Extracted from:

Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

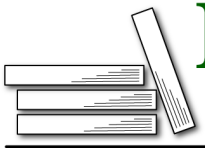
The Pragmatic Programmers

Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

Elisabeth Hendrickson

Edited by Jacquelyn Carter



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-02-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2013

Have you ever encountered a failure that was extremely difficult to reproduce? Perhaps you've seen a catastrophic error that happens only sporadically, or maybe you stumbled on corrupted data and could not trace the root cause.

Such defects are often triggered when something happens during a brief window of vulnerability: a moment in time when all the conditions line up just right so something can go very wrong. A file happens to be locked just when the software attempts to write to it. Your session times out just at the moment you try to access secure content. A race condition exists where on rare occasions one part of the system attempts to update a record before another part finishes creating it.

These conditions are usually short-lived and hard to spot. You might not know how to cause them intentionally or even how to tell when they're open so you can take advantage of them. If you can't see the window of vulnerability, discovering or reproducing bugs related to it becomes a frustrating matter of trial and error.

Fortunately, there's a systematic approach to discovering and exploiting these windows of vulnerability using state models. In this chapter, you'll learn how to map a state model and use heuristics to surface surprises related to timing.

It's important to note that you'll get more out of this chapter if you focus on observable behavior rather than on implementation details. Modeling from an external perspective enables you to see states and responses that stem from the whole technology stack and not just from the layer of that stack with which you are most familiar. So even if you happen to have knowledge of the internals of the software, you'll need to set that knowledge aside so that you can focus on what you see happening rather than on what you know is going on under the covers.

8.1 Recognizing States and Events

Noticing state transitions and the events that trigger them is easier said than done. Sometimes the states are subtle and fleeting, triggered by the passage of time or by an event you cannot control directly. Such states are especially important to identify because they are usually the most interesting to explore, yet their transitory nature makes them difficult to see. Similarly, noticing all the conditions that trigger a shift in states requires careful attention to detail. In this section you'll find a handy guide to identifying even hard-to-spot states and events.

States Are Behavioral Modes

My friend Alan A. Jorgensen uses three simple questions to detect states:

- Are there things I could do now that I could not do before?
- Are there things I cannot do now that I could do before?
- Do my actions have different results now than before?

Whenever the questions reveal differences in behavior, Alan knows he has found a new state.

You can also detect states by attending to the language you use to describe what's happening. Watch for situations where you use the word *while* in describing the software, such as in these examples:

- "While the system is importing data…"
- "While the system is running the report…"
- "While the account is suspended…"
- "While the call is on hold…"
- "While the server collects usage data…"

Any time you can use the word *while* in describing behavior, you've identified a state.

Consider an example. Imagine you are identifying states associated with logging into the system. You are currently on the login screen, as shown in Figure 2, *The login user interface*, on page 7.

The login screen represents a state: the system is waiting for login credentials. You can't do anything interesting until you log in. You might call the current state "Logged Out."

Once you provide a username and password, you might notice that there is a period of time after you enter the login information and before you see the first screen of the application. There's an animated spinning wheel signaling you that something is happening.

When the spinner is on the screen, you could say "While it's authenticating…" Aha! "Authenticating" is another state. It's a transitional state that will end when the system finishes taking whatever steps are necessary to authenticate your credentials.

Events Trigger State Transitions

User actions are the most obvious kind of event: you take an action and it triggers a response in the system. In the login example above, you click the

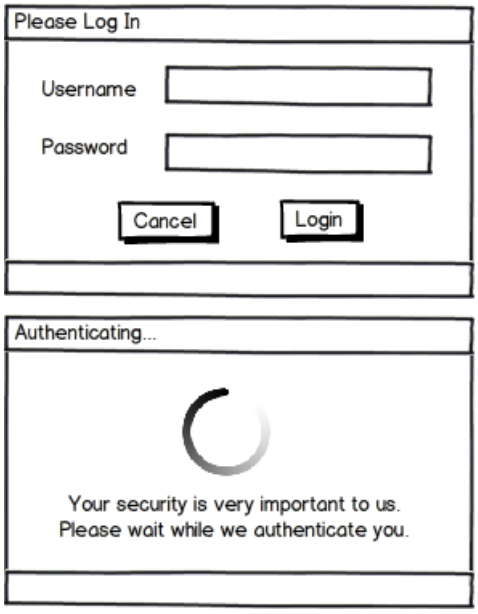


Figure 2—The login user interface

Login button and the system transitions into the authenticating state. User actions aren't confined to GUI interfaces. You might issue a command at the command line or make a server request using an API.

However, user actions are just one kind of event. Other kinds of events are harder to control and thus may be harder to spot. As you're looking for events, consider the following:

- *Externally generated events*: any event coming from outside the software. For example, if your software monitors the contents of a directory on the file system, changing the contents of that directory is an example of an externally generated event.
- *System-generated events*: any event triggered by the software itself. Often these events are the result of the system completing some background activity: loading or exporting data, connecting to a remote server, authenticating a user, or performing some background calculation. Any time you notice that there is a delay between your action and the system's response, it is likely that there is an interstitial state and associated system-generated event hidden behind the scenes.
- *Passage of time*: some events, such as timeouts, are simply the result of the passage of time. Look for conditions that occur after a given duration,

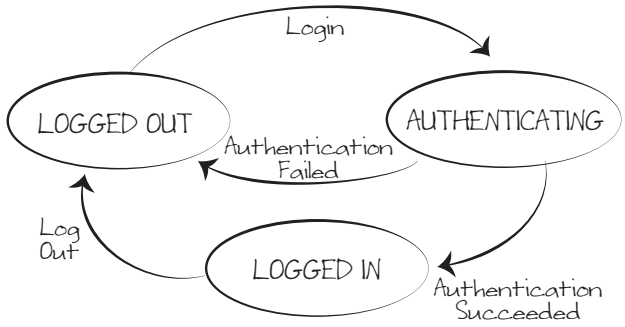
such as alerts. Also watch for events that occur at a specific time, such as a backup scheduled for midnight every night.

Finally, remember that every state is triggered by an event. Thus, if you identify a state, back up to identify the event that triggered the state. As you are doing this analysis, you're likely to find that you switch back and forth between looking for events and looking for states since the two things are so inextricably intertwined.

8.2 Diagramming a State Model

Now that you know how to recognize states and events, you can draw a diagram of the relationship between them.

For example, a state model for the login example might look like the following. Note that states appear in circles, while arrows between the circles represent the possible transitions. The labels on the arrows are the events.



Creating a state model takes more than just noticing states and events. So many states and events exist even in a simple system that the task can quickly become overwhelming. Here are some strategies to keep you on track.

Narrow Your Focus

You can narrow the scope of your model by identifying a single target, such as a feature or workflow. Take uploading a file: the workflow involved might involve states like Selecting a File, Uploading, and Confirming Upload.

You might even choose to focus on the life cycle of an entity within the system. For example, imagine you are exploring a bug-tracking system (something that is near and dear to many testers' hearts). A given bug record is an entity. Bugs typically have an entire life cycle, with states like New, Assigned, Fixed, Deferred, Verified, and Closed. User accounts are another, different entity that might have states like Active or Suspended.

Choosing a focused target is important. Otherwise you'll find yourself suffering from analysis paralysis, identifying more and more states and transitions but not having any time left to use that knowledge to help your explorations. If you think you might not have chosen a sufficiently well-defined target, try naming it. If you cannot give it a simple name, it's probably more than one target.

Identify a Perspective

Consider a phone call. You could map states from the perspective of the caller. In that case, an event might include taking the phone off the hook to get a dial tone. Another event might include putting the phone back on the hook. If the caller pressed the speakerphone button before replacing the handset on the phone (another event), then the call will be in a different state than if the caller simply hung up the handset.

Alternatively you could map the states from the perspective of the call. The call would not come into existence until two parties are connected. Once connected, the call might be placed on hold, forwarded, or connected with a third party. The events might still involve user actions on a handset (e.g., pressing the Hold button), but the perspective dictates which states and events belong in the diagram and thus narrows the scope.

Dial Up or Down the Level of Abstraction

If you are modeling a simple small interaction, you can do so at a low level of detail. You can map out all the teeny transitory states between user-generated events. This will give you tremendous insight into those pernicious, narrow windows of vulnerability.

However, you'll need to work at a higher level of abstraction if your target is something bigger. Instead of listing out numerous separate transitory states, you'll lump them together into one big state with a less specific name.

Imagine you are modeling states associated with launching some kind of client program that connects to a server. It could be any kind of client: a point-of-sale system, an email client, a music player…anything.

If you chose a very narrow target, like the launch process, you would identify states at a lower level of abstraction. So you would look for indications of distinct transitory states between the time the user double-clicked on the icon for the client and the time the client is ready for the user to begin interacting with it. You might notice states like "Connecting to the Database" and "Caching Data."

If, however, your target is the states of the client itself, discovering all the transitory states associated with the launch would be too much detail. You'd get bogged down in analyzing fine-grained details when you really wanted to look at the big picture. So you would roll all those substates associated with the client starting up into a single state, "Launching."