DOI:10.1145/2755501

Article development led by acmqueue
queue.acm.org

Use states to drive your tests.

BY ARIE VAN DEURSEN

Testing Web Applications with State Objects

END-TO-END TESTING OF Web applications typically involves tricky interactions with Web pages by means of a framework such as Selenium WebDriver.[12] The recommended method for hiding such Web-page intricacies is to use page objects,[10] but there are questions to answer first: Which page objects should you create when testing Web applications? What actions should you include in a page object? Which test scenarios should you specify, given your page objects?

While working with page objects during the past few months to test an AngularJS (https://angularjs.org) Web application, I answered these questions by moving page objects to the *state* level. Viewing the Web application as a state chart made it much easier to design test scenarios and corresponding page objects. This article describes the approach that gradually emerged: essentially a state-based generalization of page objects, referred to here as *state objects*.

WebDriver is a state-of-the-art tool widely used for testing Web applications. It provides an API to access elements on a Web page as they are rendered in a browser. The elements can be inspected, accessing text contained in tables or the element's styling attributes, for example. Furthermore, the API can be used to interact with the page—for example, to click on links or buttons or to enter text in input forms. Thus, WebDriver can be used to script click scenarios through a Web application, resulting in an end-to-end test suite of an application.

WebDriver can be used to test against your browser of choice (Internet Explorer, Firefox, Chrome, among others). It comes with different language bindings, allowing you to program scenarios in C#, Java, JavaScript, or Python.

Page Objects. WebDriver provides an API to interact with elements on a Web page as rendered in a browser. Meaningful end-user scenarios, however, should not be expressed in terms of Web elements, but in terms of the *application domain*.

Therefore, a recommend pattern for writing WebDriver tests is to use page objects. Such objects provide an API over domain concepts implemented on top of Web-page elements, as illustrated in Figure 1, taken from an illustration by software designer Martin Fowler.[4]

These page objects hide the specific element locators used (for example, to find buttons or links) and the details of the underlying "widgetry." As a result, the scenarios are more readable and easier to maintain if page details change.

Page objects need not represent a full page; they can also cover part of the user interface such as a navigation pane or an upload button.

To represent navigation through the application, "methods on the PageObject should return other PageOb-

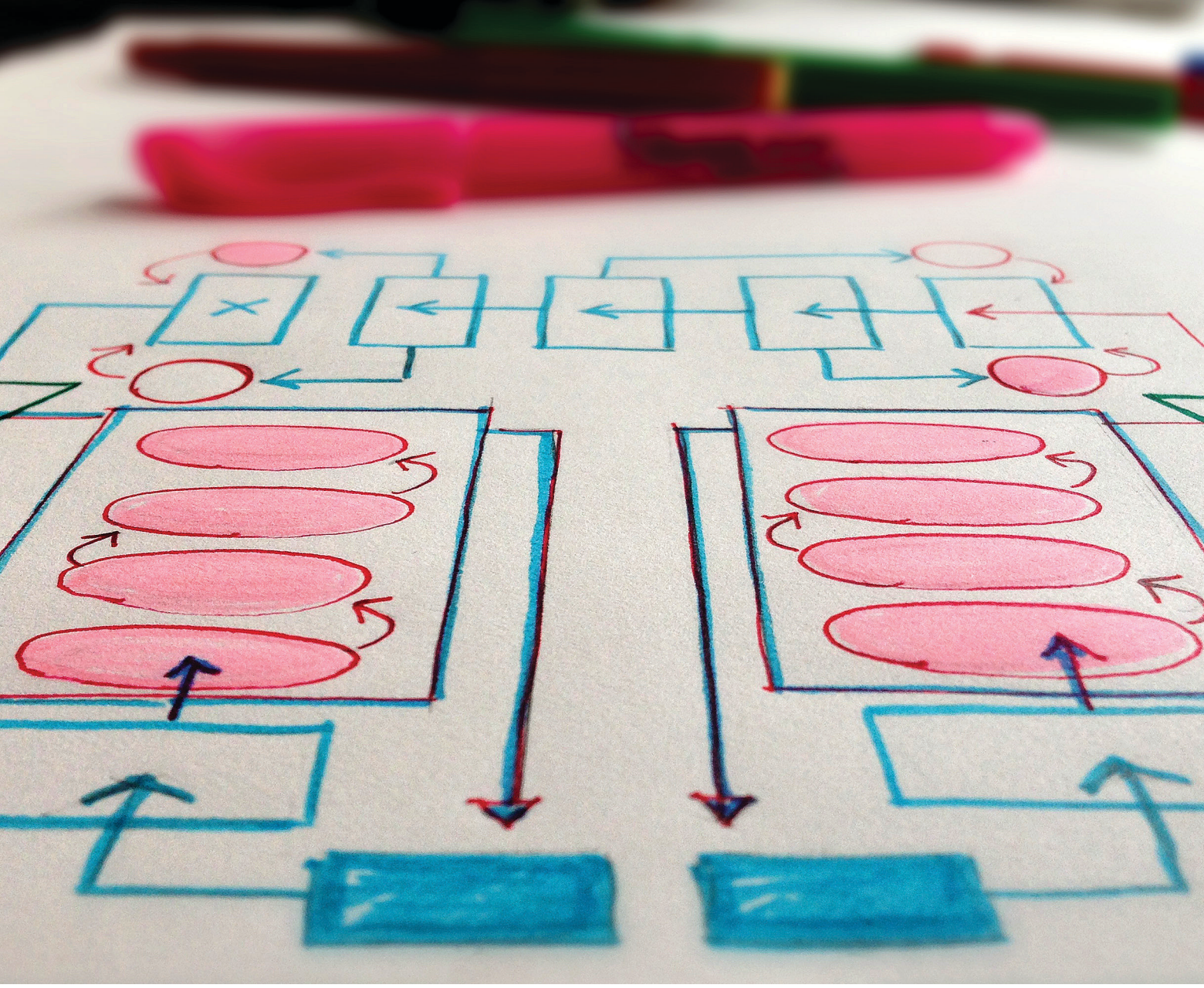


IMAGE BY ALICIA KUBISTA/ANDRIJ BORYS ASSOCIATES

jects."[10] It is this idea that this article takes a step further, leading to what we will refer to as *state objects*.

Modeling Web Apps with State Charts

To model Web application navigation, let's use statecharts from the Unified Modeling Language (UML). Figure 2 shows a statechart used for logging into an application. Users are either authenticating or authenticated. They start not being authenticated, enter their credentials, and if they are OK, they reach the authenticated state. From there, they can log off to return to the page where they can authenticate.

This diagram traditionally leads to two page objects:

- One for the login page, corresponding to the authenticating state.
- One for the logoff button, present on any page shown in the authenticated state.

To emphasize these page objects represent states, they are given explicit responsibilities for state navigation and state inspection, and they become state objects.

State Objects: Checks and Trigger Methods

Two types of methods can be identified for each state object:

- *Inspection methods* return the value of key elements displayed in the browser when it is in the given state, such as a user name, the name of a document, or some metric value; They can be used in test scenarios to verify the browser displays the expected values.
- *Trigger methods* correspond to an imitated user click and bring the browser to a new state. In the authenticating state users can enter credentials and click the submit button, which, assuming the credentials are correct, leads the browser to the next authenticated state. From there, the user can click the logoff button to get back to the authenticating state.

It is useful to combine the most important inspection methods into one self-check of properties that must hold whenever the application is in that particular state. For example, on the authenticating state, you would expect fields for entering a user name or a password; there should be a submit button; and perhaps the URL should include the login route. Such a self-check method can then be used to verify the browser is indeed in a given state.

Scenarios: Triggering and checking events. Given a set of state objects,

Figure 1. Page objects. (Figure by Martin Fowler)

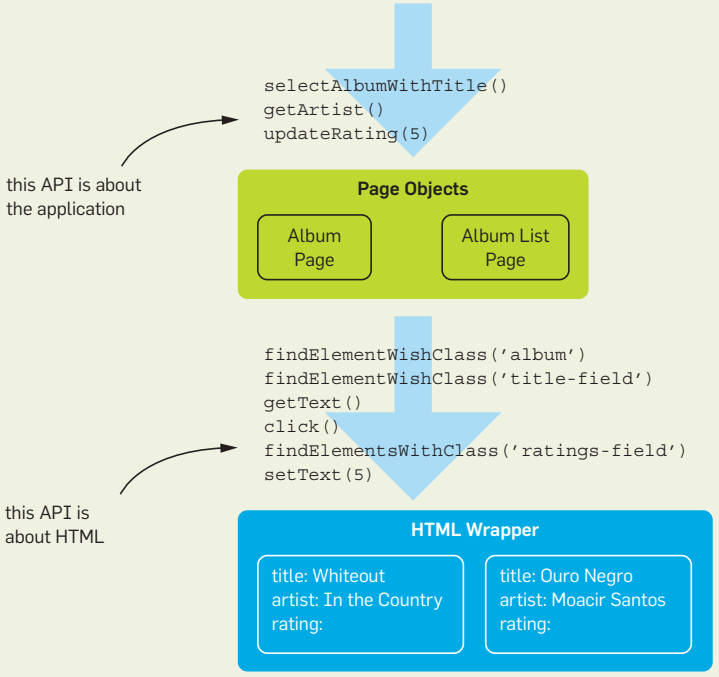


Figure 2. Logging into an application.

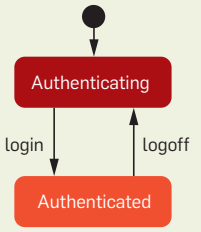


Figure 3. Login with conditional events.

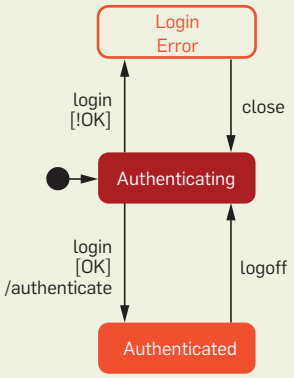
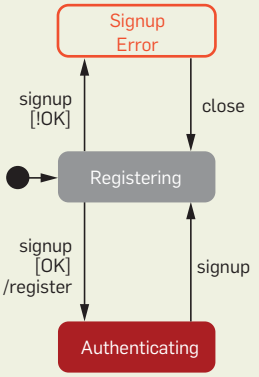


Figure 4. Registering a new user.



test cases describe relevant *scenarios* (paths) through the state machine. For example:

1. Go to the login URL.
2. Verify you are in the Authenticating state via self-check.
3. Enter correct credentials and submit.
4. Verify you are in the Authenticated state.
5. Hit logoff.
6. Verify you are in the Authenticating state.

Thus, a scenario (test or acceptance) is a sequence of actions, each followed by a check the expected state has been reached.

Conditional events. Besides successful login, a realistic login procedure should also handle attempts to log in with invalid credentials, as shown in Figure 3. The figure shows an extra state in which an error message is displayed. This extra state gives rise to an extra state object, corresponding to the display of the appropriate message. As action, it just has a close button leading back to the original login page.

The extra state naturally leads to another test scenario:

1. Go to the login URL.
2. Conduct the Authenticating self-check.
3. Enter invalid credentials and submit.
4. Conduct the Login Error self-check.
5. Hit close.
6. Conduct the Authenticating self-check.

In Figure 3 the edges are of the form

```
event [condition] / action
```

Thus, a trigger (click) can be conditional, and besides leading to a new state it can also result in an action (server side).

When testing such transitions, you trigger the event, ensure the condition is met, and then verify: that you can observe any effects of the required action; and that you reach the corresponding state.

Expanding Your State Chart

To drive the testing, you can expand the state chart to cater to additional scenarios. For example, related to authentication is registering a new user, shown in Figure 4. This Figure includes the Authenticating state, but not all of its outgoing edges. Instead, the focus is on a new *Registering* state and the transitions that are possible from there.

This again gives rise to two new state objects (for registration and for displaying an error message), and two additional scenarios. Thus, when developing tests for a given page, it is not necessary to consider the full state machine: focusing on states of interests is sufficiently helpful for deriving test cases.

Super states. States that have behavior in common can be organized into *super states* (also called OR-states). For example, once authenticated, all pages may have a common header, containing buttons for logging out, as well as

for navigating to key pages (for example, for managing your account or obtaining help, shown in Figure 5).

Edges going out of the super state (such as `logout`) are shorthand for an outgoing `logout` event for each of the four internal states (the substates). Expanding the shorthand would lead to the diagram in Figure 6 where the five outgoing edges of the super state are expanded for each of the four internal states, leading to 4 * 5 = 20 (directed) edges (drawn as two-way edges to keep the diagram manageable).

The typical way of implementing such super states is by having reusable HTML fragments, which in AngularJS, for example, are included via the ngInclude directive.[1]

In such cases, it is most natural to create a state object corresponding to the common `include` file. It contains presence checks for the required links or buttons and event checks to see if, for example, clicking settings indeed leads to the Settings state.

A possible test scenario would then be:

1. [Steps needed for login.]
2. Conduct Portfolio self-check.
3. Click settings link.
4. Conduct Settings self-check.
5. Click help link.
6. Conduct Help self-check.
7. Click account link.
8. Conduct Account self-check.
9. Click portfolio link.
10. Conduct Portfolio self-check.
11. Click logout link.
12. Conduct Authenticating self-check.

This corresponds to a single scenario testing the authenticated navigation pane. It tests that clicking the account link from the Help page works. It does not, however, check that clicking the account link from the Settings page works. In fact, this test covers only four of the 20 edges in the expanded graph.

Of course you can create tests for all 20 edges. This may make sense if the app-under-test has handcrafted the navigation pane for every view instead of using a single `include` file. In that case you may have reason to believe the different trails could reveal different bugs. Usually, however, testing all expanded paths would be overkill for the `include` file setting.

State traversals. The single test scenario for the navigation header visits five different states, in one particular order, shown in Figure 7. This is a rather long test and could be split into four separate test cases (Figure 8).

In unit testing, the latter would be the preferred way. It has the advantage of the four tests being independent: failure in one of the steps does not affect testing of the other steps. Moreover, fault diagnosis is easier, since the failed test case will be clearer.

The four independent tests, however, are likely to take considerably longer: every test requires explicit authentication, which will substantially slow down test execution. Therefore, in end-to-end testing it is more common to see shared setup among test cases.

In terms of JUnit's (http://junit.org) setup methods,[9] a unit test suite would typically make use of the @Before setup, which is executed again and again just before every @Test in the class. End-to-end tests, on the other hand, are more likely to use @BeforeClass in order to be able to conduct expensive setup methods only once.

Modal dialogs are typically used to disable any interaction until the user has acknowledged an important message ("Are you sure you..."). Examples are the login or sign-up error messages shown earlier. Such modal dialogs call for a separate state and, hence, for distinct page objects, offering an accept event to close the dialog.

Modal dialogs can be implemented using browser alerts (and WebDriver must accept them before testing can continue) or JavaScript logic. In the latter case, an extra check to be tested could be the dialog is indeed modal

Figure 5. Common behavior in a super state.

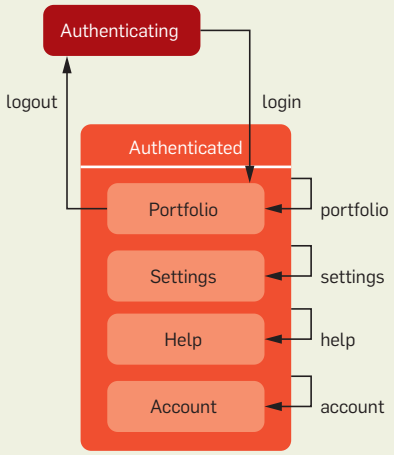


Figure 6. Expanding a super state.

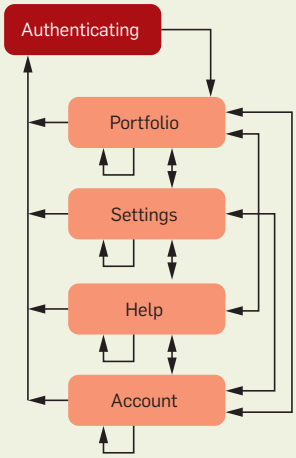


Figure 7. Single test traversing multiple states.

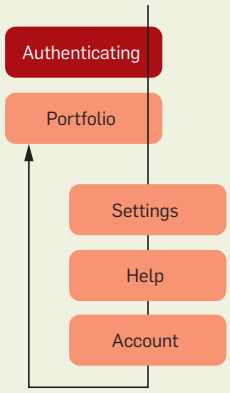


Figure 8. Separate tests for different states.

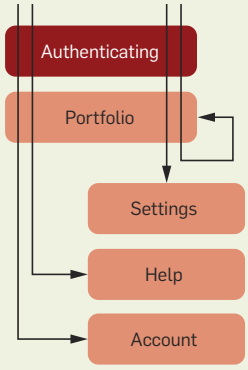


Figure 9. Login state machine with error handling and shared navigation.

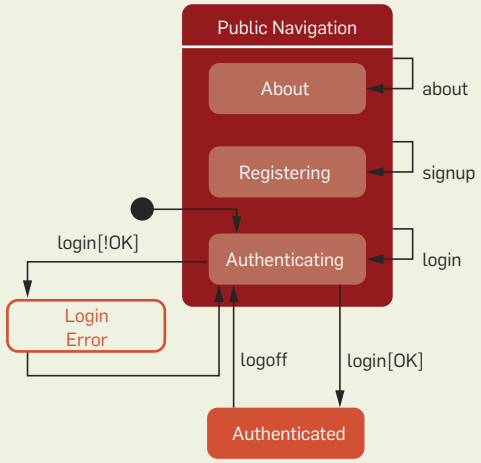


Figure 10. Using transition annotations to simplify diagrams.

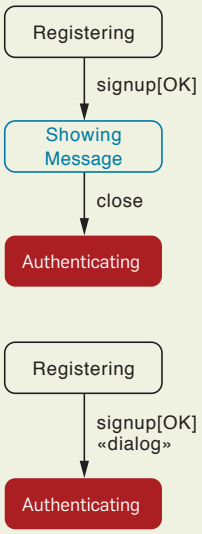
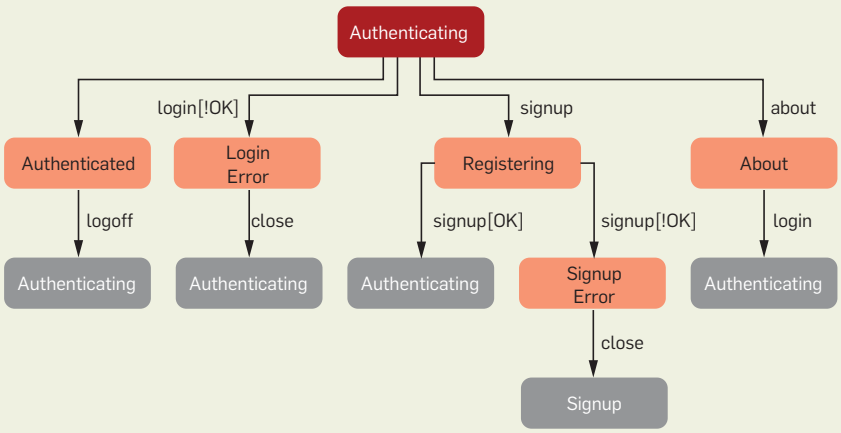


Figure 11. Transition tree.



(that is, that any other interaction with the page is disabled).

If the modal dialog is triggered by a state contained in a super state, the dialog state is *not* part of the super state (since the super-state interaction is disabled in the dialog). Thus, the correct way to draw the login state machine showing error handling and shared navigation would be as illustrated in Figure 9. Here the error dialog is not part of the navigation super state, as it permits only the close event and not clicking, for example, the about link.

Some applications are fairly dialog-intensive—for example, when dealing with registration, logging in, forgetting passwords, among others. Many of these dialogs serve only to notify the user of a successful state change. To simplify the state diagrams the dialogs can then be drawn as annotations on the edges, as in Figure 10.

The diagram at the top is the full version, and the one at the bottom is the abbreviated version. Note the `<<dialog>>` annotation is important for implementing the test. The test cases *must* click the close button of the dialog; otherwise, testing is blocked.

The transition tree. To support reasoning about state reachability, as well as state and transition coverage, it is helpful to turn a state diagram into a transition tree, as shown in Figure 11. (For more information, see Robert Binder's chapter on testing state machines in *Testing Object-oriented Systems*.[3])

The tree in Figure 11 has been derived from the state machine showing sign-up and authentication as presented earlier. Starting from the initial authenticating state, let's do a breadth-first traversal of the graph. Thus, per state you first visit its direct successors. If you enter a state you visited already, the visited state is drawn in gray as a leaf node. Then proceed to the next unvisited state.

The tree helps when designing tests for an individual state: the path to that state in the tree is the shortest path in the graph to that state. Besides, it clearly indicates which outgoing edges there are for a state.

The tree is also helpful for designing a test suite for the full state machine: writing one test case for each path from the root to a leaf yields a test suite that covers all transitions and, hence, covers all states in the machine.

Covering paths. While focusing on individual states and their transitions is a good way to spot and eliminate basic faults, a trickier set of defects is visible only when following a *path* along multiple states.

As an example, consider client-side caching. A framework such as AngularJS makes it easy to enable caching for (some) back-end HTTP calls. Doing this right improves responsiveness and reduces network round-trips, since the

results of back-end calls are remembered instead of requested over and over again.

If, however, the results are subject to change, this may lead to incorrect results. For example, the client may request an overview page with required information on one page, modify the underlying data on the next page, and then return to the original overview page. This corresponds to the red path in Figure 12.

With caching enabled, the Portfolio state will cache the back-end call results. The correct caching implementation of the Settings state would be to invalidate the cache if changes were made. As a result, when revisiting the Portfolio state the call will be made again, and the updated results will be used.

A test case for this caching behavior might look as follows:

1. [Take shortest route to Portfolio.]
2. Collect values of interest from Portfolio.
3. Click the settings link to navigate to Settings.
4. Modify settings that will affect Portfolio values of interest.
5. Click the portfolio link to navigate back to Portfolio.
6. Assert that modified values are displayed.

In the AngularJS application mentioned previously, this test case caught an actual bug. Unfortunately, it is difficult or expensive to come up with a test strategy that covers *all* such paths possibly containing bugs.

In the general case, in the presence of loops there are infinitely many potential paths to follow. Thus, the tester will need to rely on expertise to identify paths of interest.

The transition tree-based approach described previously provides so-called round-trip coverage[2]—that is, it exercises each loop once until it gets back at a node already visited (one round-trip). Assuming all super states are expanded, this strategy would lead to a test case for the caching example.

Alternative criteria include all length-N paths, in which every possible path of a given length must be exercised. The extra costs in terms of the increased number of test cases to be written can be substantial, however, so achieving such a criterion without automated tools is typically hard.

In terms of state objects, testing paths will not lead to new state objects—the states are already there. The need to assert properties along the path, however, may call for additional inspection methods in the state objects.

Going Backward

The browser's back button provides state navigation that requires special attention. While this button makes sense in traditional hypertext navigation, in today's Web applications it is not always clear what its behavior should be.

Web developers can alter the button's behavior by manipulating the history stack. As such, you want to be able to test a Web application's back-button behavior, and WebDriver provides an API call for it.

In terms of state machines, the back button is not a separate transition. Instead, it is a reaction to an earlier (click) event. As such, back-button behavior is a property of an edge, indicating the transition can be "undone" by following it in the reverse direction.

UML's mechanism for giving special meaning to elements is to use annotations (profiles). In Figure 13 explicit `<<back>>` and `<<noback>>` annotations have been added to the edges to indicate whether the back button can be used after the click to return to the initiating state. Thus, for simple navigation between the About, Registering, and Authenticating states, the back button can be used to navigate back.

Between the Authenticated and Authenticating state, however, the back button is effectively disabled: once logged off, clicking "Back" should not allow anyone to go to content requiring authentication. Knowing which transitions have special back behavior will then guide the construction of extra test scenarios verifying the required behavior.

Super States with History

As a slightly more sophisticated example of a super state, consider a table that is sortable across different columns. Clicking on a column header causes the table to be sorted, giving rise to a substate for every column (Figure 14).

The events come out of the super state, indicating they can be triggered from any substate and go into a par-

Figure 12. Faults on longer paths.

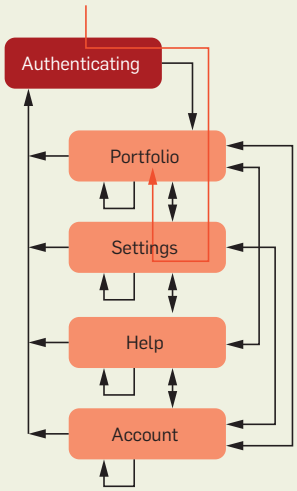


Figure 13. Back-button annotations.

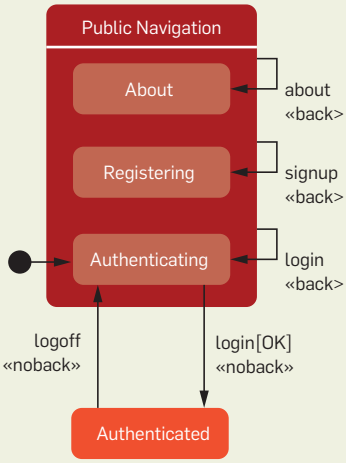


Figure 14. Super state with history.

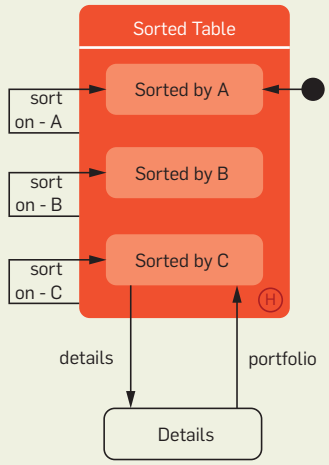


Figure 15. Orthogonal regions.

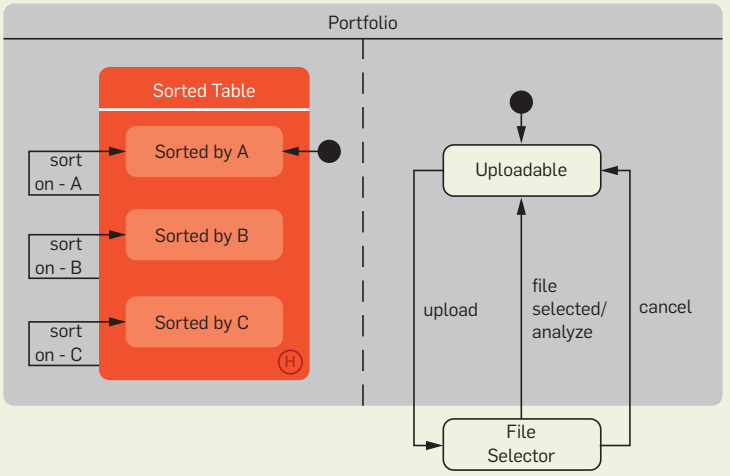


Figure 16. PhoneCat AngularJS Application.

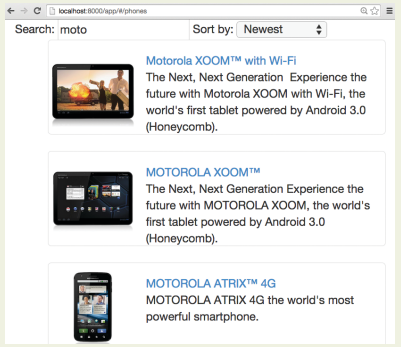
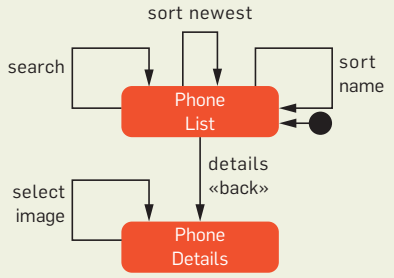


Figure 17. State diagram for phone catalog.



ticular substate. When leaving the sortable table page—for example, by requesting details for a given row—a design decision needs to be made about whether returning to that page (in this case by clicking the portfolio link) should yield a table sorted by the default column (A in this case) or should restore the sorting according to the last column clicked.

In UML statecharts, the first option (returning to super state's initial state) is the default. The second option (returning to the super state's state as it was before leaving) can be indicated by marking the super state as a History state, labeling it with a circled H. In both cases, if this behavior is important and requires testing, an extra path (scenario) is needed to verify the super state returns to the correct state after having been exited from a non-initial state.

And-states. Today's Web applications typically show a number of independent widgets, such as a contact list in one and an array of pictures in another. These widgets correspond to little independent state machines that are placed on one page.

In UML statecharts, such states can be described by *orthogonal regions* (also called AND-states), as shown in Figure 15. The Figure shows a Portfolio state, which consists of both a sortable table and an Upload button to add items. These can be used independently, as indicated by the two halves of the Portfolio state separated by the dashed line. The upload dialog is modal, which is why it is outside the Portfolio class. After uploading, the table remains sorted as it was, which is why it is labeled with the H.

Such orthogonal regions can be used to represent multiple little state machines present on one page. State transitions in these orthogonal regions may come from user interaction. They can also be triggered through server events (over Web sockets) such as push notifications for new email and stock price adjustments, among others.

From a testing perspective, orthogonal regions are in principle independent and therefore can be tested independently.

Like OR-states, AND-states can be expanded, in this case to make all possible interleavings explicit. This blows up the diagram considerably and, hence, the number of potential test cases. While testing a few of such interleavings explicitly makes sense and is doable, testing all of them calls for automated test generation.

State-based stories. Last but not least, states and state diagrams can also be helpful when describing requirements with user stories and acceptance scenarios. For example, there is a natural fit with the story format proposed by technology consultant Dan North.[9] Such stories consist of a general narrative of the form "as a... I want... so that...", followed by a number of acceptance scenarios of the form "given... when... then".

In many cases, these acceptance scenarios can be simply mapped to testing a single transition in the state diagram. The scenario then takes the form:

Given I have arrived in some state
When I trigger a particular event
Then the application conducts an action
And the application moves to some other state.

Thus, the state objects allow the API to interact with the state machine as suggested by these acceptance scenarios. A single test case moves from one state to another, and a full feature is described by a number of acceptance test cases navigating through the state machine, meanwhile checking how the application behaves.

AngularJS PhoneCat Example

As a complete example of testing with WebDriver and state objects, consider the AngularJS PhoneCat tutorial (https://docs.angularjs.org/tutorial). A screen shot of the PhoneCat application in action is shown in Figure 16. It comes with a test suite written in Protractor, which is the WebDriverJS[11] extension tailored toward AngularJS applications.

The application consists of a simple

list of phones that can be filtered by name and sorted alphabetically or by age. Clicking on one phone leads to full details for the given phone type.

The WebdriverJS test suite provided with the tutorial consists of three test cases for each of the two views (phone list and phone details), as well as one test for the opening URL, for a total of seven test cases.

The test suite in the original tutorial does not use page (or state) objects. To illustrate the use of state objects, I've rewritten the PhoneCat test suite to a state object-based test suite, which is available from my PhoneCat fork on GitHub (https://github.com/avandeursen/angular-phonecat/pull/1).

The state diagram I used for the PhoneCat application is shown in Figure 17. It leads to two state objects (one for each view). These state objects can be used to express the original set of scenarios. Furthermore, the state diagram calls for additional cases, for example for the 'sort-newest' transition not covered in the original test case.

The figure also makes clear there is no direct way to get from Phone Details to the Phone List. Here the browser's back button is an explicit part of the interaction design, which is why the `<<back>>` annotation was added to the corresponding transition. (Note this is the only edge with this property: clicking "Back" after any other transition while in the Phone List state exits the application, as per AngularJS default behavior).

Since the back button is essential for navigating between the two views, the state-based test suite also describes this behavior through a scenario.

Lastly, as the Protractor and WebDriverJS APIs are entirely based on asynchronous JavaScript promises,[11] the state object implementations are asynchronous as well. For example, the Phone List state object offers a method that "schedules a command to sort the list of phones" instead of blocking until the phones are sorted. In this way, the actual scenarios can chain the promises together using, for example, the `then` promise operator.

AngularJS in production. Most of the figures presented in this article are based on diagrams created for a Web application developed for a Delft University of Technology spinoff company. The application lets users register, log in, upload files, analyze and visualize them, and inspect analysis results.

The application's end-to-end test suite uses state objects. It consists of about 25 state objects and 75 scenarios. Like the PhoneCat test suite, it uses Protractor and consists of about 1,750 lines of JavaScript.

The end-to-end test suite is run from a TeamCity (https://www.jetbrains.com/teamcity/) continuous integration server, which invokes about 350 back-end unit tests, as well as all the end-to-end scenarios upon any change to the back end or front end.

The test suite has helped fix and find a variety of bugs related to client-side caching, back-button behavior, table sorting, and image loading. Several of these problems were a result of incorrect data bindings caused by, for example, typos in JavaScript variable names or incomplete rename refactorings. The tests also identified back-end API problems related to incorrect server configurations and permissions (resulting in, for example, a 405 and an occasional 500 HTTP status code), as well as incorrect client / server assumptions (the JavaScript Object Notation returned by the server did not conform to the front end's expectations).

Conclusion

When doing end-to-end testing of a Web application, use states to drive the tests:

- Model interactions of interest as small state machines.
- Let each state correspond to a state object.
- For each state include a self-check to verify the browser is indeed in that state.
- For each transition, write a scenario conducting self-checks on the original and target states, and verify the effects of the actions on the transition.
- Use the transition tree to reason about state reachability and transition coverage.
- Use advanced statechart concepts such as AND-states, OR-states, and annotations to keep your diagrams concise and comprehensible.
- Consider specific paths through the state machine that may be error prone; if you already have state objects for the states on that path, testing the behavior along that path should be simple.
- Exercise the end-to-end test suite in a continuous integration server to spot integration problems between HTML, JavaScript, and back-end services.

As with page objects, the details of the browser interaction are encapsulated in the state objects and hidden from the test scenarios. Most importantly, the state diagrams and corresponding state objects directly guide you through the overall process of test-suite design. C

Related articles on queue.acm.org

Rules for Mobile Performance Optimization
Tammy Everts
http://queue.acm.org/detail.cfm?id=2510122

Scripting Web Service Prototypes
Christopher Vincent
http://queue.acm.org/detail.cfm?id=640158

Software Needs Seatbelts and Airbags
Emery D. Berger
http://queue.acm.org/detail.cfm?id=2333133

References

1. AngularJS. ngInclude directive; https://docs.angularjs.org/api/ng/directive/ngInclude.
2. Antoniol, G., Briand, L.C., Di Penta, M. and Labiche, Y. A case study using the round-trip strategy for state-based class testing. In *Proceedings 13th International Symposium on Software Reliability Engineering*. IEEE (2002), 269–279.
3. Binder, R.V. *Testing Object-oriented Systems.* Addison-Wesley, Reading, PA, 1999, Chapter 7.
4. Fowler, M. PageObject, 2013; http://martinfowler.com/bliki/PageObject.html.
5. Harel, D. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8(3): 231–274.
6. Horrocks, I. *Constructing the User Interface with Statecharts.* Addison-Wesley, Reading, PA, 1999.
7. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C. Improving test suites maintainability with the page object pattern: An industrial case study. In *Proceedings of the Testing: Academic and Industrial Conference—Practice and Research Techniques*. IEEE (2013), 108–113.
8. Mesbah, A., van Deursen, A. and Roest, D. Invariant-based automatic testing of modern Web applications. *IEEE Transactions on Software Engineering* 38, 1 (2012) 35–53.
9. North, D. What's in a story?; http://dannorth.net/whats-in-a-story/.
10. Selenium. Page Objects, 2013; https://github.com/SeleniumHQ/selenium/wiki/PageObjects
11. Selenium. Promises. In WebDriverJS User's Guide, 2014; https://code.google.com/p/selenium/wiki/WebDriverJs#Promises.
12. SeleniumHQ. WebDriver; http://docs.seleniumhq.org/projects/webdriver/.

Acknowledgments

Thanks to Michael de Jong, Alex Nederlof, and Ali Mesbah for many good discussions and for giving feedback on this post. The UML diagrams for this post were made with the free UML drawing tool UMLet (version 13.2).

Arie Van Deursen is a professor at Delft University of Technology where he leads the Software Engineering Research Group. To help bring his research into practice, he co-founded the Software Improvement Group in 2000 and Infotron in 2010.