Record and playback at the GUI level can often create more problems than it solves. Before you abandon this method altogether, though, consider deploying it within the application itself.

LOGICAL CAPTURE/REPLAY

BY MICHAEL SILVERSTEIN

INFO TO GO

- Capture/replay interactions with an application at the business and control logic level are semantically meaningful.
- Logic-level capture/replay can be used to implement a number of different automation approaches.

Capture/replay is a much maligned but potentially useful means of accelerating the development of automated functional tests. In capture/replay, the test developer interacts with an application under test, typically through the graphical user interface (GUI), while some capture tool simultaneously generates an automated test script that is configured to play back an equivalent sequence of interactions. This is supposed to save time that would have otherwise been spent hand-coding test scripts. The problem is that GUI-based tests tend to be fragile, and the recorded interactions tend to contain a lot of content, but little meaning about the intent of the interactions. GUI capture/replay tools know only how to interact with gesture widgets. They have no idea what sort of application they are operating on or what domain terminology the application under test uses.

Instead, wouldn't it be great if test scripts were written in terms of what they were trying to accomplish? When you make an airline reservation, you don't tell the booking agent which buttons to click and which fields to enter text into. You just tell him where you'd like to go and when. Shouldn't test scripts have the same clarity? It would certainly make them a lot less fragile, and much easier to understand.

Perhaps a better place to capture and replay interactions is within the application itself, at the business and control logic level. This frees test developers from the constraints that GUI-centric automation places on them and opens up the possibility of creating scripts that are couched in terms of the application's business terminology.

GUI Interactions

Consider the "logon" shown in Figure 1.

What if we want to enter an invalid password and verify that the application responds with the text of "Invalid name or password"? A typical GUI capture/replay tool might generate a script that looks something like this:

```
set_focus_to_window("Log on");
enter_text("field1", "Mike");
enter_text("field2","invalidpassword");
click_button("button1");
set_focus_to_window("Error!");
verify(get_field("field1"),"Invalid name or password");
click_button("button1");
```

There are (at least) four problems with this:

1. If the structure of the view were to change substantially, the script would break.

2. It is difficult to determine the intent of the generated script just by reading it, partly because the script ends up performing four discrete steps in order to initiate the single operation of logging in a user, and three steps to verify and dismiss an error dialog.

3. The control names (such as "field1" and "button1") are not descriptive, so the script is hard to read. Many GUI construction tools automatically name controls. This example is actually more readable than it might normally be because the text fields are named in the order you would fill them in. They aren't always in that order.

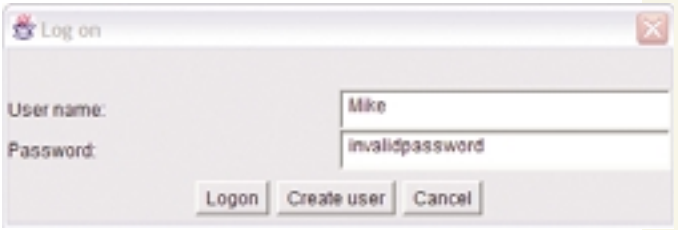


Figure 1: A simple user logon dialog

4. The fact that "field1" and "button1" appear in both the "Log on" window and the error dialog further obfuscates the intent of the script. Even if this were not the case, we still have references to controls in two different windows to sift through, which can be confusing to someone trying to make sense of the script. This is especially true for more complex windows.

Tests that interact at the GUI level possess a high degree of fragility and complexity. In addition, each line of a script contributes only a small part to the whole. Combined, these factors make GUI-centric scripts difficult to comprehend and increase the likelihood that they will degrade over time, either through changes in the views or through errors caused by changes to the scripts themselves.

Logic-Level Interactions

So, how do we fix this? The procedure for implementing capture/replay within the application itself, bypassing the GUI entirely, is relatively simple. The core idea is to capture interactions in a form that is concise and meaningful, at locations within the application where that meaning is known. To do this, the application developer(s) will build in simple calls to a recorder utility within the code wherever an operation that might need to be captured for

Logic-Level Capture/Replay Construction

1/ The application developer inserts calls to a recorder utility anywhere in the code where meaningful operations are initiated, passing whatever parameters are necessary to recreate the interaction at playback.

2/ The recorder utility generates automated tests in terms of the application's domain and in a desired format.

3/ The test developer interacts with the application in order to record tests, just as he would with a GUI capture tool. The only difference is that the generated scripts are much more readable.

a test is initiated. The test developer uses the same process of interacting with the application under test to capture recordings of the interactions that he would normally use with a traditional GUI capture tool. The main difference is that the recorder captures the essence of operations performed by the application, as well as the result of their execution, instead of interactions with GUI controls. The tests that are automatically captured are concise, meaningful, and easy to maintain, instead of verbose descriptions of interactions with individual controls.

This scheme asks application developers to add individual statements within the application code to call a recorder and to configure the recorder to output meaningful test scripts. These calls to a recorder are what provide the domain knowledge, terminology, and context that are missing in GUI capture. There are many ways that application developers can structure code to make this scheme flow naturally with minimal effort. This particular approach to structuring also has the side benefit of helping with maintainability and good design. As an added bonus, it accidentally adds useful features such as a framework for undo/redo and a debug log.

Operations, Artifacts, and Operation-Centric Artifacts

An **operation** is considered a discrete service or transaction provided by an application. An end user usually initiates an operation through an application via a GUI. Examples of operations are:

- Log on to the application with user name *xxx* and password *yyy*.
- Create a new customer named *xxx* at address *yyy*.
- Query all flights between *aaa* and *bbb*, between times *ccc* and *ddd*, on month *mmm*, day *dd*, with a maximum of *ss* stops.
- Calculate a loan payment schedule given principal *ppp*, rate *rrr*, term *ttt*, and payments per year *yyy*.

Examples of things that are not considered operations are:

- Enter text in name field.
- Select customer name in a list (unless that effects a change in the state of the application).

Instead, these are just fragments of the setup required prior to initiating an operation.

An **artifact** is simply the product of a capture session. In GUI capture/replay, an artifact is a statement in a tool's scripting language. That statement, when executed by the tool, acts on a GUI control.

An **operation-centric artifact** could be a sentence such as "Log on to the application with user name *xxx* and password *yyy*." That's meaningful to anyone reading it, but a tool can't replay it. So, instead of a human language, we use a programming language:

logon(String name, String password)

Even though it's not English, the process is clear.

Operation-centric tests tend to be much more compact than GUI-centric tests. Notice that the GUI script for logging on to the application shown earlier is four times longer than the operation-centric representation, not counting validation.

Another popular form for an artifact is one or more rows in a table of action words with parameters. (For more on this concept, see the Buwalda and Cunningham texts cited in this issue's StickyNotes at www.stqemagazine.com.) During playback, the rows of the table are read, and the action words and parameters are used to make calls to application APIs.

The Recorder Utility

GUI capture/replay tools operate outside the application. Logic-level capture/replay tools require that code be added to the application. One of the first things you might be thinking is that you don't want a bunch of test code mixed in with the application, and this is certainly a valid concern. The logic for generating artifacts should be isolated into a *recorder utility*. All that should be implemented within your application are statements that call the recorder.

The recorder utility has the responsibility for starting recording, stopping recording, and generating an artifact that can later be used to replay the request and check that the result is the same as when the artifact was recorded. This process may seem difficult, but it is actually very simple to do. (An example recorder is provided in this issue's StickyNotes.)

What follows is a sample call that the application can make to a recorder:

Recorder.addReplayAndValidation(*operation-description, expected-result*)

Reading from left to right, there are four parts:

- The recorder itself
- A message to the recorder that tells it what to do
- A description of the operation the user just initiated
- The result of the operation, which the recorder uses to generate validation

From this information, the recorder generates an artifact. Since the program is in Java, it might make sense for the artifact to be a Java statement in a test script. Here's an artifact that replays a logon operation:

```
BusinessSystem.logon("Mike", "please");
```

When the script is replayed, this artifact causes the test tool to replay the logon operation. BusinessSystem is the *test automation interface* between the tests and the application under test.

The recorder can also add two types of validation:

- Validate that the value returned by the operation is correct
- Validate that the application under test is in the expected state

page 39
full-page ad
SQE Training

Suppose logon returns a value to indicate success or failure. It is simple to capture that value and have the script validate that replaying the operation produces the same value. Here's an example:

```
TestTool.assertEqual(BusinessSystem.logon("Mike", "please"), true);
```

After the business system logs "Mike" on, the tool checks that the return value of logon is true.

When you have a recorder that has knowledge of the operation being recorded, instead of GUI interactions, the recorder can also know what the state of the application should be after the operation has completed. In those cases, you can add code to the recorder that generates artifacts to validate the state. Here's an example that would follow the previous validation statement:

```
TestTool.assertEqual(BusinessSystem.isLoggedOn("Mike", "please"),
true);
```

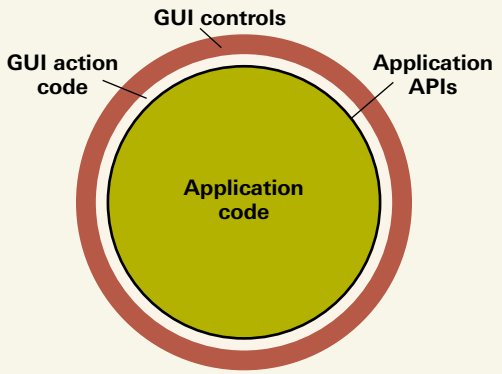
This example uses the test automation API to determine if a particular user with a particular password is logged on. The result is compared to a captured reference value.

Model-View Separation

Well-structured applications make a clear distinction between the external interface (view) and the control and logic layers (model). This is called model-view separation. The core idea is to have two distinct parts:

- The **model:** code that provides a callable interface for the various operations provided by the application
- The **view:** code that simply presents a GUI on the screen and delegates action requests to the model but has no real "intelligence" of its own. This is the interface between the model and the end user.

The GUI is simply a lightly coupled front-end to the application code. In a well-structured application, the GUI code handles user actions by simply delegating requests to the underlying application code. The application code then performs operations on behalf of the GUI. The GUI displays returned values and other indications of the state of the application on the screen. Operational requests from the GUI to the application generally pass along values filled in by the user in fields and other data-oriented GUI controls.



Structured this way, the application can provide its services without need of a view, and your automated test scripts can call the application model code directly via APIs (or through some intermediary layer) without regard to the presence of GUI controls.

Structure and Patterns

The logic-level capture/replay scheme is lightweight enough that you need to change your application only slightly to enable it. You need only a simple recorder that can be called to generate replay statements, and the calls to the recorder within the application itself.

Yet, as I implemented this scheme for the first time, I noticed that certain ways of structuring an application, combined with several basic design patterns, made the implementation of the capture mechanism flow naturally and contributed to a very strong implementation of the application itself. Enabling the application for the scheme made for a better implementation and incurred almost no development overhead at all.

Some design patterns that work well include model-view separation, singleton patterns, command patterns, and façade patterns. Model-view separation is a cornerstone of good design, and using it in this context pays great dividends. (See the accompanying sidebar for details.)

A singleton ensures that a class has a single instance and provides a known, global point of access. If you need to provide global or static methods, consider using a singleton. We'll talk more about singletons later in the article.

The command pattern encapsulates an operation and the information (parameters) required to execute it within one class. Typically, you implement one command class for each type of operation that your application performs. If you use the command pattern, you may well end up with a hierarchy of command classes, often rooted in an abstract class that provides common behavior. Command classes also provide a nice place to include other ancillary services, such as logging, undo/redo, and in this case, calls to a recorder.

The façade pattern provides a unified interface for all or part of an application or subsystem, and makes the interfaces of multiple classes accessible via one class. Façade methods usually do little more than delegate requests to the classes that they conceal.

We can create a façade class that provides API methods for executing operations, one for each operation. Each API method should know how to create, configure, and execute a respective command object. We can make the façade class available as a singleton to both the view and the tests. When a command class executes, it can easily locate the recorder because of its singleton interface and call it to capture an operation.

Design patterns are commonly accepted programming solutions to regularly encountered problems. Further reading on patterns is provided in this issue's StickyNotes.

Singletons and Recorders

The most prominent advantage of presenting recorder APIs via singleton instance methods, instead of via global static meth-

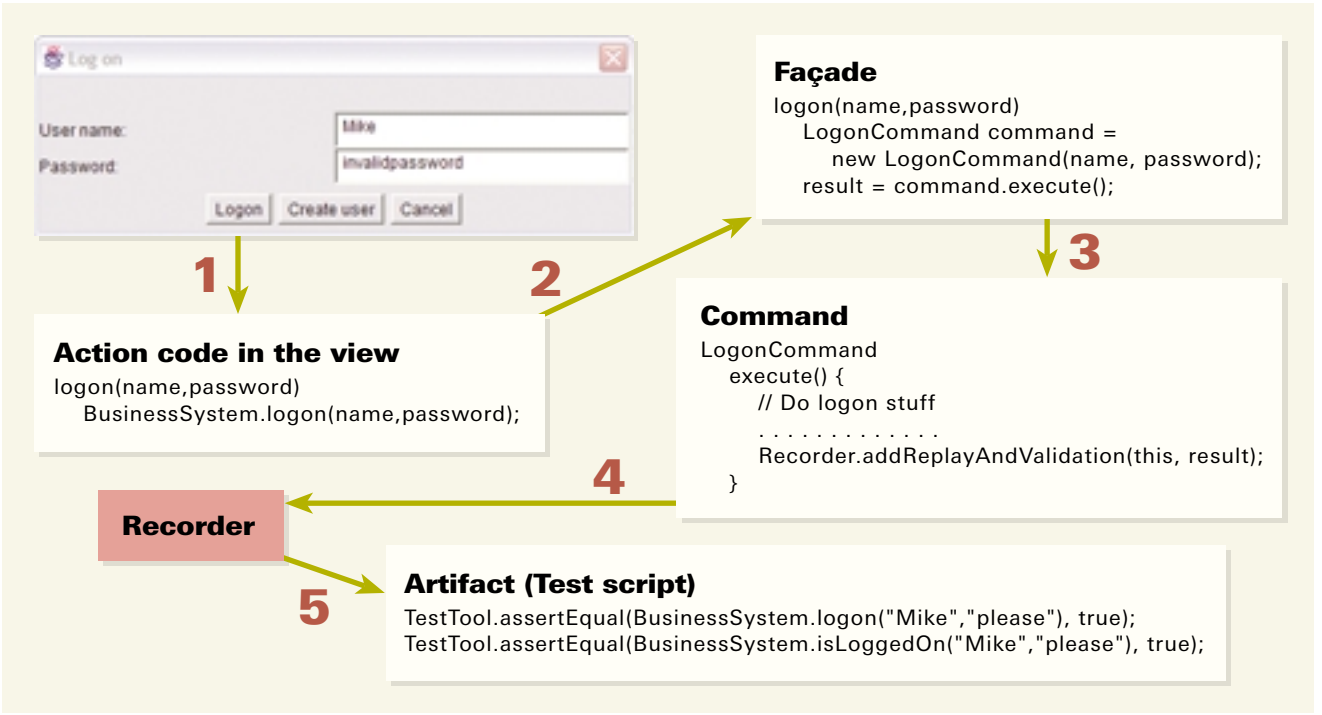


Figure 2: The basic pieces for creating a simple test-script-based capture artifact

ods, is that with a singleton we can easily swap-in different recorder implementations to generate different types of recorded artifacts. For example, we can have two recorders: one that generates scripts made up of Java statements, and one that generates action-word spreadsheets. All we have to do to change the generation scheme is to alter which recorder instance is returned by a singleton accessor method.

Singletons and Automation APIs

We can also use the singleton pattern when implementing a single point of access for initiating operations. In this case, the most important benefit is that the APIs for initiating operations are easily accessible from both the GUI code and the tests themselves.

The diagram in Figure 2 shows the basic pieces for creating a simple test-script-based capture artifact. In Figure 2, the user fills in the **name** and **password** fields in a dialog and clicks the **Logon** button. This causes the following to happen:

1. The view action code for the button click is called.

2. The view action code should do no more than delegate the request to the business logic via a façade-class API for the logon operation. Remember: We want clean model-view separation.

3. The façade logon method creates an instance of a logon command and tells it to execute.

4. The logon command code does whatever it needs to in order to honor its operation, then calls the recorder addReplayAndValidation(LogonCommand, String) method, passing it the command object and the result of executing it.

5. In this example, the recorder generates a test-script-based artifact as a statement that will call the logon API in the façade with the same arguments. The script also includes a call to a helper method (in this example, named assertEqual) in a test framework class (TestTool) that will confirm that the results of the operation are the same when the test script is executed as they were when the operation was captured, or log the failure. (Discussion of the actual test tool execution framework is beyond the scope of this article.)

Logic-level capture replay is not just for client-side applications written with a GUI. You can also use it to automate server-side tests. The key is to use the same architecture described above, but instead of calling the façade from the view, simply call it from whatever component receives client requests.

For instance, you could create a J2EE servlet to delegate requests from a browser to a façade. The rest of the code would be the same as that described above. During replay, you would simply execute the test on the server itself. You might want to implement a remote test execution interface that enables you to execute tests from a client machine.

Test Development Process

The process for using logic-level capture/replay is not much different from what test developers are currently using with GUI capture/replay tools. They will still use the application under test in order to capture artifacts that can be used by a tool to replay interactions. The key difference in this scheme is that the test developers may end up working with the application developers to identify operations that they would like to automate. This will guide the application developers as to where they need to add calls to the recorder. Test developers will also ask application developers which parts of the state of the application they want validated. The application developers will provide APIs and structure to support that testing.

Note that testability and good design support each other. In

Tips & Cautions

TIPS

- Have a clear separation between the model and the view, which places little or no logic in the view itself. If you are automating tests for a server, keep the code that receives client requests simple, delegating as much as possible to other downstream classes.
- Identify each discrete operation provided by the application under test, and create simple interfaces for those operations that make no assumptions about how they are being called.
- Arguments for operation API methods, as well as returned values, should be primitive values such as String, int, float, etc. These are much easier to capture and pass within generated playback artifacts. If you need to pass a complex object to an API method, it is best to do one of two things: (1) Pass the values required to construct the object to the API and let the API method deal with assembling it, or (2) If the object is already held in a common area, pass a simple identifier for the object to the API and let the API retrieve the object by that identifier.
- If the artifacts are script statements, it is a good idea to generate comments to go along with them. These can include descriptions of the operations performed, as well as parameters passed.
- You will need to perform some simple testing of the connection between the view and the model, through either traditional GUI testing or manual testing. Remember that the purpose of testing at this level is not to test that the operations are performing correctly, but to test that the connection between the view and the model is operating correctly.

CAUTIONS

- Too little separation between the model and the view may make it difficult to implement this approach, especially if there are no clearly defined, view-independent APIs for initiating operations.
- This approach assumes that the logic within the view itself is trivial, and that the risks inherent in not exercising the code paths within the view are minimal. If you have included a lot of logic within the view, this approach could potentially cause you to miss testing that logic.
- As always, test development should follow from some set of intentionally designed test case scenarios. Creating tests by haphazardly recording interactions will almost always fail, regardless of the form the recorded interactions take or the level at which playback occurs.

general, well-designed code by its nature will be more testable than poorly designed code. Likewise, designing with testability in mind generally results in well-structured, more maintainable code, so testability does not necessarily mean "extra work" for the application developer.

Debugging

Because the generated artifacts are concise and meaningful, one of the side benefits to developers is that the recorder can be used as a basis for a logging mechanism. Having a log of a sequence of operations, the parameters passed to them, and the returned value and state of the application under test can be a powerful tool for locating the cause of a defect, without even opening a debugger. The best part is that once the capture mechanism is in place, it is free.

Exploratory Testing

Exploratory testing is the act of manually interacting with the application under test for the purpose of testing, while simultaneously designing the path that those interactions take. James Bach describes it as "any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests." (For more on Bach's view of exploratory testing, see this issue's StickyNotes and James's new article on page 16 of this issue.)

One of the difficulties of exploratory testing is discovering a defect as a result of exploration and not remembering exactly what steps you took to get there. Logging tools can help. Because the capture mechanism described here can create a log of a sequence of operations and the parameters passed to them, it is very easy for exploratory testers to see what path they took. It is also conceivable that an exploratory tester might use or pass this log on to a test developer in order to create automated tests based on some or all of the paths taken.

Conclusion

If you've got a reasonably well structured system implementation, it is very easy to add in a mechanism to capture interactions with operations that system provides and to generate playback artifacts that are meaningful. Performing operation-centric capture/replay avoids many of the pitfalls of traditional GUI-centric capture/replay. STQE

Michael Silverstein is CTO and a co-founder of SilverMark, Inc., a leading provider of automated testing tools, training, and mentoring. He has been developing and shipping working software since 1981. Contact Mike at msilverstein@silvermark.com. Read more about SilverMark at www.silvermark.com.

STICKYNOTES

For more on the following topics go to www.stqemagazine.com

- Action words
- Patterns
- Exploratory testing
- Sample recorder
- Implementation details

page 43
full-page ad
Powerpass