Testing with State Machines (ISQTB 4.3.3, ACM Queue)

Arie van Deursen

What is true about the following statements:

I. Virus software on your computer incorrectly identifies a harmless program as a malicious one.

II. An incorrectly written unit test succeeded during the component testing phase, but failed after the test was fixed.

A. Statement I is false negative and statement II false positive.

B. Statement I is false positive and statement II is false negative.

C. Both statements are false positive.

D. Both statements are false negative.

What is true about the following statements:

I. Virus software on your computer incorrectly identifies a harmless program as a malicious one.

II. An incorrectly written unit test succeeded during the component testing phase, but failed after the test was fixed.

A. Statement I is false negative and statement II false positive.

B. Statement I is false positive and statement II is false negative.

C. Both statements are false positive.

D. Both statements are false negative.

Why would you conduct regression testing?

A. To confirm a fixed defect has actually been fixed.
B. To ensure that no new defects popped up after modifications.
C. To test the impact of a change environment to an operational system.
D. To validate that the right software has been build.

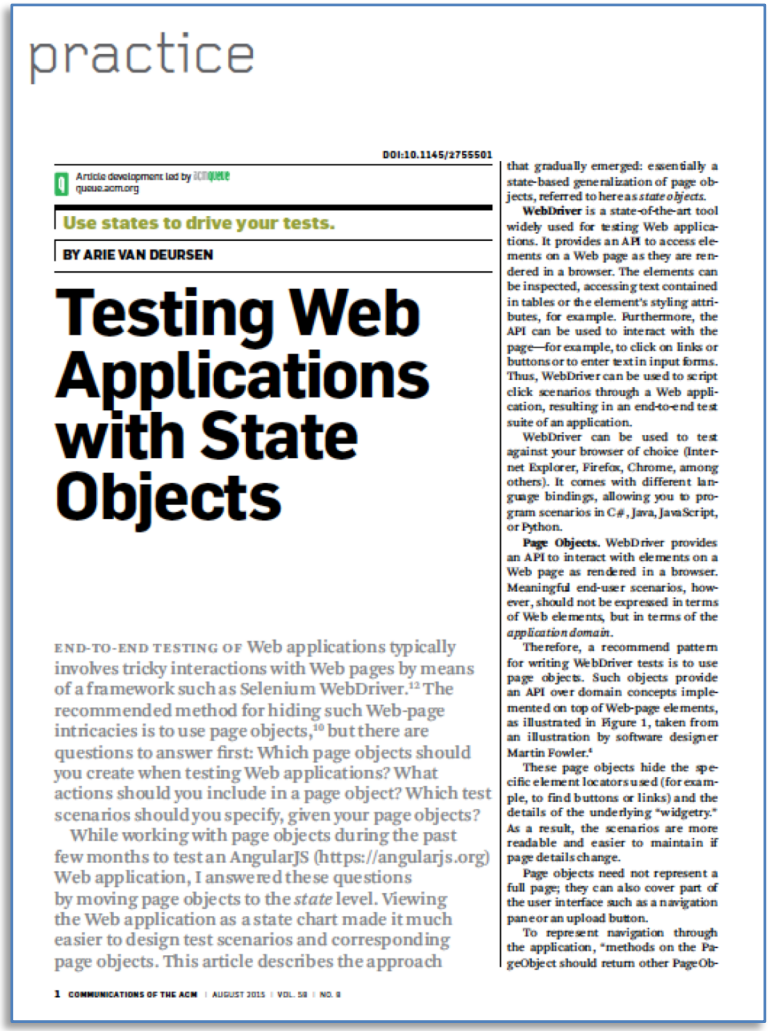
Why would you conduct regression testing?

A. To confirm a fixed defect has actually been fixed.

B. To ensure that no new defects popped up after modifications.
C. To test the impact of a change environment to an operational system.
D. To validate that the right software has been build.

State Based Testing

- Specifying State Machines Effectively
- Deriving Test Suites from State Machines
- Implementing State Based Test Suites



practice

DOI:10.1145/2755501

Article development led by 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*.

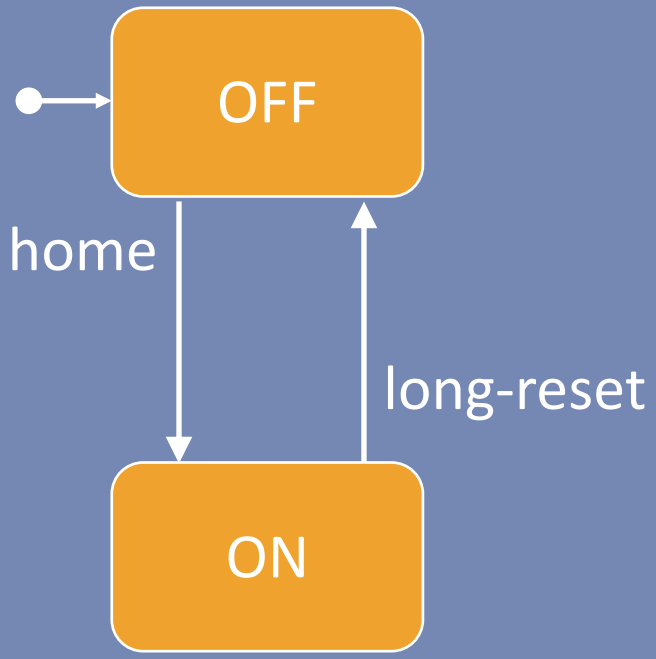
Therefore, a recommend pattern for writing WebDriver tests is to use page objects. Such objects provide an API over domain concepts implemented on top of Web-page elements, as illustrated in Figure 1, taken from an illustration by software designer Martin Fowler.[4]

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.

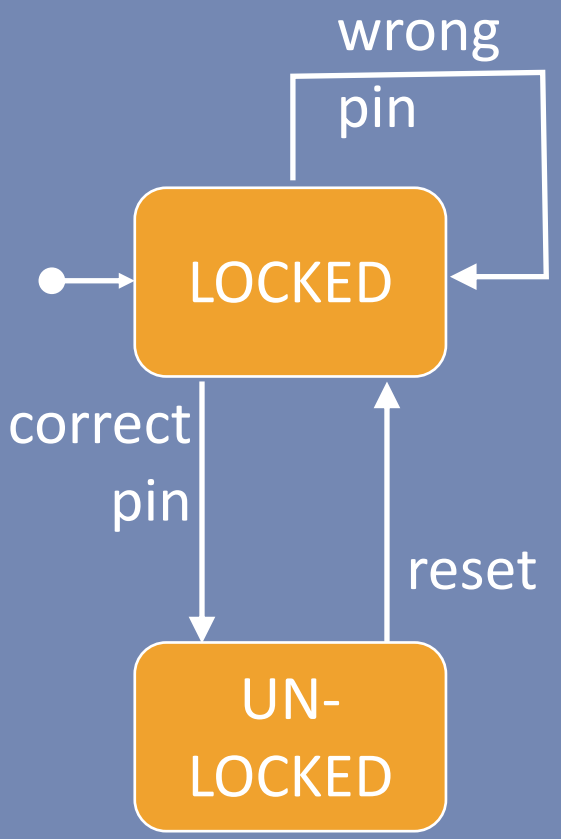
Page objects need not represent a full page; they can also cover part of the user interface such as a navigation pane or an upload button.

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

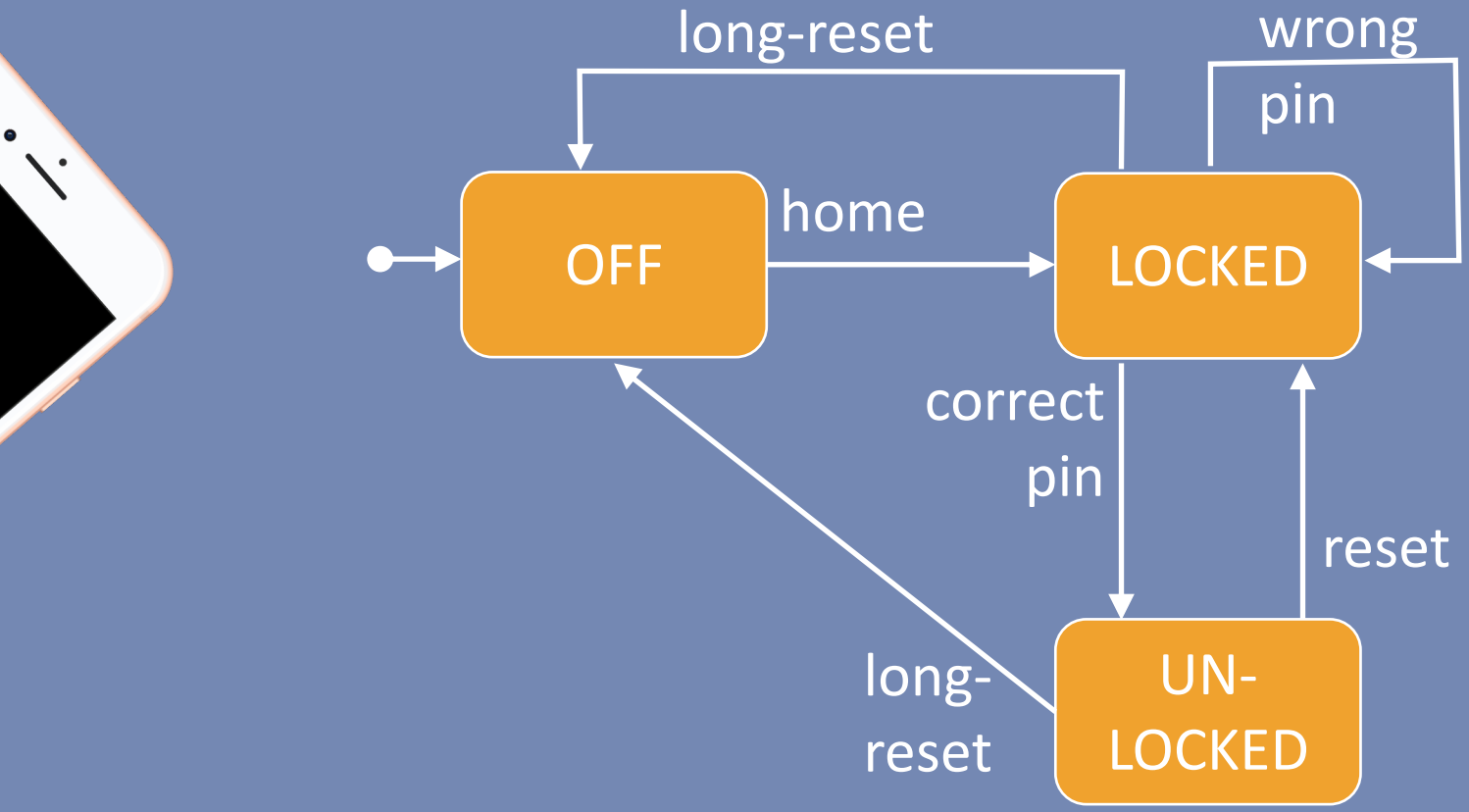
State Machines: On versus Off



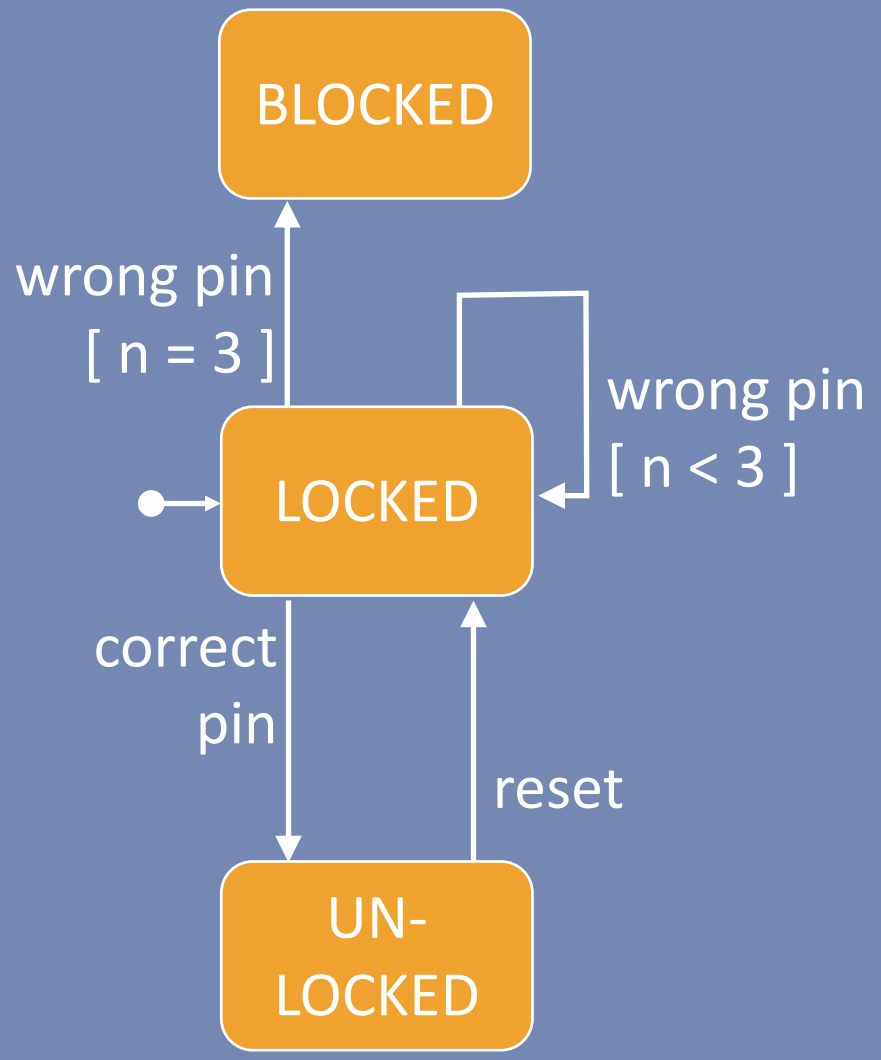
Unlocking



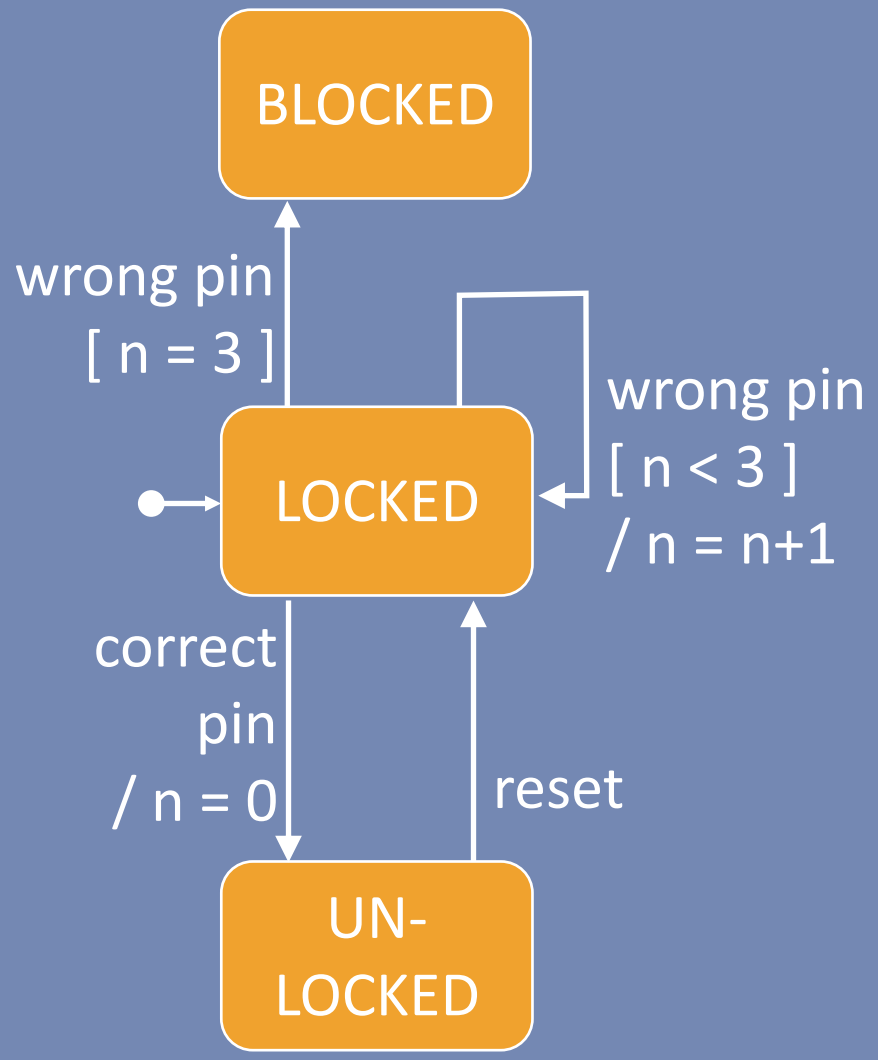
On/Off + Unlocking



Conditional Transitions

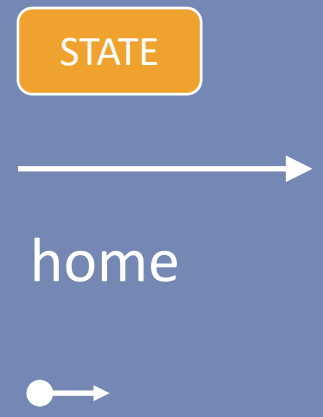


Actions


BLOCKED
wrong pin
[n = 3]
LOCKED
wrong pin
[n < 3]
/ n = n+1
correct
pin
/ n = 0
reset
UN-
LOCKED

UML State Diagrams

- States
- Transitions
- Events
- Initial state
- [Conditions]
- / Actions

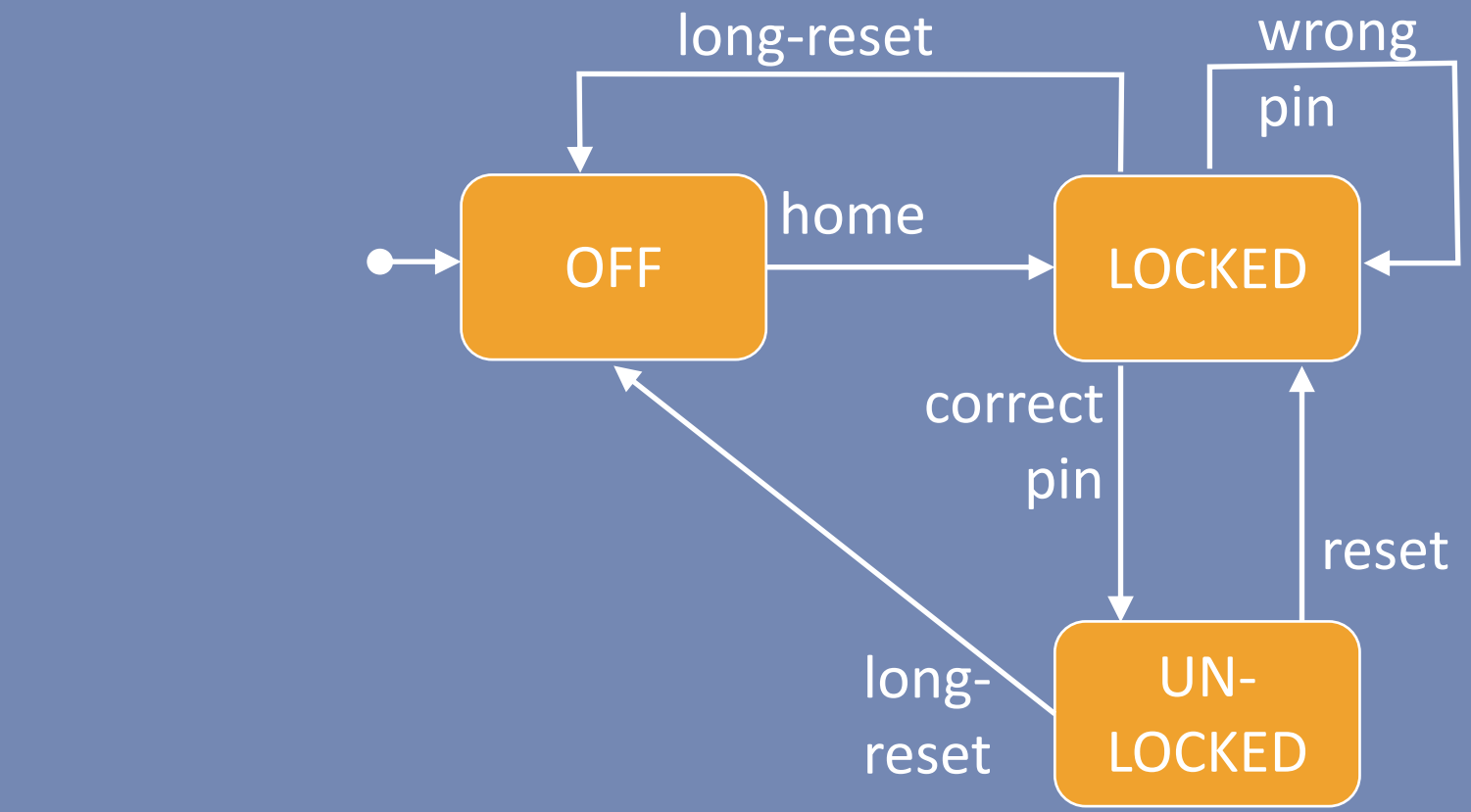


States

- State:
  - reflects as much of a system’s history
  - as needed to determine current allowed behavior

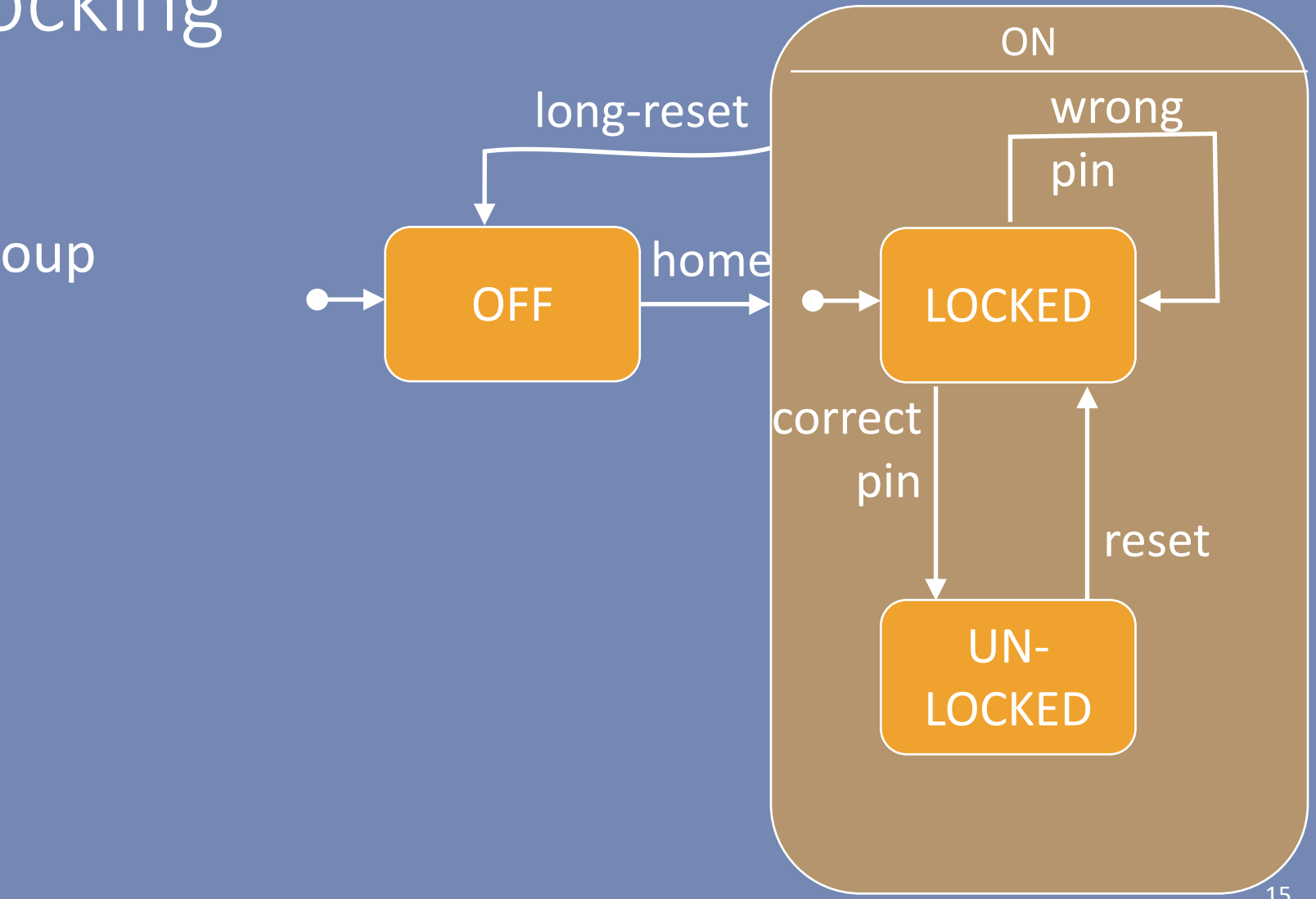
- State machine:
  - Imposes constraints on ordering of events

Super States in UML State Charts: On/Off + Unlocking



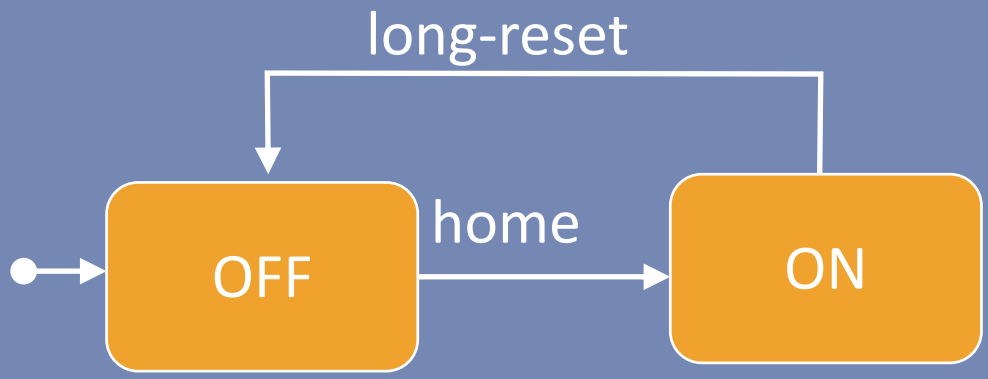
Super States in UML State Charts: On/Off + Unlocking

- Super states can group states together

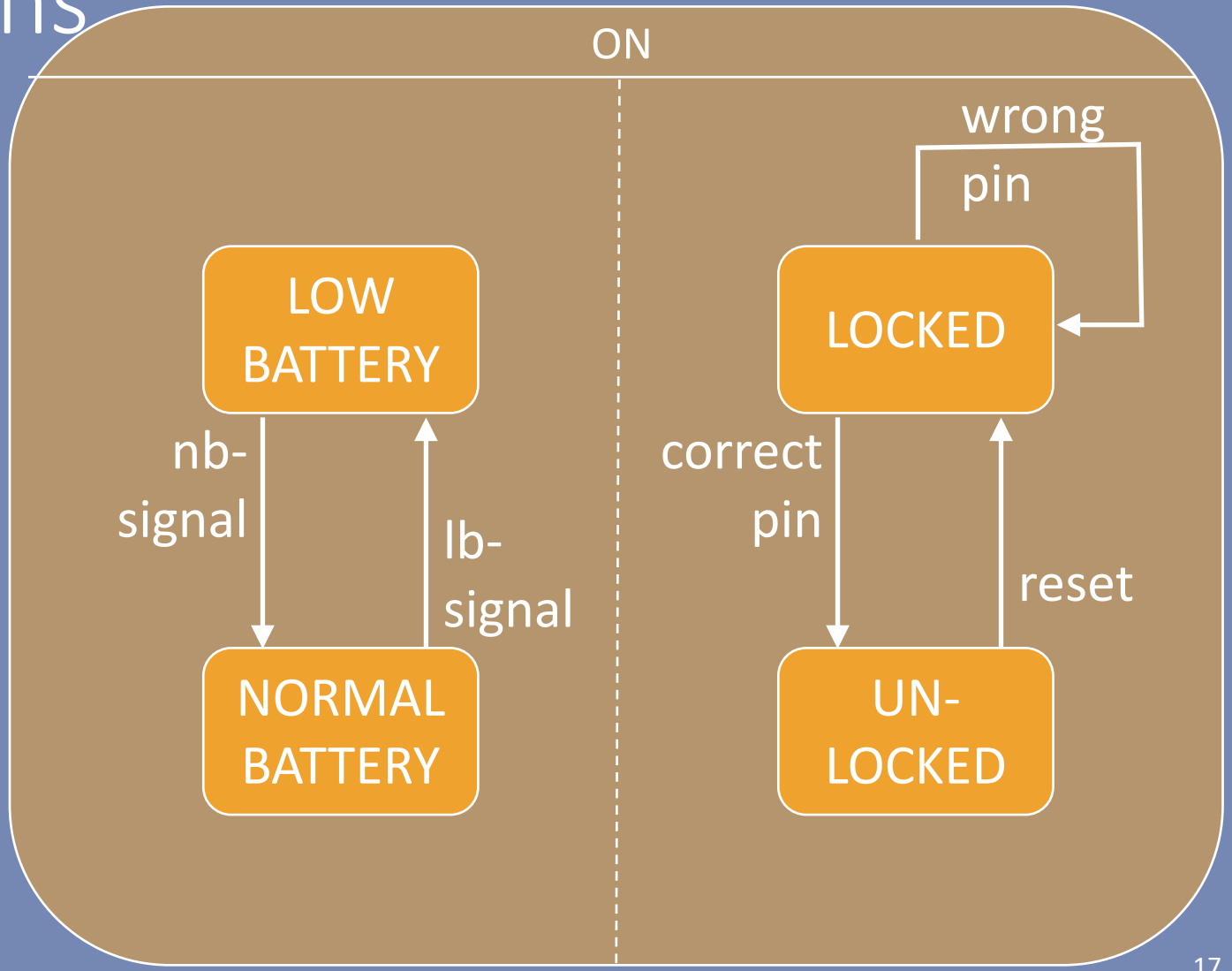


Super States in UML State Charts: On/Off + Unlocking

- Super states can group states together
- Super states can be (un)folded
- Outgoing transitions depart from *all* sub states



“And”-States in UML State Charts: Concurrent Regions



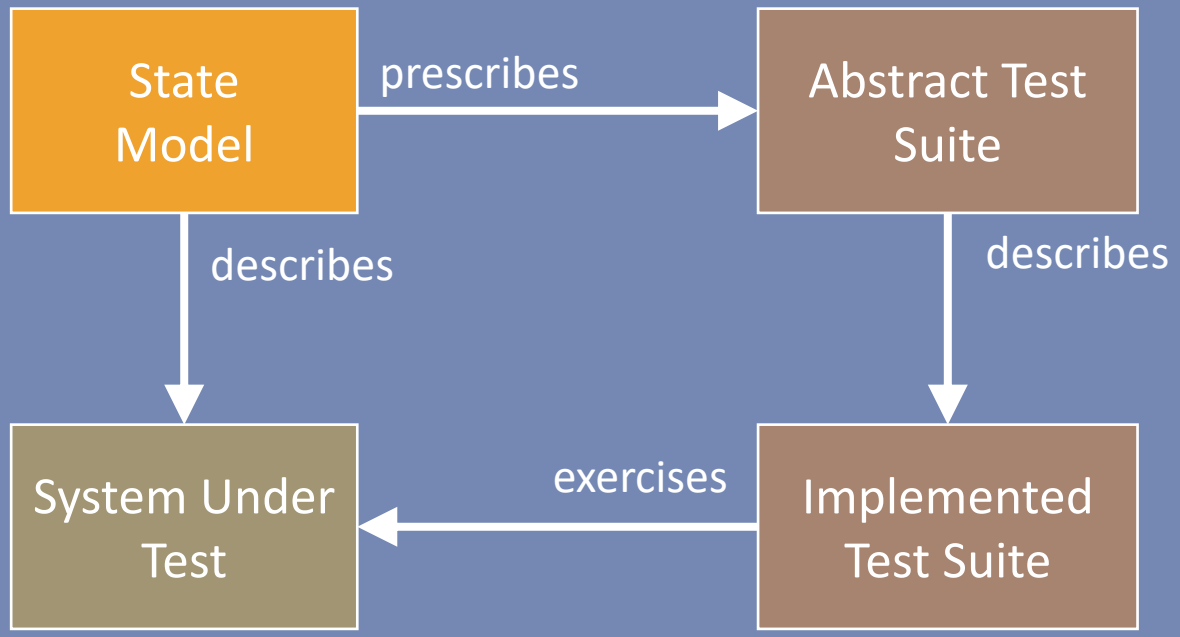
Creating a State Machine

- Create multiple smaller ones
- Focus on ‘regions of interest’
- Optionally use conditions on the edges
- Optionally describe actions
- Use super-states to group common behavior

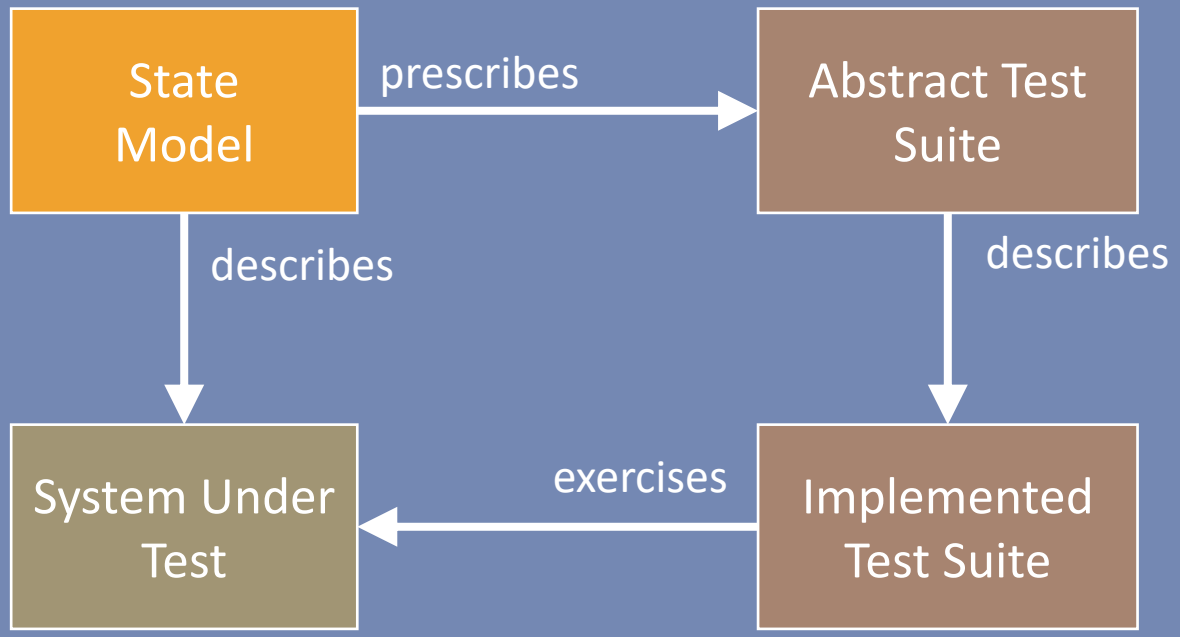
“Useful” in our setting:
Testing!

“All models are wrong, but some are useful”
(George Box, 1976)

State-Based Testing



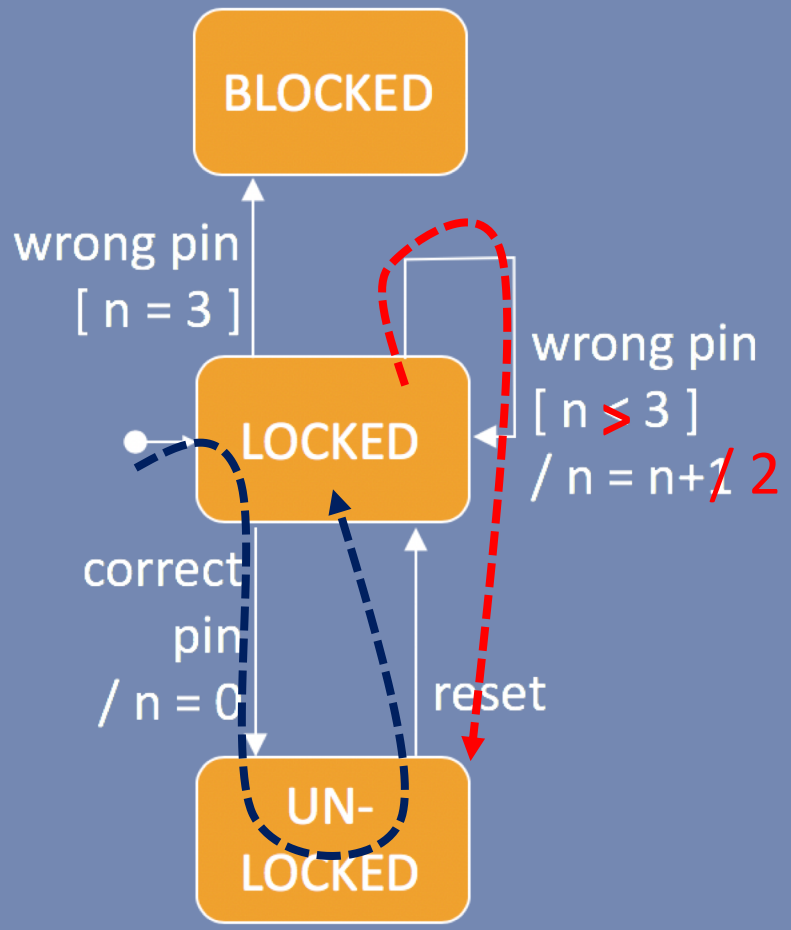
State-Based Testing



Fault Model

- Does event lead to the correct state?
- Does event trigger with the right condition?
- Does event invoke the correct action?
- Do states behave the same on second visit?
  - *(“History Sensitivity”)*
- ...

Derive test suite from state chart to find such faults



State Machine *Test Adequacy*

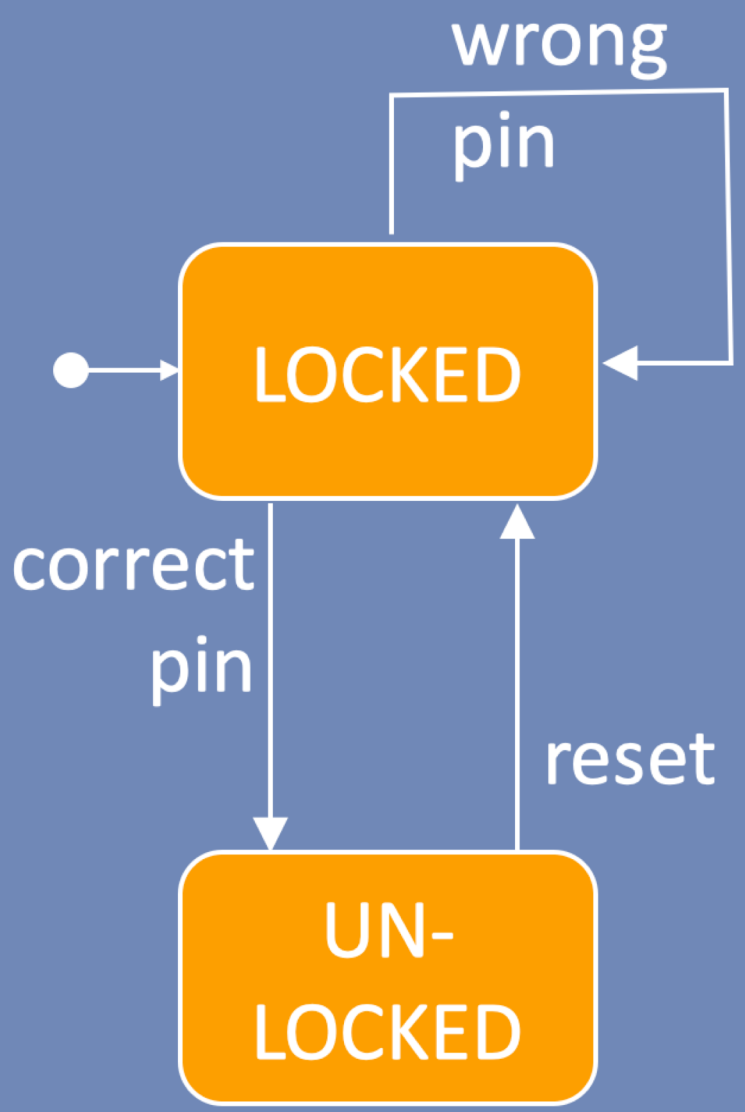
- State coverage:
  - Reach every state
- Transition coverage
  - Exercise every transition
- Path coverage
  - Exercise sequence of transitions

- Test can be considered "adequate" with respect to a *criterion*
- *Coverage* is the percentage of elements (as defined in criterion) that is actually covered

ISTQB: “Transition Testing”

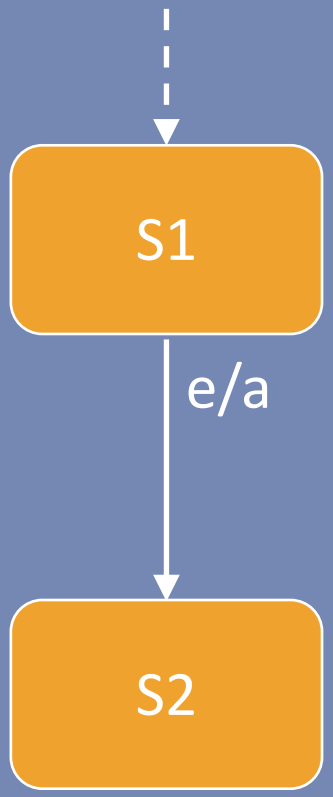
- Each transition is a “test condition”.
- Transition coverage: Percentage of transitions covered
- One test case is *path,* and can cover multiple transitions
- Which test cases to derive?

Testing Unlocking



1. Startup
2. Check you're "Locked
3. Enter correct pin
4. Check you're "Unlocked"
5. Hit reset
6. Check you're "Locked"

Testing One Transition



- Bring program in state S1
- check you're indeed in state S1

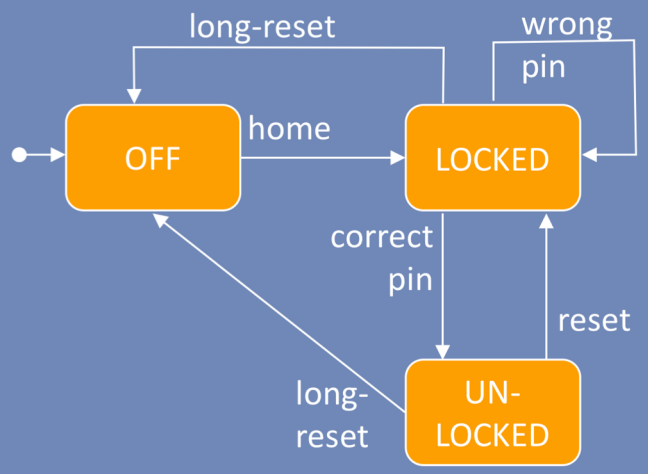
- trigger event e

- check effect of action *a* on e
- check you've arrived in state S2

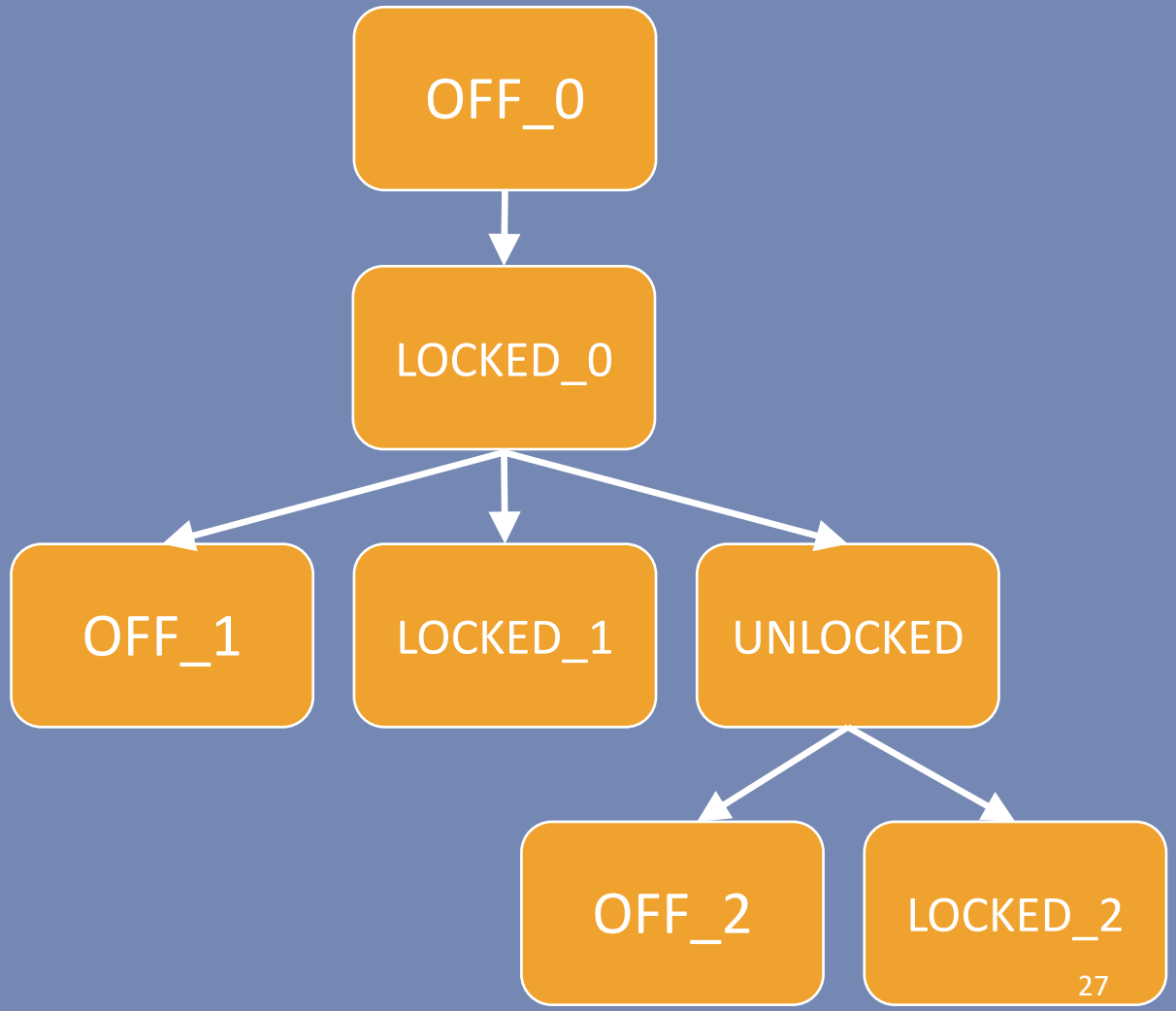
Testing a Full State Diagram

- Cover all transitions?
- What about paths longer than one transition?
- What about loops?

- Turn graph into a *transition tree*.
  - Breadth-first graph traversal

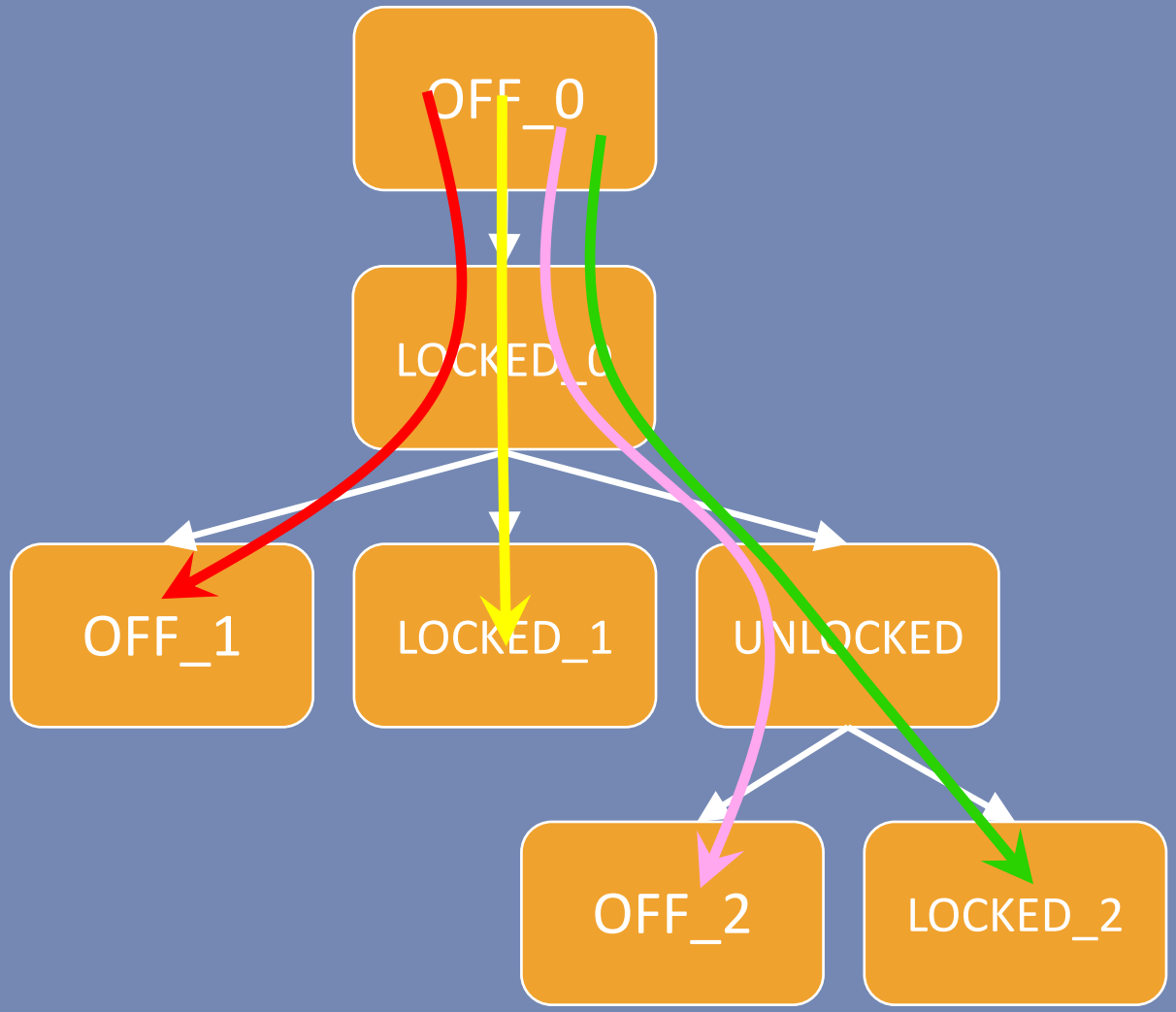


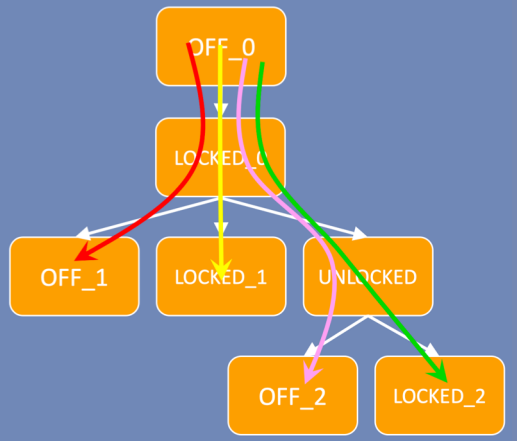
The Transition *Tree*



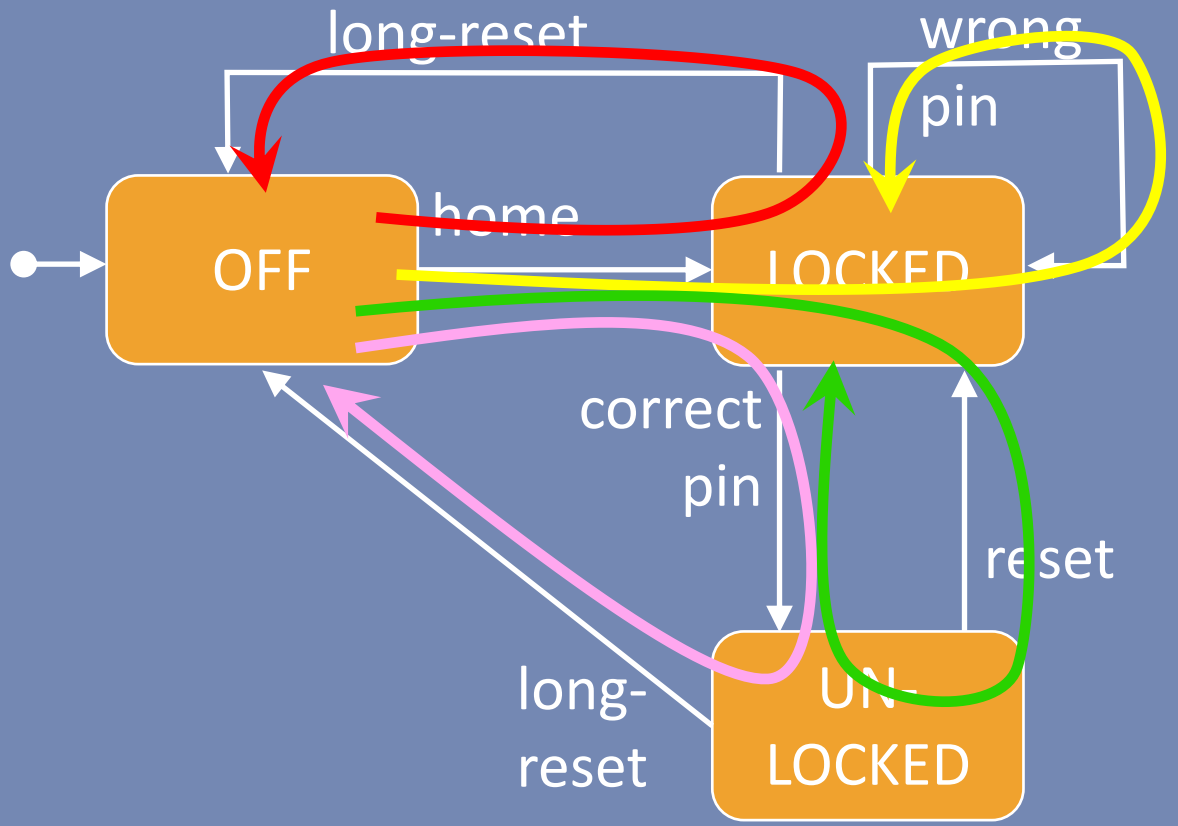
Tests from the Transition Tree

- Each path from root to leaf is a test case
- Covers multiple transitions
- Unrolls loops once

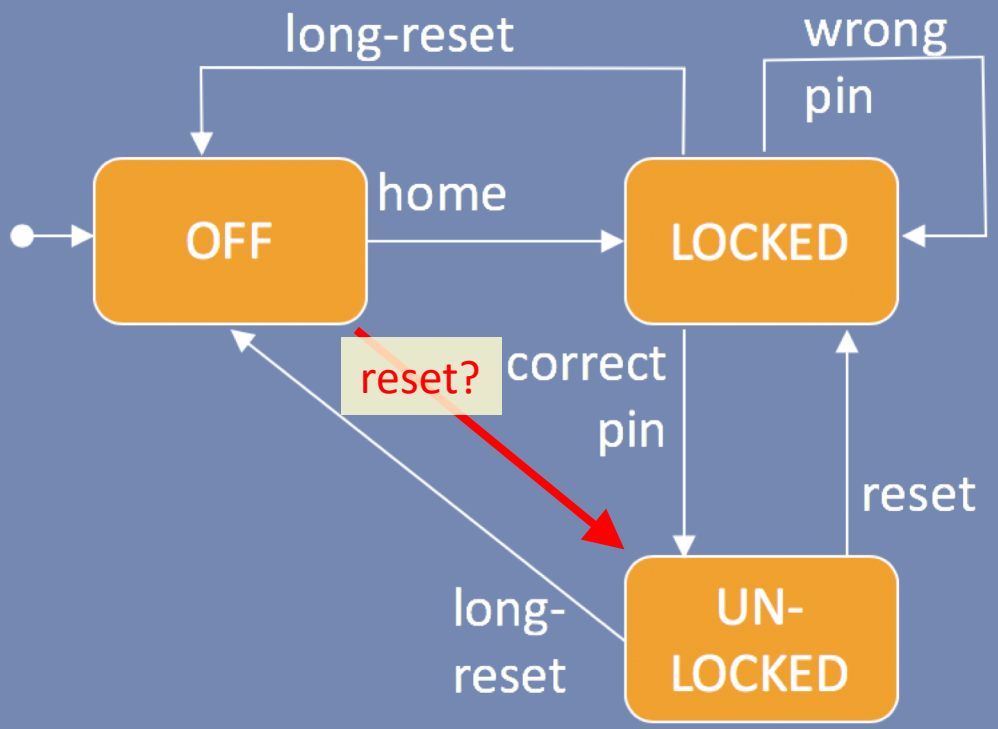




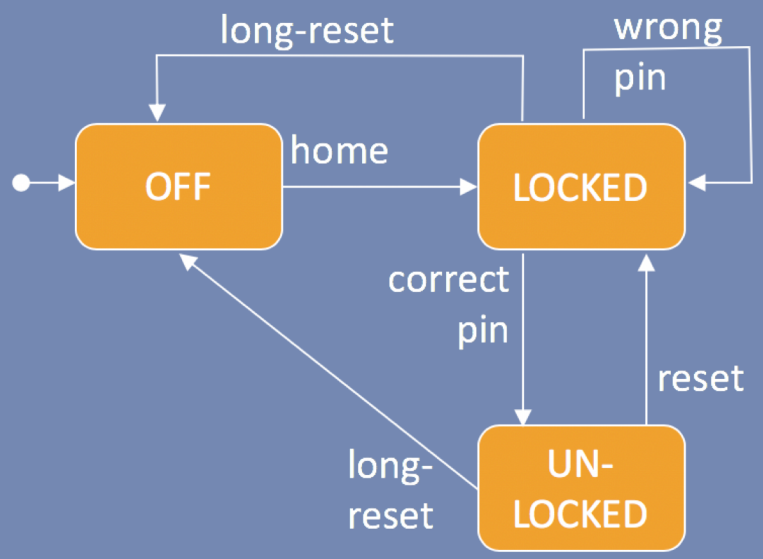
Which Paths to Test?



Sneak Path Testing



Transition Table



STATES	events				
	home	long-reset	reset	wrong-pin	correct-pin
OFF	LOCKED				
LOCKED		OFF		LOCKED	UNLOCKED
UNLOCKED		OFF	LOCKED		

Sneak Path Testing

- Create transition table
- Determine effects of empty cells
  - Default: do nothing
- Test each empty cell:
  - Default action
  - No state change

STATES	events				
	home	long-reset	reset	wrong-pin	correct-pin
OFF	LOCKED				
LOCKED		OFF		LOCKED	UNLOCKED
UNLOCKED		OFF	LOCKED		

- Verifies that illegal transitions cannot occur

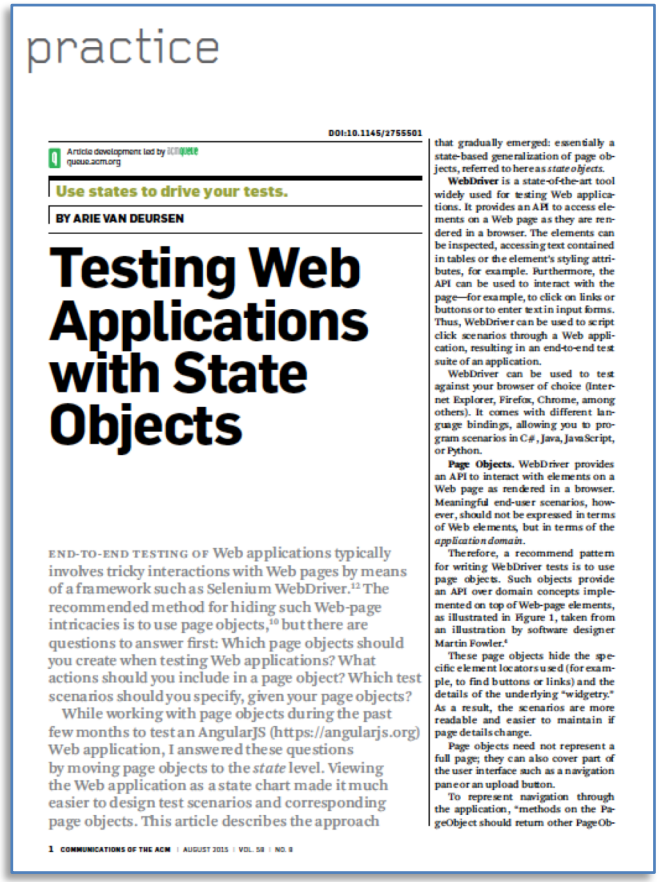
Implementing State-Based Tests

- Class as a state machine
- Inspection methods
  - See current state
- Trigger methods
  - Transition to next state

- Test scenario:
  - sequence of triggers and inspections

State Based Web Application Testing

- Specifying State Machines Effectively
- Deriving Test Suites from State Machines
- Implementing State Based Test Suites



practice

DOI:10.1145/2755501

Article development led by 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 hereas *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]

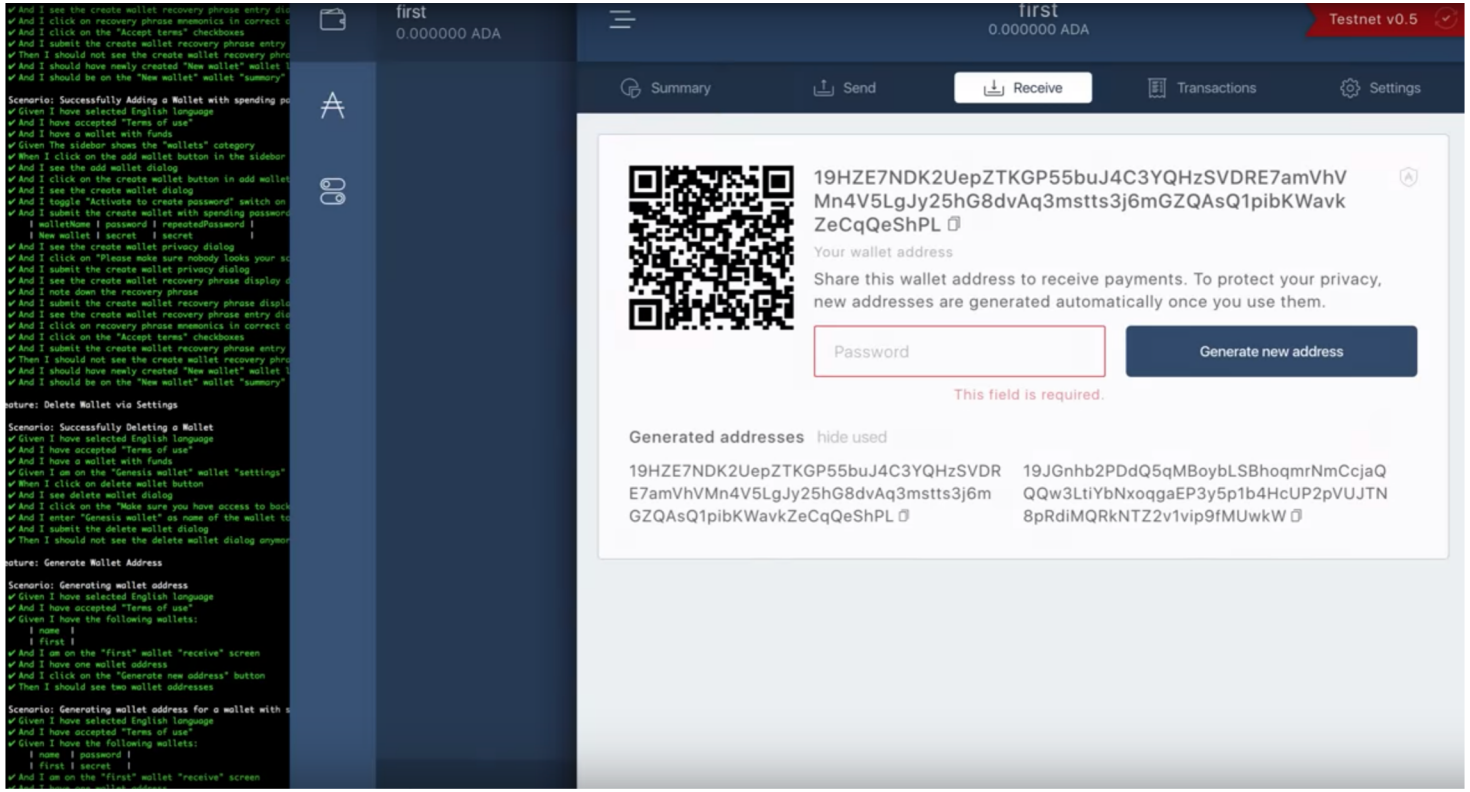
These page objects hide the specific element locators used (for example, to find buttons or links) and the details of the underlying "widgetry." As a result, the scenarios are more readable and easier to maintain if page details change.

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-

1 COMMUNICATIONS OF THE ACM | AUGUST 2015 | VOL. 58 | NO. 8

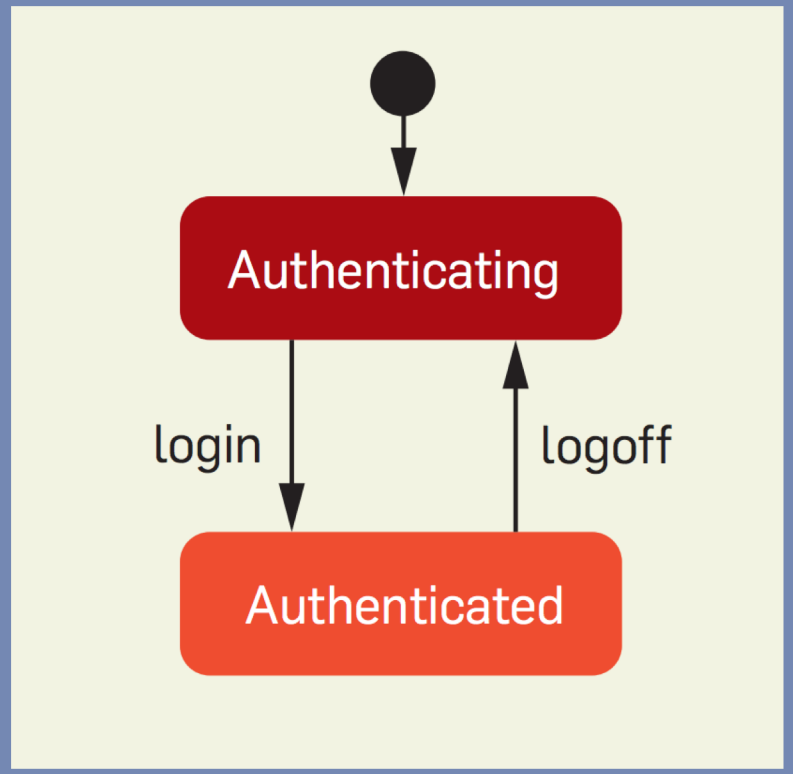
Daedalus Acceptance Tests (Cardano crypto currency wallet)

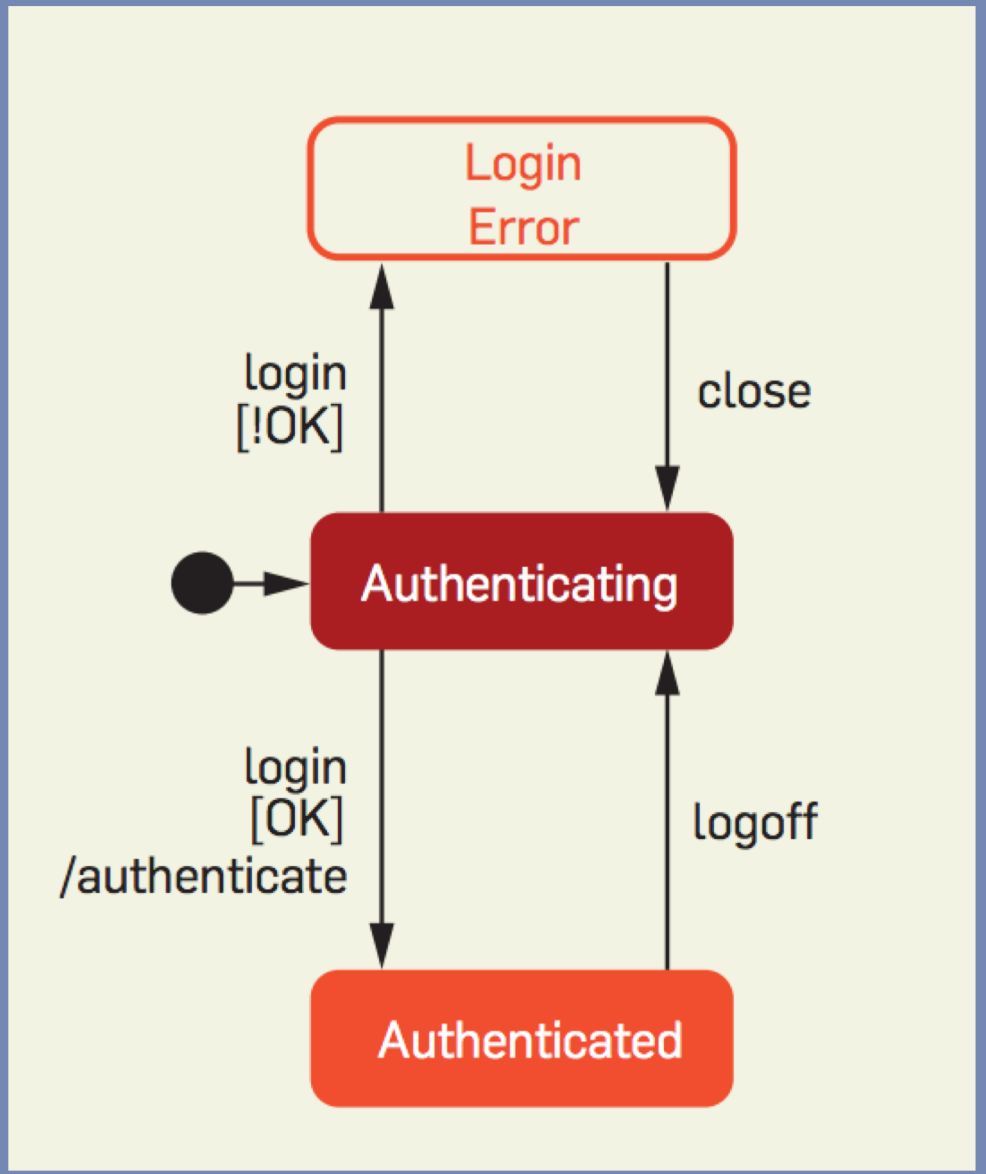


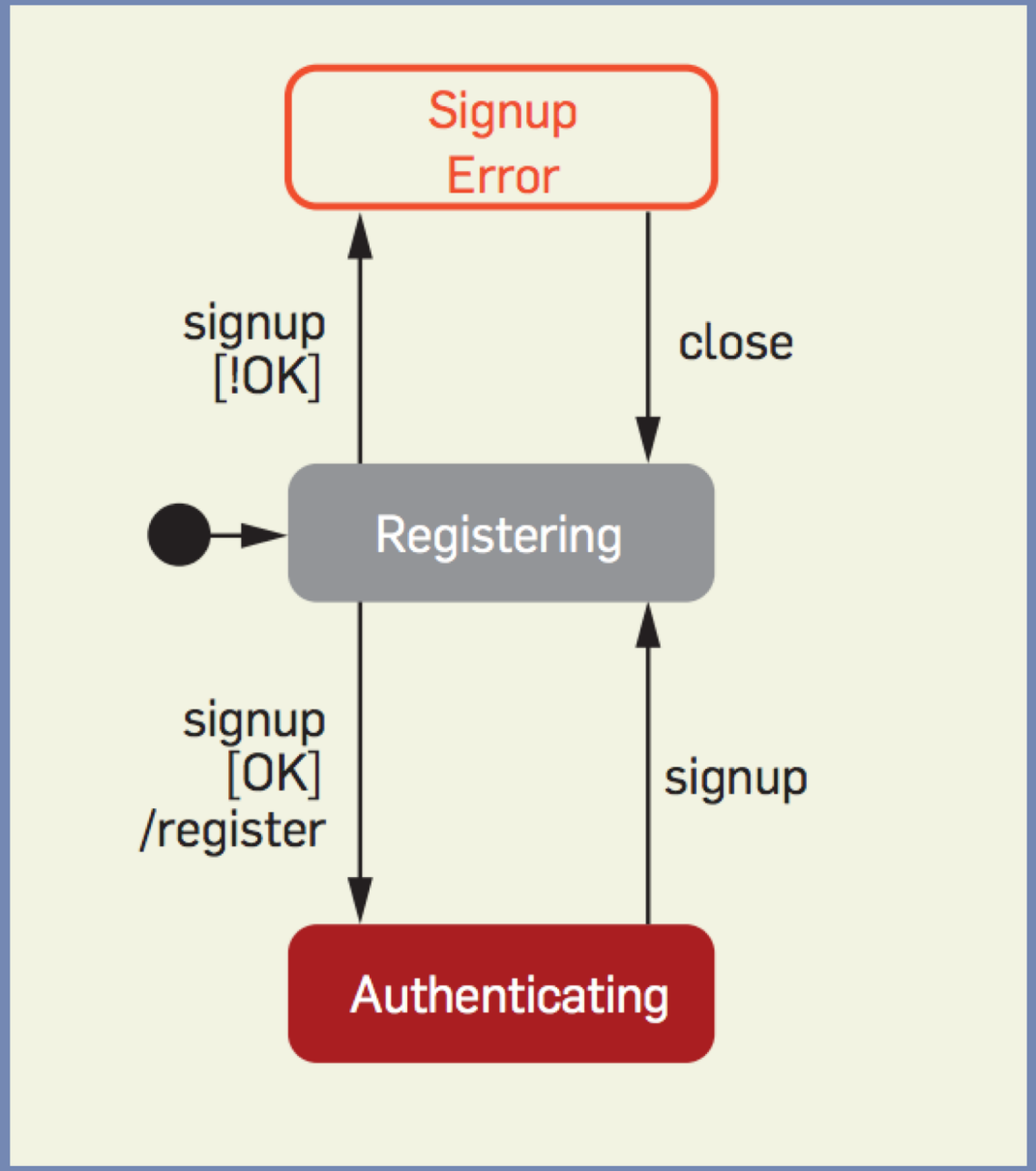
https://www.youtube.com/watch?v=xwLqM0gJ62M

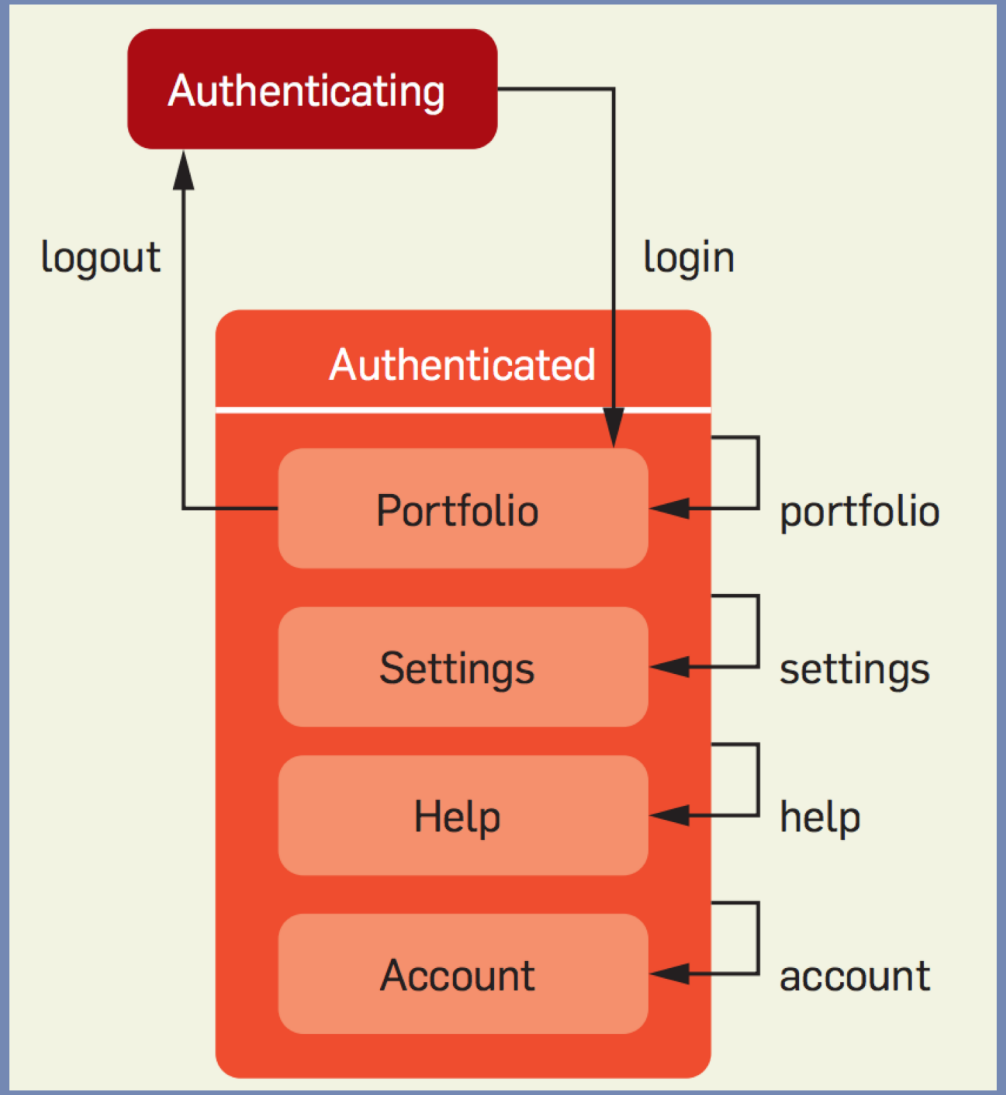
Abstractions

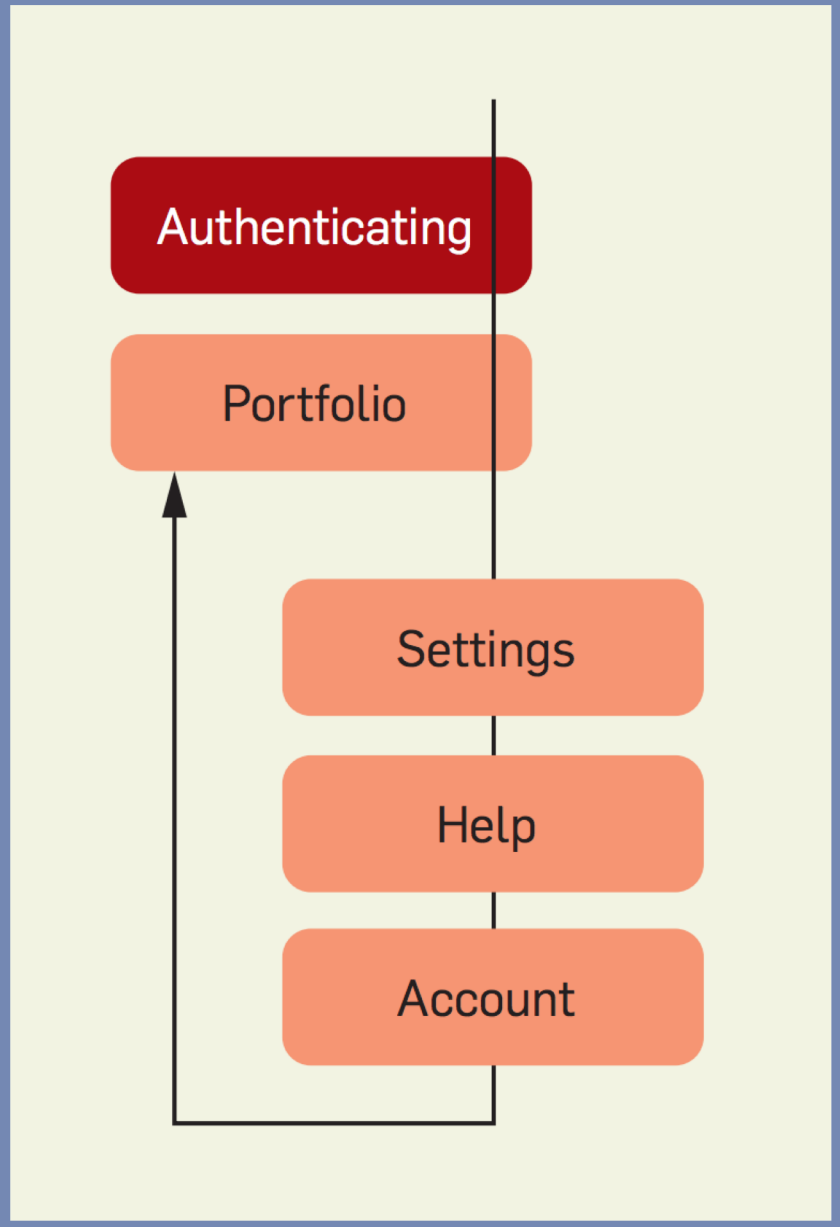
- System’s interface may be at wrong level of abstraction
- Web app: ”webdriver” provides access in terms of clicks and html elements
- Wanted: access in terms of domain concepts.

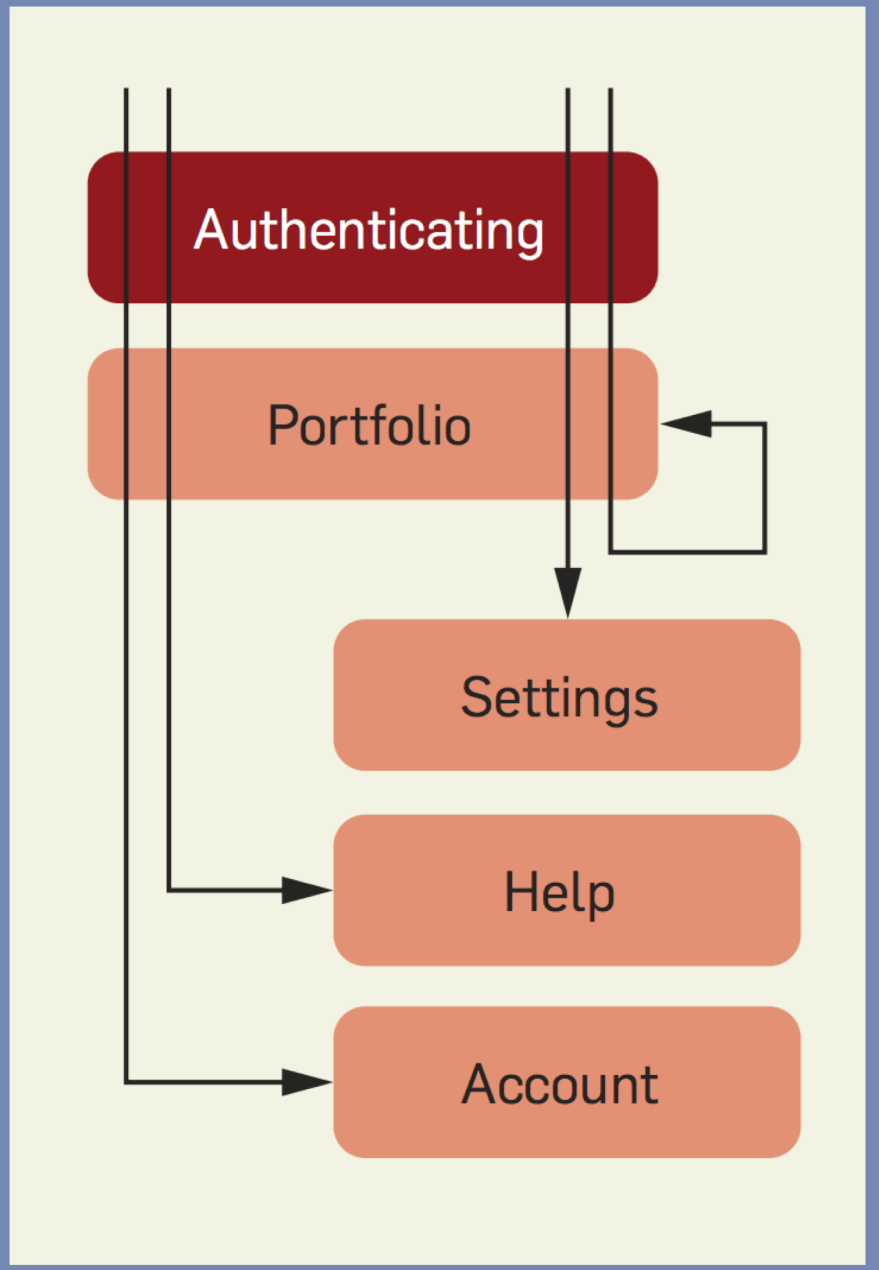

Authenticating
login
logoff
Authenticated

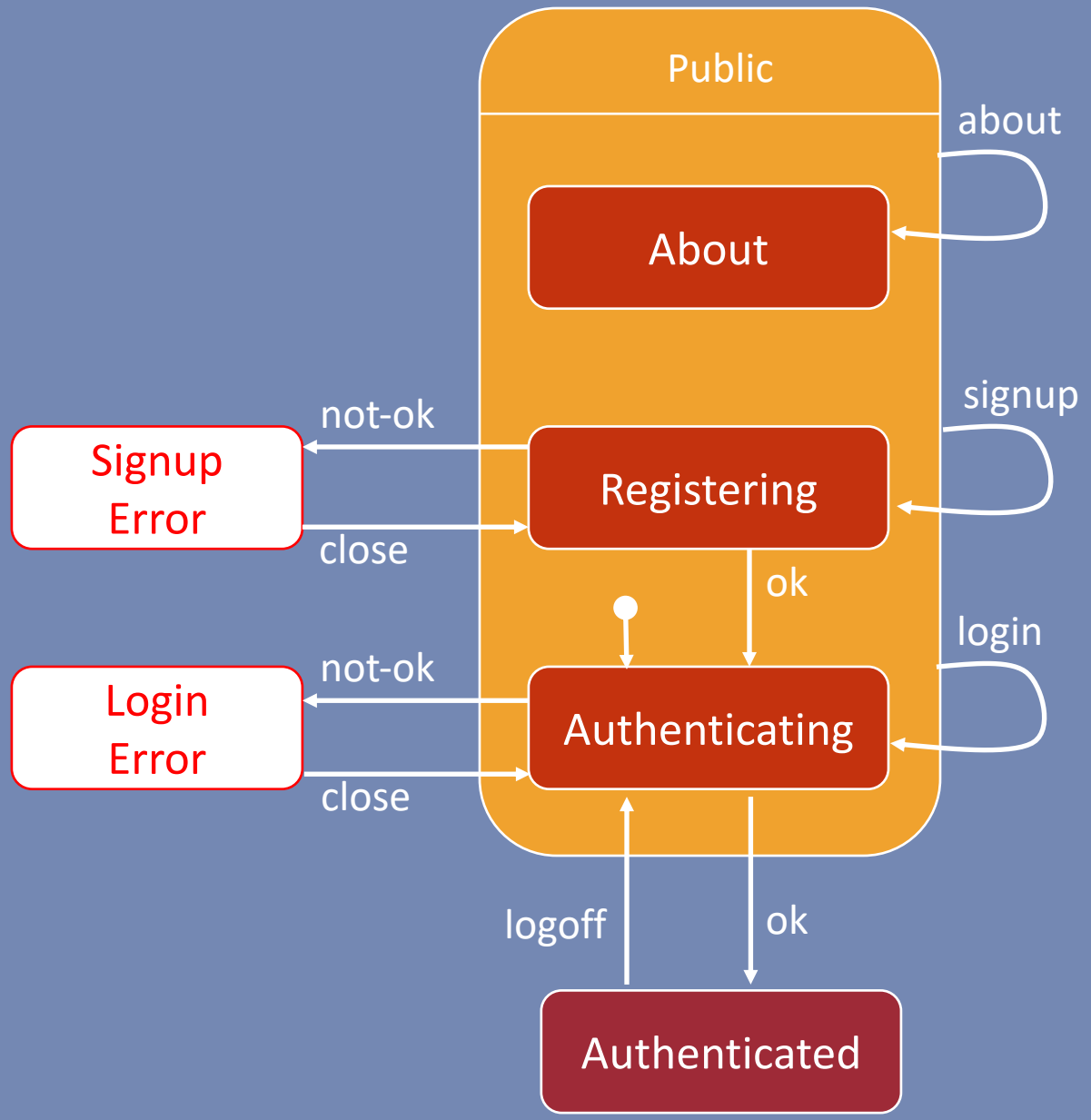

Login
Error
login
[!OK]
close
Authenticating
login
[OK]
/authenticate
logoff
Authenticated

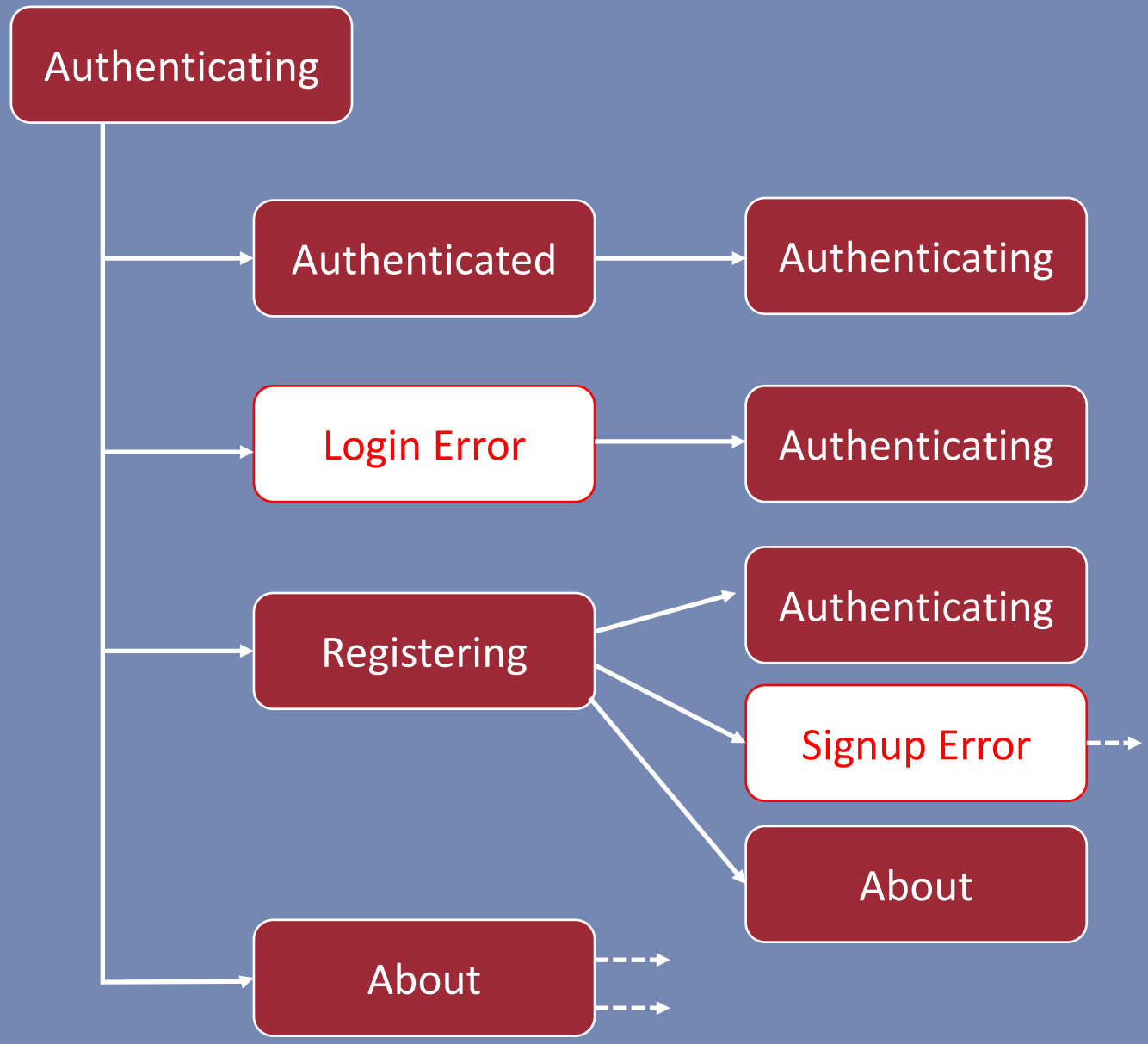

Signup
Error
signup
[!OK]
close
Registering
signup
[OK]
/register
signup
Authenticating

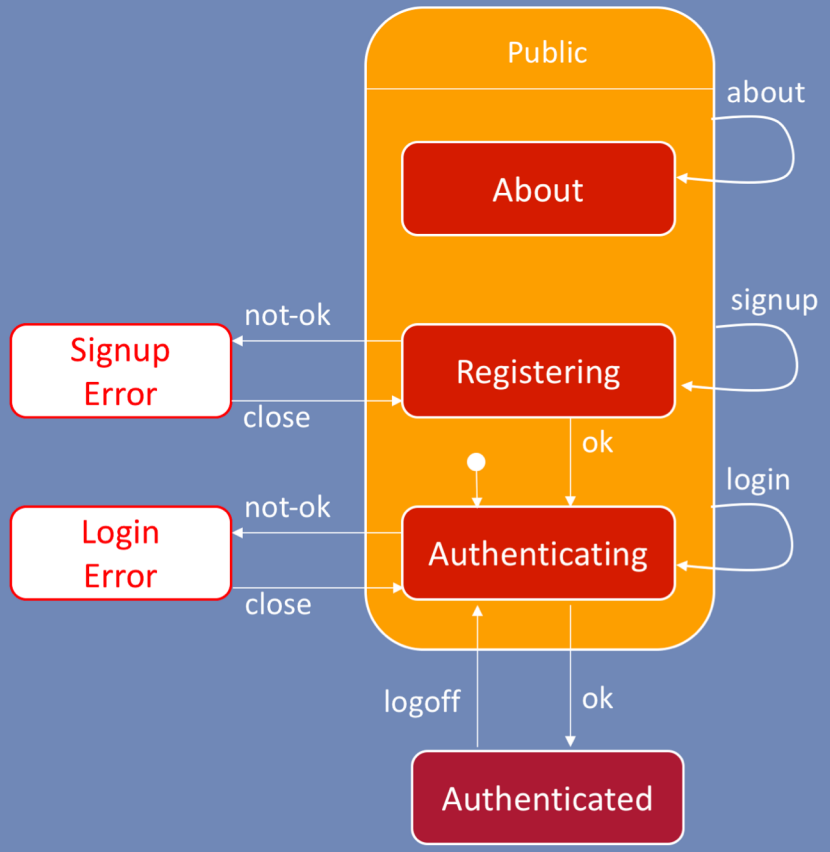

Authenticating
logout
login
Authenticated
Portfolio
Settings
Help
Account
portfolio
settings
help
account


Authenticating
Portfolio
Settings
Help
Account


Authenticating
Portfolio
Settings
Help
Account


Public
about
About
signup
not-ok
Signup
Error
Registering
close
ok
login
not-ok
Login
Error
Authenticating
close
logoff
ok
Authenticated


Authenticating
Authenticated
Authenticating
Login Error
Authenticating
Registering
Authenticating
Signup Error
About
About



	about	signup	login
ABOUT	ABOUT	REG'ING	AUTH'ING
REG'ING	ABOUT	REG'ING	AUTH'ING
AUTH'ING	ABOUT	REG'ING	AUTH'ING
AUTH'ED			
LOGIN-ERR			
REG-ERR			

	ok	not ok	logoff	close
ABOUT				
REG'ING	AUTH'ING	REG-ERR		
AUTH'ING	AUTH'ED	LOGIN-ERR		
AUTH'ED			AUTH'ING	
LOGIN-ERR				AUTH'ING
REG-ERR				REG'ING

Jeff Atwood

@codinghorror

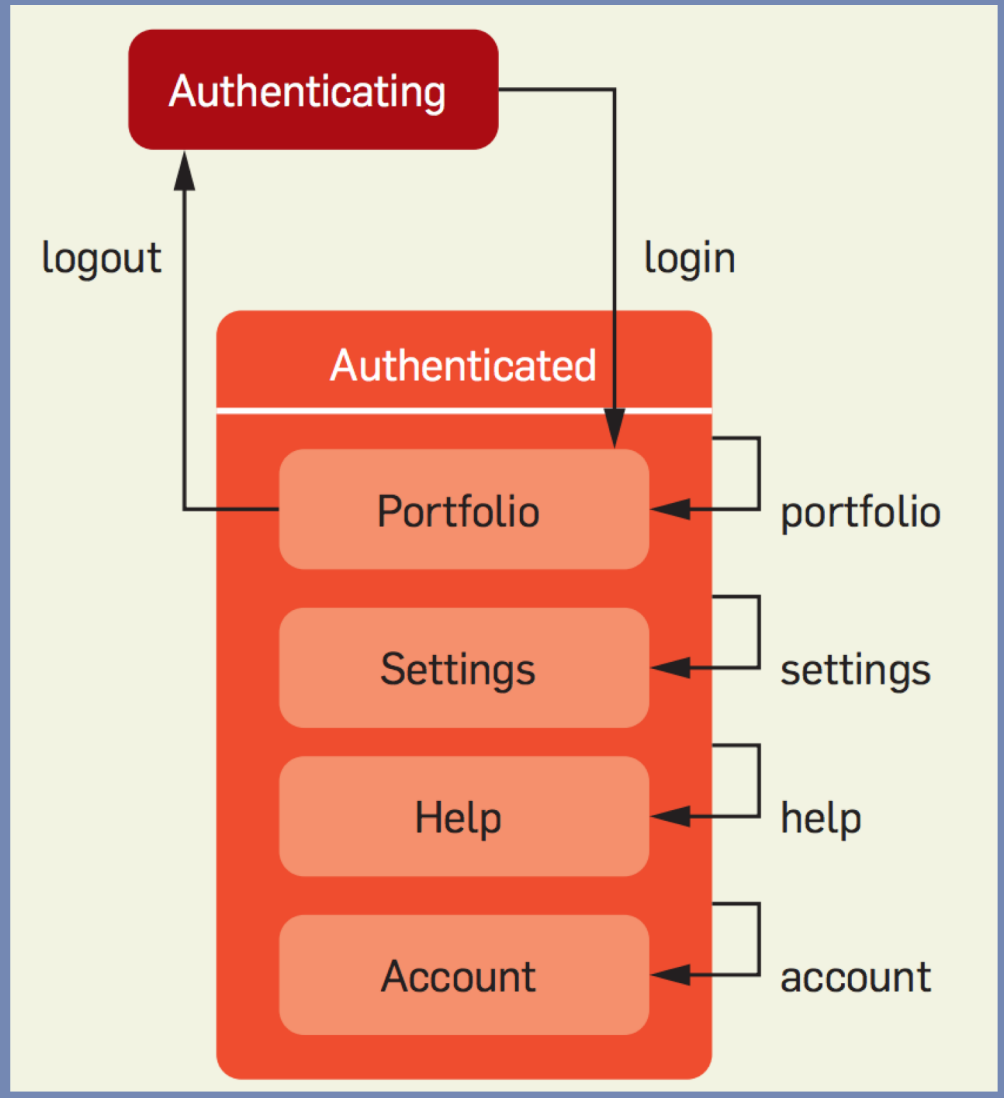
Follow

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.

RETWEETS 2,076

LIKES 2,527

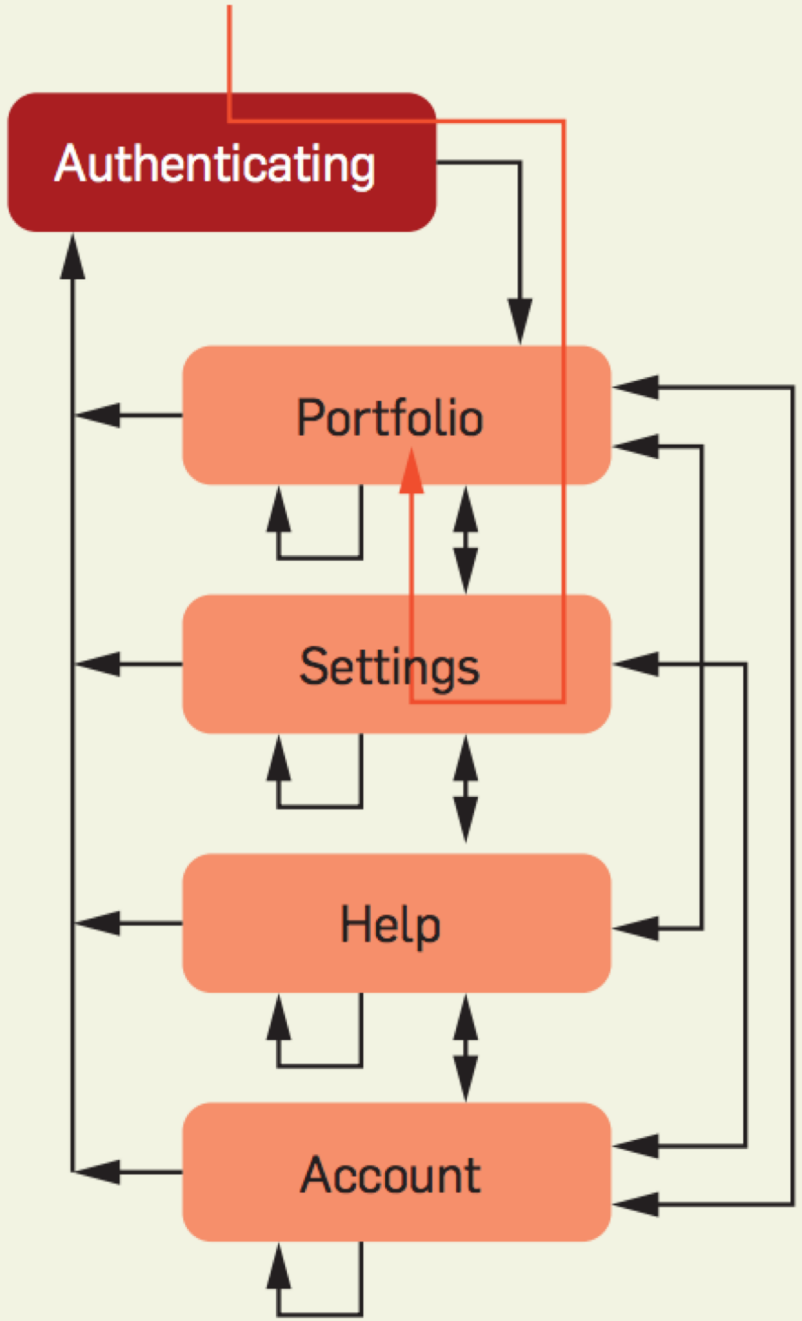
- First time portfolio visit:
  - fetch result and *cache*
- Second portfolio visit:
  - Result cached – no need to fetch!
- Visit settings, and change some
  - *Requires invalidating the cache.*
- Third portfolio visit:
  - *Should fetch again.*



Testing Revisits

- Test scenario needed:

- Authenticating
- 1st porftfolio visit
- Change of settings
- 2nd portfolio visit
  - results should be affected by change of settings



State-Based Software Testing: Recap

State Modeling

- Create small diagrams for behavior of interest
- Abstract using super states
- State reflects past behavior
- State determines future behavior
- State manipulation requires triggering and inspection

State-Based Testing

- Create (UML) state machine
- Turn diagram into tree
- Create *conformance test suite* covering all round trip paths
- Turn diagram into table
- Identify implicit <State, Event> pairs
- Write *sneak path test suite* for missing pairs.