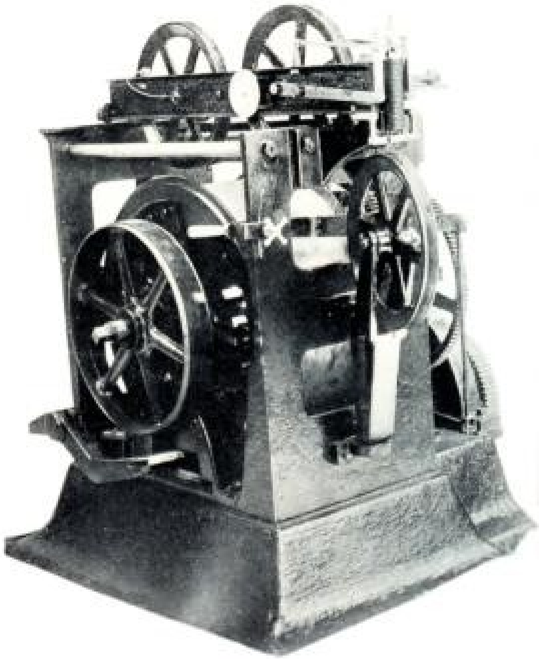
An introduction to

Test Automation



Scope

- Dynamic vs. static
- Functional vs. non-functional
- Acceptance vs. unit (vs. module vs. integration)
- Frameworks vs. drivers
- Running tests vs. generating tests
- Full scale automation vs. helping manual testing
- Test execution vs. test management

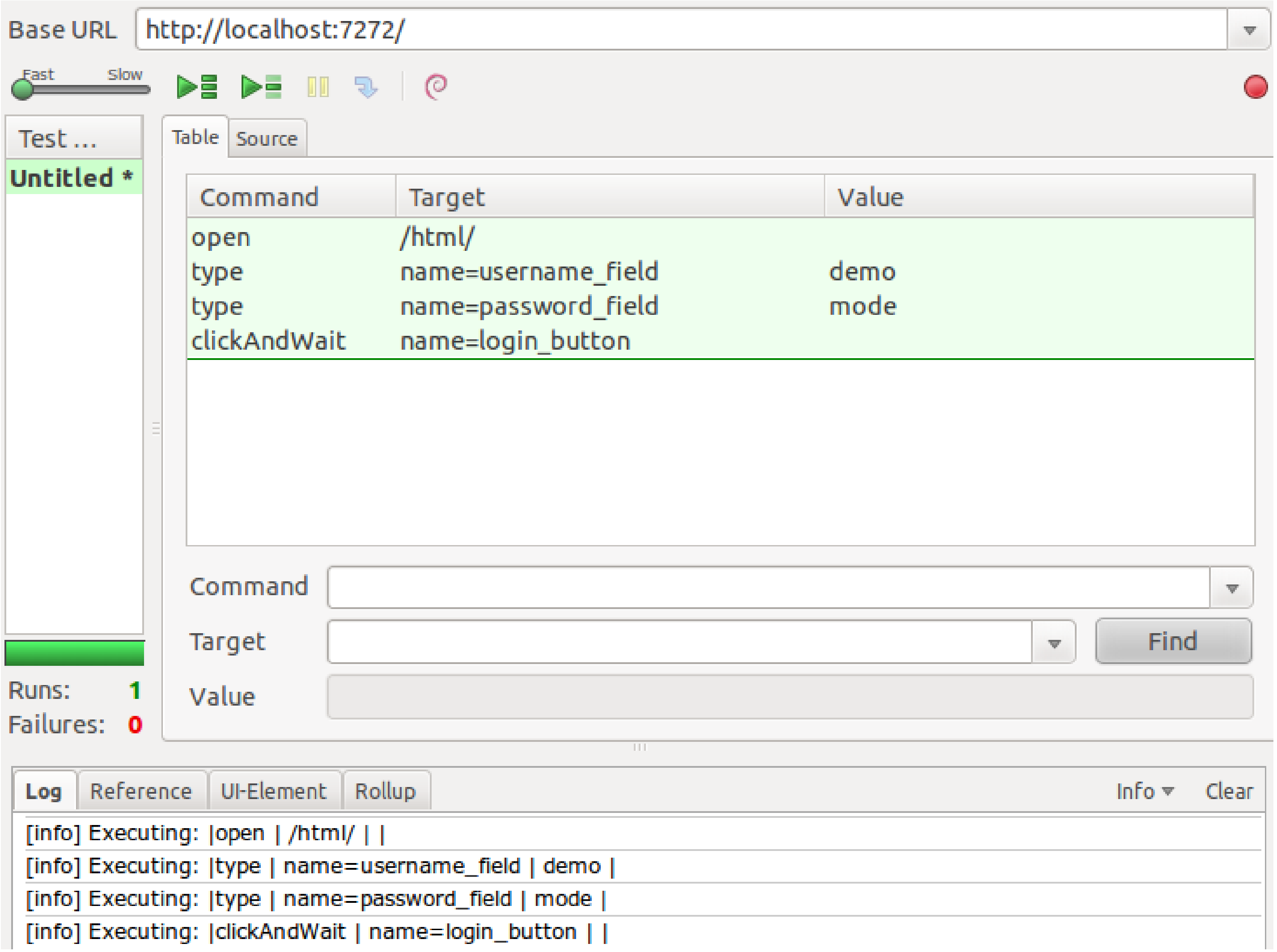
Different scripting approaches

- Record and playback
- Linear scripting
- Modular scripting
- Data-driven testing
- Keyword-driven testing

Record and playback

- Capture interaction with system and replay it
- Popular approach among commercial tools

Record and playback: Example



Base URL http://localhost:7272/

Command	Target	Value
open	/html/	
type	name=username_field	demo
type	name=password_field	mode
clickAndWait	name=login_button	

```
[info] Executing: |open | /html/ | |
[info] Executing: |type | name=username_field | demo |
[info] Executing: |type | name=password_field | mode |
[info] Executing: |clickAndWait | name=login_button | |
```

Selenium IDE

Record and playback: Benefits

- Very easy and fast to create initially
- No programming skills needed

Record and playback: Problems

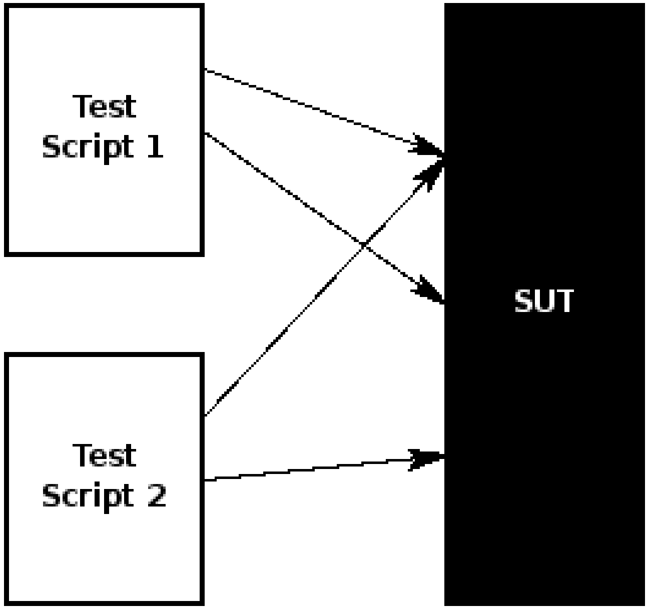
- Does not *test* anything unless checkpoints added
- Very fragile
  - Often single change in UI can break all tests
- Hard to maintain
  - Plenty of separate test scripts
  - No modularity or reuse
- System must be ready before automation can start
  - Does not support acceptance test driven development (ATDD)

Record and playback: Summary

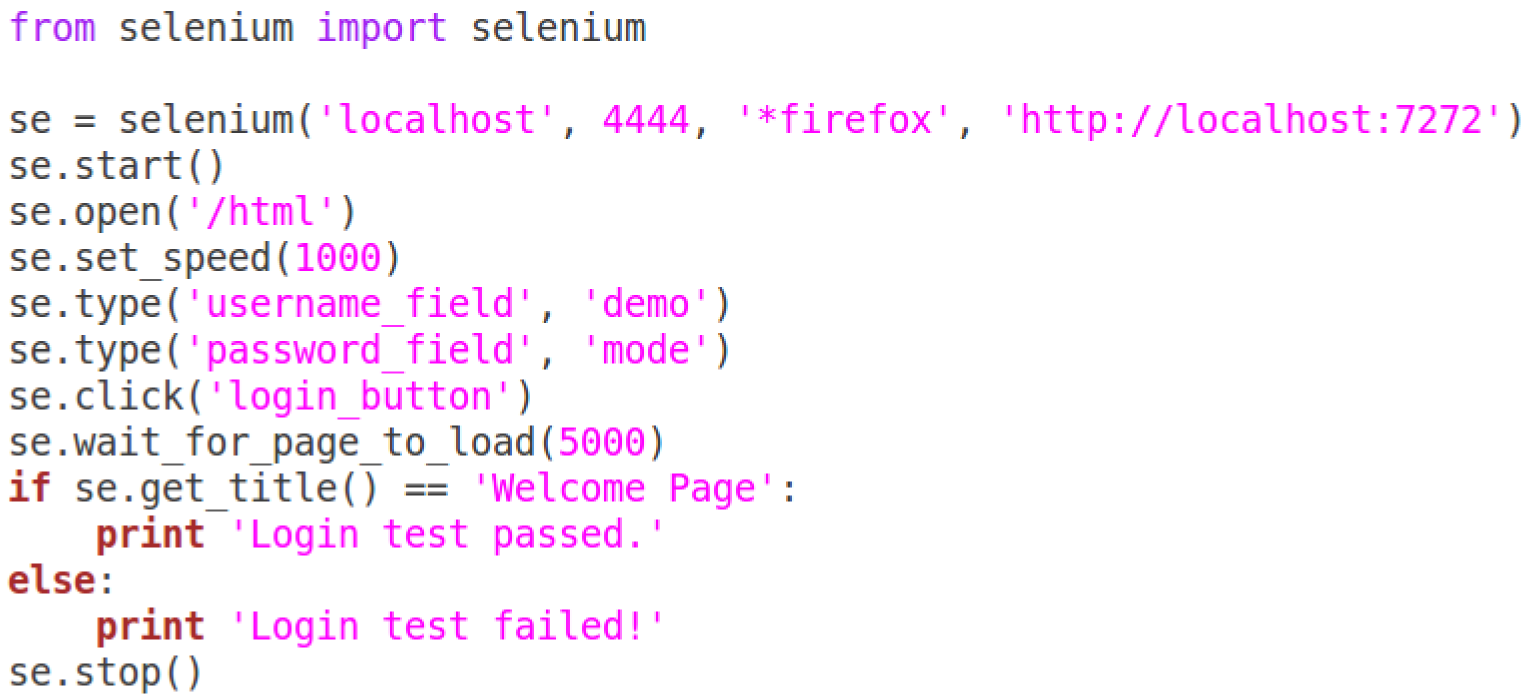
- Seldom a good approach in general
- Never a good basis for large scale automation

Linear scripting

- Non-structured scripts interact directly with the system under test (SUT)
- Can use any programming language
- Also produced by capture and replay tools



Linear scripting: Example



```python
from selenium import selenium

se = selenium('localhost', 4444, '*firefox', 'http://localhost:7272')
se.start()
se.open('/html')
se.set_speed(1000)
se.type('username_field', 'demo')
se.type('password_field', 'mode')
se.click('login_button')
se.wait_for_page_to_load(5000)
if se.get_title() == 'Welcome Page':
    print 'Login test passed.'
else:
    print 'Login test failed!'
se.stop()
```

Selenium RC Python API

Linear scripting: Benefits

- Fast to create initially
- Flexible
- Can use common scripting languages
  - No license costs

Linear scripting: Problems

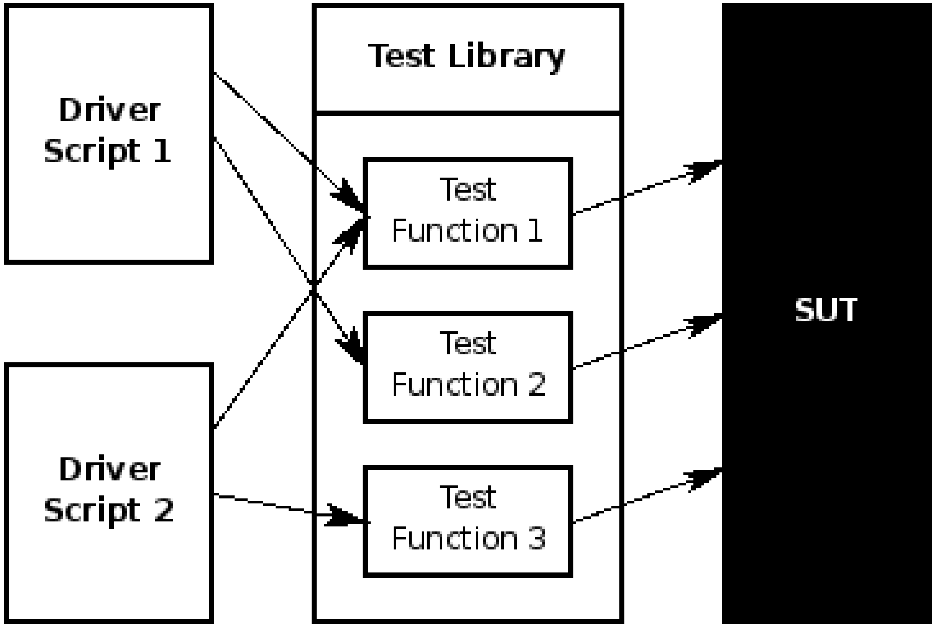
- Creating tests requires programming skills
- Very fragile
  - One change in the system may break all scripts
- Hard to maintain
  - Plenty of test scripts
  - No modularity or reuse

Linear scripting: Summary

- Adequate for simple tasks
- Never a good basis for large scale automation

Modular scripting

- Driver scripts “drive” test execution
- Interaction with the SUT done by functions in a test library



Modular scripting: Example

← Test library

↓ Driver script

```
from selenium import selenium
from urlparse import urlsplit

class Browser(object):

    def __init__(self, url, browser='*firefox'):
        base, path = self._split_url(url)
        self.selenium = selenium('localhost', 4444, browser, base)
        self.selenium.start()
        self.selenium.window_maximize()
        self.selenium.set_speed(1000)
        self.selenium.open(path)

    def _split_url(self, url):
        tokens = urlsplit(url)
        return '://'.join(tokens[:2]), ''.join(tokens[2:])

    def input_username(self, username):
        self.selenium.type('username_field', username)

    def input_password(self, password):
        self.selenium.type('password_field', password)

    def click_login_button(self):
        self.selenium.click('login_button')
        self.selenium.wait_for_page_to_load(5000)

    def verify_title(self, expected):
        title = self.selenium.get_title()
        if title != expected:
            raise AssertionError("Expected title to be '%s' but it was '%s'"
                                 % (title, expected))

    def close(self):
        self.selenium.stop()
```

```
from seleniumlibrary import Browser

browser = Browser('http://localhost:7272/html')
browser.input_username('demo')
browser.input_password('mode')
browser.click_login_button()
try:
    browser.verify_title('Welcome Page')
except AssertionError, err:
    print 'Login test failed:', err
else:
    print 'Login test passed.'
finally:
    browser.close()
```

Modular scripting: Benefits

- Reuse of code
  - Creating new tests gets faster
- Maintainability
  - Changes require fixes in smaller areas
- Driver scripts are simple
  - Even novice programmers can understand and edit
  - Creating new ones is not hard either

Modular scripting: Problems

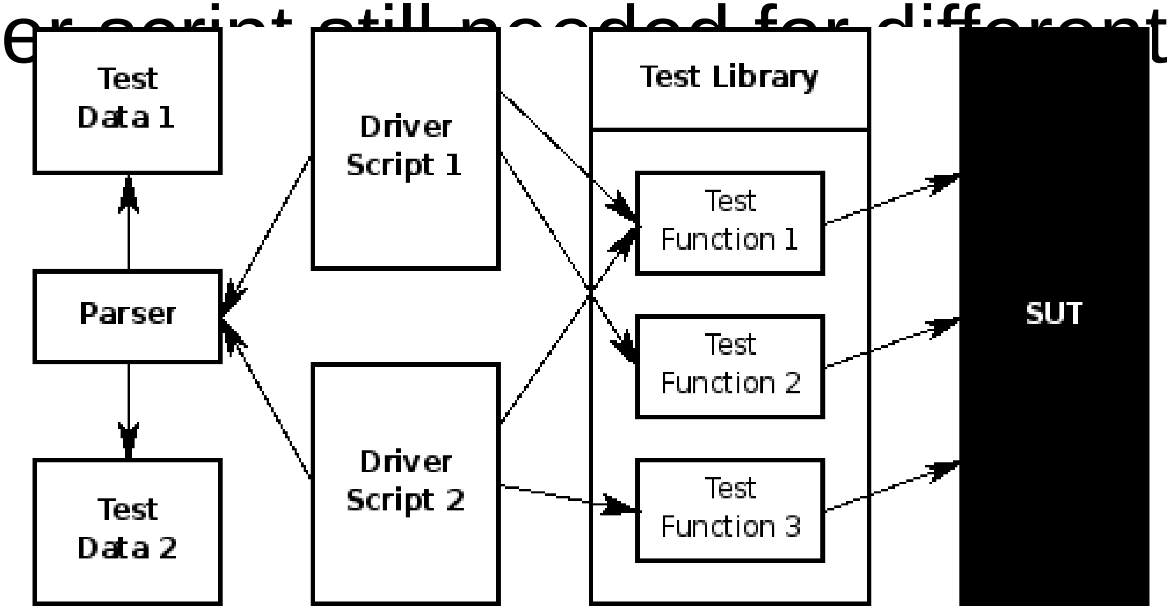
- Building test library requires initial effort
  - Takes time
  - Requires programming skills
- Test data embedded into scripts
  - Requires some understanding of programming
- New tests require new driver scripts

Modular scripting: Summary

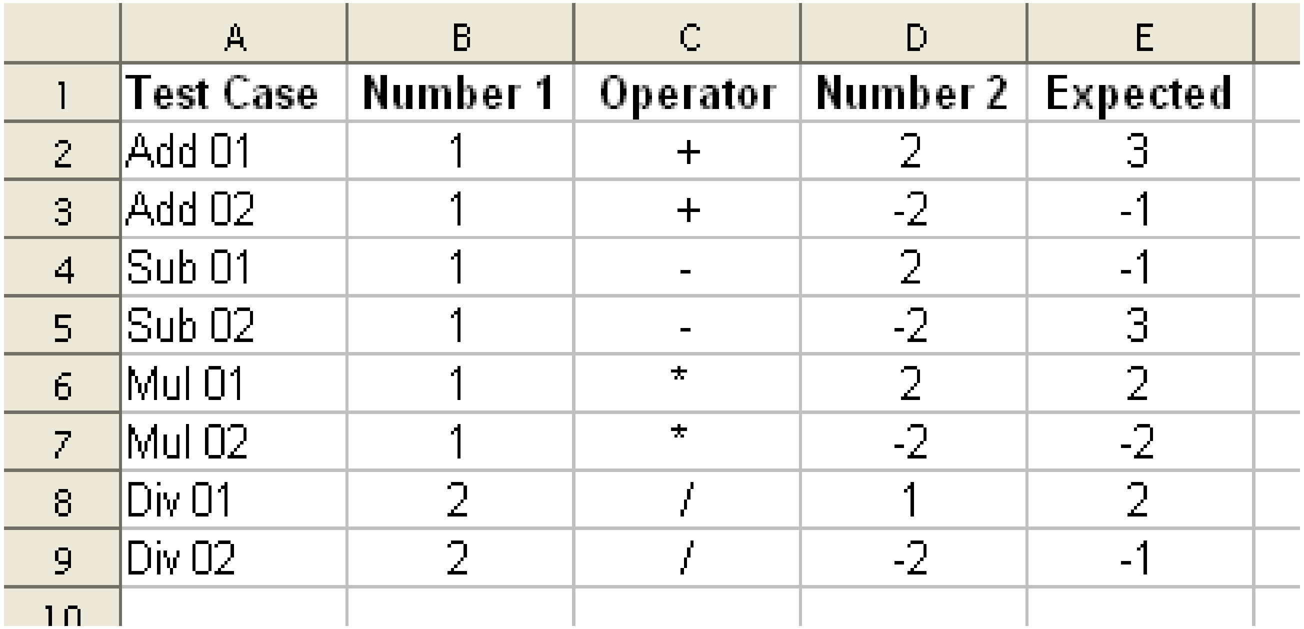
- Good for simple tasks
- Works also in larger usage *if* everyone who needs to understand tests can program
- Not good for non-programmers

Data-driven testing

- Test data taken out of test scripts
  - Customarily presented in tabular format
- One driver script can execute multiple similar tests
- New driver script still needed for different kinds of tests



Data-driven testing: Example



Test Case	Number 1	Operator	Number 2	Expected
Add 01	1	+	2	3
Add 02	1	+	-2	-1
Sub 01	1	-	2	-1
Sub 02	1	-	-2	3
Mul 01	1	*	2	2
Mul 02	1	*	-2	-2
Div 01	2	/	1	2
Div 02	2	/	-2	-1

Data-driven testing: Benefits

- Test libraries provide modularity
  - Same benefits as with modular scripting
- Creating and editing existing tests is very easy
  - No programming skills needed
- Maintenance responsibilities can be divided
  - Testers are responsible for the test data
  - Programmers are responsible for automation code

Data-driven testing: Problems

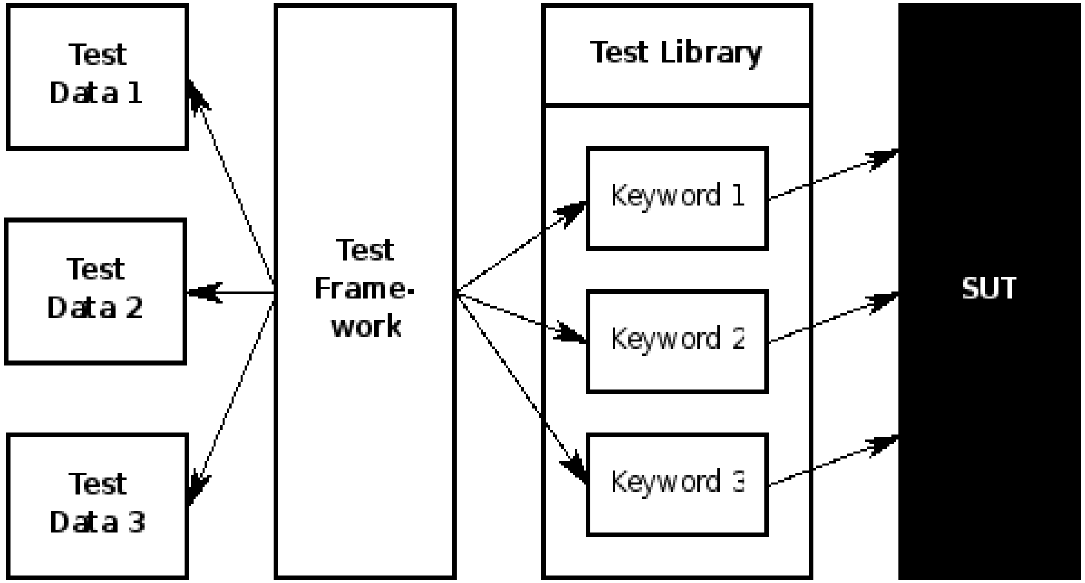
- Test cases are similar
  - For example '1 + 2 = 3' and '1 * 2 = 2'
- New kinds of tests need new driver script
  - For example '1 * 2 + 3 = 6'
  - Creating driver scripts requires programming skills
- Initial effort creating parsers and other reusable components can be big

Data-driven testing: Summary

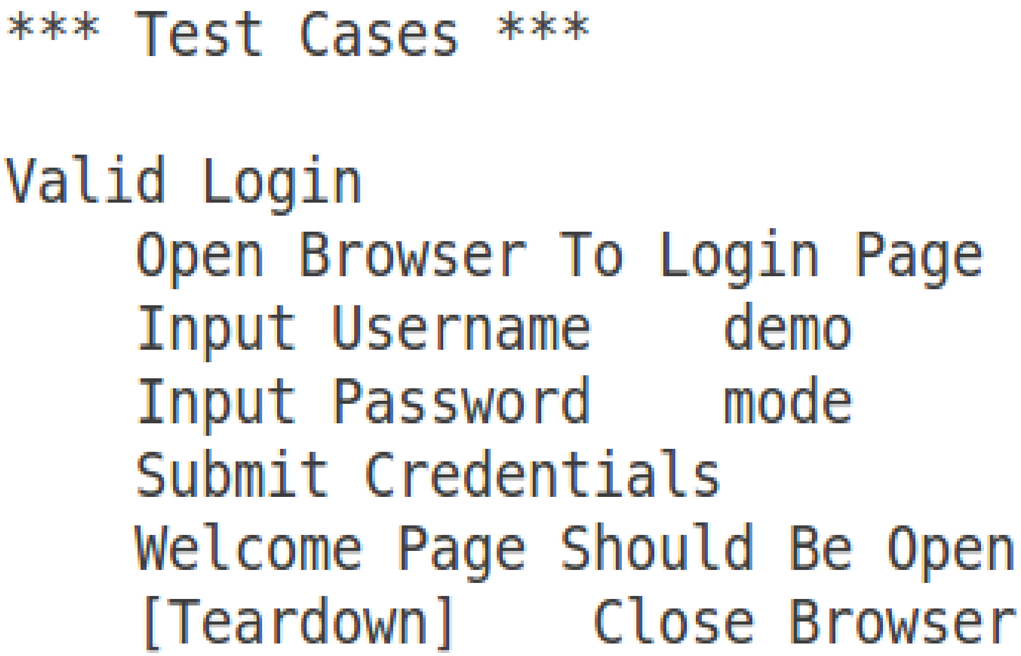
- Good solution even for larger scale use
- New kinds of tests requiring programming is a minus
- May be an overkill for simple needs

Keyword-driven testing

- Not only test data but also directives (keywords) telling how to use the data taken out of the test scripts
- Keywords and the test data associated with them drive test execution



Keyword-driven testing: Example



```
*** Test Cases ***

Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Robot Framework syntax from SeleniumLibrary demo:
http://bit.ly/rf-web-test-demo

Keyword-driven testing: Benefits

- All same benefits as with data-driven testing
  - Non-programmers can create and edit tests
  - Separation of test data and code
- Tests can be constructed freely from keywords
  - Non-programmers can create also *new kinds* of tests
  - With suitable keywords also data-driven tests possible
- All tests can be executed by one framework
  - No need to create and maintain separate driver scripts

Keyword-driven testing: Problems

- Initial effort can be really big
  - But there are open source solutions available!

Keyword-driven testing: Summary

- Very good solution for large scale use
- Use existing solutions if you can
- May be an overkill for simple needs

Interacting with the SUT

- Testability
- Testing through GUI
- Testing below GUI
- Other interfaces

Testability

- The hardest part of automation is interacting with the system under test
  - Especially hard with GUIs
  - Programming APIs are easiest
- Important to make the system easy to test
- Some common guidelines
  - Add identifiers to GUI widgets
  - Textual outputs should be easy to parse
  - Consider providing automation interfaces

Testing through GUI

- Same interface as normal users use
- Can be technically challenging or impossible
  - Not all GUI technologies have good tools available
- Often fragile tests
- Often relative slow to execute
- Good approach to use when feasible

Testing below GUI

- Automating through business layer often easy
- Tests typically run very fast
- But you still need to test the GUI
    - Test the GUI is wired correctly to the business logic
    - GUIs always have some functionality of their own
- Pragmatic hybrid solution:
    - Test overall functionality below the GUI
    - Some end-to-end tests through the GUI—not necessarily even automated

Other interfaces

- Not all systems have a GUI
- Many systems have multiple interfaces
  - Programming APIs, databases, server interfaces, command line, …
  - Automation framework which can utilize different drivers works well in these situations
- Non-GUI interfaces generally easy to automate
  - Most of them targeted for machines
  - Test library is just another client

When to automate and by whom?

- After development by separate team
- During development collaboratively

Automation after development

- Often by different team
  - In worst case on a different floor, building, or continent
  - Communication problems
- Typical in waterfall-ish projects
- Slow feedback loop
- Testability problems can be show stoppers
  - Often hard to get testability hooks added afterwards
  - May need to resort to complicated and fragile solutions

Collaborative automation

- Automation considered an integral part of development
  - Collaboration between testers and programmers
- Typical in Agile projects
  - In acceptance test driven development (ATDD) automation started before implementation
- Testability normally not a problem
  - Programmers can create testability hooks
  - Testability and available tooling can be taken into account even with technology decisions

Supporting practices

- Version control
- Continuous integration

Version control

- Test data and code should be stored the same way as production code
- Recommended to store tests with the production code
  - Easy to get an old version of the software with related tests
- Lot of great open source alternatives available
  - Subversion, Git, Mercurial, …
  - No excuse not to use

Continuous integration

- Key to full scale automation
- Tests are run automatically when
  - New tests are added
  - Code is changed
- Can also have scheduled test runs
  - Useful if running all tests takes time
- Great open source solutions available
  - Jenkins/Hudson, Cruise Control, BuildBot, …
  - Custom scripts and cron jobs can be retired

Available tools

- Commercial
- Open source
- Freeware

Commercial tools

- Good ones tend to be expensive
  - But not all expensive are good
  - Even cheap licenses can prevent full team collaboration
- Often hard to integrate with
  - Other automation tools (especially from other vendors)
  - Version control and continuous integration
- Hard or impossible to customize
- Risk of product or company discontinuation

Open source tools

- Large variety
  - Some are great—others not so
- Normally easy to integrate with other tools
- Free, as in beer, is good
  - Everyone can use freely
- Free, as in speech, is good
  - Can be customize freely
  - Can never really die

Freeware tools

- Getting rare nowadays
  - Most free tools are also open source
- No license costs
- Tend to be easier to integrate with other tools than commercial
- Hard or impossible to customize
- Risk of discontinuation

Generic skills to learn

- Scripting languages
  - Python, Ruby, Perl, Javascript, ...
- Regular expressions
  - A must when parsing textual outputs
- XPath and CSS selectors
  - A must when doing web testing
- SQL
  - A must with databases
- Using version control

Is manual testing still needed?

- YES!!
- Avoid scripted manual testing
  - Automate it instead
- Concentrate on exploratory testing
  - Machines are great for running regression tests
  - Humans are great for finding *new* defects

Questions?
Thanks!