test run may not execute a Composite Execution Behavior sufficient numbers of times to cause each of its child Execution Behaviors to be selected. Over the course of many months, however, each child Execution Behavior will be selected many times over. The Execution Behavior Manager ensures this by remembering the details of which child Execution Behaviors it selected during previous runs and using that information as the basis for its decisions during subsequent runs. This historical data can also be fully or partially ignored if the person creating the test run so desires.

Biasable

Other factors also affect the process of selecting an Execution Behavior. Each Execution Behavior can declare metadata about itself, such as whether it uses the mouse or keyboard. Global and local weighting factors can then be used to skew the Execution Behavior selection process in specific directions. For example, if the menu system has just been completely redesigned, a global weighting factor could be set such that only Execution Behaviors that use the menus are selected. Local weighting factors allow similar control per test case or child Execution Behavior.

Disableable

Weighting factors allow the selection process to be adjusted on a more-or-less macro scale, but this is not always sufficient. An individual Execution Behavior may only be executable when the application is in a specific state. There is little point in selecting an Execution Behavior that can't actually be executed, so the Execution Behavior Manager must be able to ignore disabled Execution Behaviors when making its selection. Child Execution Behaviors can specify a method for the Execution Behavior Manager to use to determine whether the Execution Behavior can be executed at the current time. This method can use whatever means it likes to make this determination, save it cannot change application state in any noticeable way. If no such method is specified, the Execution Behavior is assumed to always be available.

Removing Test Data Generation From The Test Case: Data Manager

Systematic variability pervades this system. Test cases simply call methods in the Logical Functional Model. Any particular LFM method may in fact be a Composite Execution Behavior with many child Execution Behaviors, one of which will be selected according to some scheme. LFM methods and test cases using a Physical Object Model get variability across control manipulation mechanisms. However, one potential source of variability is still missing: data.

Problem: Test Cases Need Test Data

While the actions a test case takes are of utmost importance, the data used to take those actions are very important as well. "Create a file" is not a very good test case. What kind of file? Where? What should it be named? How big should it be? What kind of data should it contain? A good test case is specific: "Create a text file named 'HonestAbe.speech' containing the Gettysburg Address in c:\StuffToMemorize" for example. However, it probably sits right next to "Create a text file named 'HonestAbe.speech' containing the Gettysburg Address in \\TestServer\Junk\StuffToMemorize" and "Create a Microsoft Word file named 'HonestAbe.speech' containing the Gettysburg Address in c:\StuffToMemorize" and a plethora of similar test cases. All of these are nearly identical, differing only in some small detail. As mentioned previously, creating and maintaining large numbers of test cases is tedious, error-prone, and a maintenance nightmare.

An obvious solution to this problem is to write a single test that is parameterized appropriately. To continue with the example, a helper method might create a certain type of file in a certain location, give it a certain name, and fill it with certain text. Test cases would then simply call this helper method passing the various data as arguments. Doing this is nothing new.

Problem: Test Case Data Isn't Shared

The problem with doing this is that the data is locked up in all those individual test cases. This makes it hard to understand what data is being used, and it makes it impossible to reuse this data elsewhere (in other tests' setup and teardown, say). (Except, of course, for the ever-present

option of cutting-and-pasting the data, with all its concomitant problems.) Understanding the data would be simpler if it was all collected it in a single location. Rather than creating many small test cases each responsible for a particular piece of data, a single test case that cycles through the complete set of data could be created.

Although this approach is often used, the data is still locked up in the test case and so reusing the data is still inconvenient. That could be fixed by moving the data out of the test case into some sort of data repository. Once the test case has loaded the data from the repository it can cycle through it just like it used to, and other test cases can now cycle through the data as well. The problem now has become that each test case must know how to find the data repository and use it to load the data. This problem can be solved by encapsulating the whole process in a helper method each test case uses. Doing this is nothing new.

Using a helper method to load data from a repository works just fine: any test case can easily run through the full set of data. However, running through the full set of data (i.e., focusing the test case on the data) is not the most common scenario. The most common scenario is wanting to introduce variability into test cases' setup and cleanup methods. The helper method could be used to do this, but getting good coverage over the full set of data would be difficult. Either the first item would always be used, or each caller would pick a random item to use, or a global "use this item next" counter would be necessary. The first doesn't give any variability, the second gives uncontrollable variability, and the third is likely to have test cases stepping on each other's toes.

The second and third options are quite close to what is wanted, however: a magic method that generates a value of a particular type of data. If this magic method could also generate a sequence of values of a particular type of data, it could be used in run-through-the-full-set-of-data scenarios as well.

Solution: Test Data Is Managed Similarly To Execution Behaviors

Data Manager does just this. Just as Execution Behavior Manager manages the selection of Execution Behaviors, Data Manager manages the selection of data. Data Manager is populated with a set of Data Providers. Each Data Provider is responsible for generating a sequence of a particular type of data: a closed set of test strings. Randomly generated strings of random lengths. Legal filenames. Illegal filenames. Clickable points for a UI widget. Each Data Provider has a unique and well-known identifier that test cases and LFM methods reference to request a particular type of data from Data Manager. Data Manager clients must also specify whether they wish a single value (i.e., the common case of needing data during test case setup or cleanup) or the full set of values (the case mentioned earlier where those values will be used to motivate a test case).

As might be expected, Data Manager has many of the same needs as Execution Behavior Manager. Replayability is necessary for regression tests and reproducing test run failures. Spreading data values around test cases requires Data Manager to remember the values it used in previous test case runs. In fact, in many ways, Execution Behavior Manager is nothing more than a highly specialized Data Provider.

The service that Data Manager provides might seem trivial. After all, it just sits there handing out data. The value gained from it, however, is decidedly non-trivial. Each context-specific set of test data is located in a single location rather than spread throughout the test cases. Test cases can consume Data Manager-generated data as easily as creating it themselves. Setup and teardown code can use this data, allowing the entire test suite to be used to get coverage over these test data rather than requiring specific test cases that explicitly hit the conditions.

Automatic Variation Over Execution And Data

The power these techniques present should be clear. A test case typically must choose from many different execution variants and data values, but true variety in choices is difficult to achieve. Moving such decisions out of the test cases and into the automation framework simplifies the test cases at the same time as it lets more of each test case test more of the application under test. Greater variability over greater area can't help but equal greater test coverage. Greater test coverage should in turn result in more bugs found earlier and less bugs sneaking through to the customer. All of which make for a better application and happier customers.

Acknowledgements

Many thanks to everyone who reviewed this paper, most especially Mike Gallacher. Thanks also to the entire Designer Tools team for assisting our efforts to make testing more efficient and productive.

Appendices

Appendix A: Composite Execution Behavior Variants

The Composite Execution Behaviors discussed in this paper are "Choose Any" Composite Execution Behaviors, where each of the child Execution Behaviors implements the relevant functionality entirely within itself, and so any one may be chosen. Other types of Composite Execution Behaviors can be imagined:

- An "Execute In Any Sequence" Composite Execution Behavior, where each of the child Execution Behaviors is to be executed but the order in which said execution occurs is of no importance.
- An "Execute In Sequence" Composite Execution Behavior, where each of the child Execution Behaviors is to be executed in a specific sequence.
- An "Execute Until" Composite Execution Behavior, where each of the child Execution Behaviors is to be executed in a specific sequence, repeating the entire sequence until a condition is met.

"Execute In Sequence" and "Execute Until" Composite Execution Behaviors turn out to be better implemented in code. A scenario requiring an "Execute In Any Sequence" Composite Execution Behavior has not yet been found, so those remain mostly theoretical for now as well.

Appendix B: Adopting Individual Pieces

Execution Behavior Manager

The Execution Behavior Manager itself is reusable as is but for a few configuration details isolated in configuration files, such as the data repository to use.

Both Composite and child Execution Behaviors can live in any class organization and hierarchy you like. The only restriction is that the children of any particular Composite Execution Behavior must all have parameters identical to each other as well as identical to or at least derivable from those of the parent Execution Behavior. Other than this, adding Execution Behaviors into your system requires a few steps:

- Define a delegate for each composite Execution Behavior you will be defining.

- Create Composite Execution Behaviors for each set of related child Execution Behaviors. A Composite Execution Behavior is simply a method that is tagged with a special attribute and whose implementation asks Execution Behavior Manager to choose one of its children, which choice the method then executes.
- If you plan on using weighting factors, adorn your child Execution Behaviors with the appropriate category attributes.

Data Manager

The Data Manager itself is reusable as is but for a few configuration details isolated in configuration files, such as the data repository to use.

Beyond that, Data Manager is simply a particular method for generating variable data and sharing the history of those choices across test cases. The only change necessary to your existing test case automation process in order to make use of Data Manager is to retrieve test data from a (perhaps built-by-you) Data Provider rather than retrieving it from some other datastore or hard-coding it.