Scripting Recipes for Testers - Draft 2

Adam Goucher
http://adam.goucher.ca
adam@goucher.ca

It used to be that testers could go through their entire career without getting their hands dirty in code. Those days are forever behind us. Now, if you want to achieve the most our of your career learning how to program is quickly becoming a mandatory skill. Being able to think critically will always be more important, but being about to both read and produce code can accelerate and extend the reach of that critical thinking.

Testers rarely are required to produce production code, but we are often called upon to create tools to assist in our investigations of product quality and increase the efficiency of those investigations. These efficiencies and improvements often take the form of

- *Improved Insight* - There is all sorts of useful quality information hidden inside the actual source code of your application.
- *Increase Ease* - Certain tasks that are difficult for a person to complete might be made easier when done by the machine.
- *Test Case Selection* - How much to test with which browser is a common problem testers face. By analyzing the actual server's logs you can get an accurate idea of your real browser distribution.
- *Test Data Generation* - Most application input needs to meet certain rules; scripts are an easy way to generate rule adhering input.
- *Oracles* - When your testing oracle is algorithmic in nature, it is often easier to have the verification contained in a script than to figure it out each iteration.
- *Reduce Costs* - In some situations, like web automation, a scripting language coupled with open source frameworks can completely replace expensive commercial testing applications.
- *Increase Speed* - Computers are really good at doing repetitive tasks fast

Which Language?

There are hundreds, if not thousands of languages a tester could choose from when creating tools for themselves and those on their team. Before getting into specific languages the distinction between a scripting language and a non-scripting language needs to be defined.

A language is considered a scripting language if it is an interpreted one. That is, there is no intermediate compilation phase between when your have written your code and it can be executed. Non-scripting languages like C and Java all have this intermediate step which decreases your scripting velocity. For the tasks tester scripts will often be used for, time-to-use is often more important than raw execution speed. Speed of execution is part of the opportunity cost you pay when using an interpreted language.

Which one you choose will depend entirely upon your product, your skill set and company culture. Here are some questions to consider when making this selection.

- Do you know the language already?
- Could you learn it?
- Is there a local mentor or collaborator?
- Is it supported across all your platforms?
- Are all those platforms considered first-class citizens?
- Can it interact with your database?
- Can it control your other tools? (Selenium, Watir, etc.)
- Can it interact with other components of your applications (by importing Java jars or .NET assemblies)?
- Does it support Unicode as a code data type?
- Is there seamless interaction with the operating system?
- Is there an active member community (to whom you can turn to for help)

Right now there are 3 languages that are mature enough to be considered for adoption in today's testing teams; Python, Ruby and Perl. You will notice that VBScript is not on the short list. VBScript knowledge has risen in profile in the testing community due to it being the language HP QuickTest Pro scripts are written in. Unfortunately, VBScript is a Windows-only technology which means that to have to re-implement scripts that need to run on non-Windows platforms. It is more efficient to just write the scripts in a portable language originally.

If you are just starting out scripting, I would recommend Python. Its enforced style conventions and syntax greatly reduce the barrier to entry. It also has a large number of built-in libraries for interacting with common systems a tester interacts with.

Perl certainly has the greatest install base of these three, but the readability and maintainability of scripts written in it has historically been a source of pain to either other authors of a script or even to a future version of yourself. Ruby is quite rapidly becoming a viable option as well for a day-to-day scripting language. The primary issue holding it back was its lack of native support for Unicode, but the newest version has support for that and many other text encodings as well.

Another item to keep in mind is that the language you choose does not have to be the one that the application you are testing is written in. In fact, in most situation it won't be unless you are using something like Ruby on Rails or Django.

The decision of which language to choose is ultimately dependent on you and the contexts you are currently in, but any of the three mentioned would not be a bad one.

Improve Insight

There is an adage I have heard on more than one occasion from the developers I am working with: *The code doesn't lie*. Since the code doesn't lie, it is a fantastic resource

for quality information. In this context the information is not style or pattern related as those can be found using the standard style checking and static analysis tools of your application's language. The information we are looking for is often embedded in comments of selection of specific variable names. Using a custom script you can extract this often hidden information.

When I first get a code base I am most interested in the presence, quantity and content of what I refer to as *developer notes*. These are breadcrumbs that the developers has left for themselves and are typically split into two categories

- TODO - These notes are placeholders for future or uncompleted work. Unfortunately, schedule pressures often lead to the developer not going back to finish the work outlined. Suddenly when a task is identified as 'done', it really isn't due to the presence of one or more TODOs. Put another way, when I am doing the dishes and I still have TODO the pots, I am not done. Regardless of whether or not I put a sticky not on the spaghetti pot.
- FIXME - The notes are indicate the presence of either a bug, or an area of code that someone feels needs to be refactored.

Both of these are examples of embedded information which increase the technical debt of our application, but does it in a silent manner. Conceptually, I have no problem with either of these notes being produced as they each serve a valuable purpose; I just want to to be done in an open, transparent manner.

One technique to ensure that the notes do not go unrecorded is to integrate checks into your version control system. Rather than have TODO or FIXME alone, a syntax of TODO - n where n is a ticket number in the bug tracking system would allow the commit to proceed. A TODO without a ticket number would be be rejected.

Integrating with version control only addresses future occurrences of these notes, and not the ones that are already there. For that we need a script. The python script, DeveloperNotes.py (Listing 1 and http://adam.goucher.ca/?p=507) is designed specifically to locate these notes. Consider it a cross-platform *grep* without all the arcane command-line switches.

To tailor it to your environment, you need only modify three variables:

- *interesting* - Put the extension(s) of the files you want to scan here
- *notes* - These are the actual strings that the script will be looking for
- *source_base* - This is the root of your source code tree

One other note about tailoring this script is that what you are scanning for will often change over time and depending on who is on the development team. I once had a developer which put 'Danger, Will Robinson' around code he thought was dangerously complicated; Robinson was added to the notes list.

This script also illustrates a couple scripting tips

Tip - Common Case - Notice how all the values are are looking for are lowercase? That is because be are turning all the content we are scanning in to lowercase as well. This means that 'todo' will find 'TODO' or 'ToDo' or any other permutation of case.

Tip - YAGNI - YAGNI stands for You Ain't Gonna Need It which means that you do just enough to get the job done. DeveloperNotes.py could be modified to use threads for parallel processing or regular expressions to look find things of interest, but it gets the job done (for me) as it is currently. The time spent modifying the script further would have to be taken from some other task.

Also related to the idea of gaining new insight into a product as a result of scripting it is the reality that as we script up the system we learn more about how it is actually built. I have successfully used script creation to understand how objects within the system are created, related and destroyed. These exploratory scripts are almost throwaway as their purpose is not automation, but informative.

Increase Ease

There are a number of tasks that testers need to complete that are, on the surface, quite difficult to accomplish. One of those tasks is verification in the UI layer that all strings are available for translation. With a bit of scripting and the use of a technique called LOUD, this is as simple as clicking through all your screens.

The LOUD technique does is modify the strings that are displayed on the screen so they have a ^ at the beginning, a $ at the end and is all caps. (It is the all caps that lends LOUD its name after the netiquette convention that capitalization implies shouting). Now you do not need to actually use translated content to see whether the string are all externalized or if they are embedded in the code itself. When you are viewing the screens you are looking for the following

- All content on the screen has a ^ at the beginning
- All content on the screen has a $ at the end
- All content on the screen is capitalized, except things that should not be translated like trademarked product names
- Sentences should not be made of multiple LOUD fragments. By dynamically creating sentences you make it hard for the translators to properly understand the context of what they are translating which increased the chance of a mistranslation. Ideally you want to have entire paragraphs as single LOUD blocks

Most modern languages have a means of putting user visible string data in external, language-specific files. LOUD.py (Listing 2 and http://adam.goucher.ca/?p=510) is a sample Java resource bundle conversion script to enable LOUD testing. In a non-

example context, it would be reading an external file as input rather than use an in-memory string.

Tip - Processing Pattern - A common pattern you will utilize when dealing with files is

- Open the file
- Process it one line at a time
- Do the manipulation in as atomic operations as possible. (Just because you could do it all on one line does not mean that it is a good, or worthwhile idea.)
- Write the new information to a file

Test Selection

With the proliferation of operating systems, web browsers and even screen resolutions, the possible testing configuration configurations is distressingly large. Some of this number can be reduced through techniques such as Equivalence Class Partitioning, but even that will leave a large number. From there you can either use your gut and make a guess about what to test with or you can actually gather some data.

A very powerful source of data about what types of browsers users use to access your web site is the web server's access logs. Yes, there are companies out there that release monthly browser market share numbers, but those represent an aggregate across the Internet. Your web server logs are explicitly within your context which gives them far greater weight when making these kinds of decisions.

Browsers.rb (Listing 3 and http://adam.goucher.ca/?p=432) is a ruby script which analyzes an Apache style access log tell you which portion of your traffic comes from which browser. Simple change the location of the log it is opening to tailor it to your environment.

Tip - Cleanse data before processing - If you know you have information captured in the log which is going to impact your outcome, such as your office IP address space, do not include it.

Tip - Capture as much information as you can - Just because you do not need the information now, if you can easily extract it now, you should. In this particular script I capture information about a number of the spiders that crawl our site, but I didn't do anything with them at the time.

Tip - Don't be afraid to reinvent the wheel - There a lot of log analyzing tools available, but commercial and open source, but a lot of them over load you with information. If you are interested in specific information from a specific source, sometimes it makes sense to reinvent the wheel

Test Data Generation

Here's the situation. You are working on an application which has financial and credit score information and as such has the constraint of information flowing one direction; in. This means that you cannot recycle account names. So what do you do?

You could use people in your immediate family, then extended family, then on your favourite television shows, then movies. But that gets exhausted fairly fast.You could then drag your hand across the keyboard to input random noise in the fields, but even that is not as random as you might thing. A better idea is to create a script which will produce new accounts for you.

Input data is always rule based. For a first name they might

- Minimum 2 characters
- Maximum 20 characters
- Spaces are permitted
- Hyphens are permitted
- Numbers are permitted
- Periods are permitted
- Anything not specifically permitted is not allowed

According to those rules, the following are all valid

- John Jr.
- Adam
- Bruce 3rd
- Mary-Jane

but so are

- nde2nd woq
- ks
- kfjlskfj-ejfek
- kwlvyg8- k

By teaching your script the rules around your input, you can generate unique data simply and easily. Loyalty.py (Listing 4 and http://adam.goucher.ca/?p=530) includes an example of dynamic test data generation in the *generate_number* function.

Tip - It doesn't matter if you cannot pronounce it - As humans we are comfortable seeing something easily identifiable as a name, and pronounceable in fields labeled as name fields. Computers don't know whether or not the name is real or just random data, all they know is that it *looks* like a name. Keep this in mind when preparing your test data.

Oracles

The opposite of random test data generation is data validation.

In testing, the principle or mechanism by which we recognize a problem is called an oracle, named after the Oracle of Delphi, not the database company. An algorithm that does validation is an oracle. Loyalty.py (Listing 4 and http://adam.goucher.ca/?p=530) also has a function called *verify_number* which is an oracle for the account number that was being tested.

Tip - Put generator and oracle together - Notice how Loyalty.py includes both the test data generator and the oracle in the same file. This is a logical pattern to follow as it keeps all the relevant code in a single file. You can also use the generator to test the oracle and vice-versa. But be careful, sometimes data can be created in an automated, scripted manner, but cannot be verified in such.

Reduce Costs

Even in boom times, there are constantly pressures on budgets across the organization. Commercial automation tools can take up a large portion of your testing budget. And sometimes you have no choice but to shell out the money, but more and more there are open source alternatives available. Where the polish of these tools is lacking is usually the presence of a nice UI to drive them, instead you need to use a scripting language and interact with it that way.

The big thing to keep in mind is that the cost reduction is strictly in software acquisition amount. The people cost is going to remain the same, and in some cases may actually increase as there is often not a large support infrastructure that is often associated with commercial tools.

The category of tools which can most easily be replaced by script controller frameworks is web automation of which there are two front runners right now.

- *Selenium*
  - Controlled by Python, Perl, Ruby (and others)
  - Has multiple sub-projects which give you Record / Playback or distributed execution
  - Startups which use Selenium as a central part of their business are starting to appear
- *Watir* - Web browser automation; Ruby
  - Ruby only
  - Original project is Internet Explorer focused; forks exist for Firefox, Safari and Opera
  - Commercial support is available from the lead developer

Even if you are using a commercial tool, a large amount of your time is going to be spent scripting in the tools language. Here is the typical evolution of a UI automation script.

1. Record the test
2. Add the testing conditions
3. Data drive scripts to increase test data without increasing scripting code
4. Evolve data driving to generate own test data (see Test Data Generation above)

Anything beyond the first step is a scripting task.

Increase Speed

There are a lot of repetitive tasks a tester does in the course of their work. The three areas were a scripting language can help decrease this time cost are regression, navigation and environment setup/teardown.

Regression tests are a prime example of something that should be automated, after all that code is supposed to be stable. These tests can run at night or on the weekends while the test team is not in the office. Computers can also run without a break, and their throughput is much greater than that of a person (in most situations).

Scripts can also increase the speed of a tester doing their work by zooming them through the UI to the screen their testing is to focus on. Take a system which deals with mortgages. Before a tester can investigate the 'Edit existing mortgage' screen you have to create an applicant, approve their mortgage, fund it and perhaps make a payment (or payments). By scripting up all the prerequisite steps you can get to the screen needed faster which means you can provide information about the area of the product you are supposed to be testing.

As anyone who has working on a multi server / service product knows, environment setup and teardown is a very time intensive task. Scripting languages can help decrease both these costs. I have had success with automated build distribution and installation by scripting up the console version of the installation wizard across a pool of test machines. When you have over 10 machines assigned to you, and close to 80 for the entire test team, the time savings add up quite quickly. Great time savings can be achieved through the seeding of test data in your database as well and of course removing any data that you created or modified through testing activities can make your environment more reusable.

Go forth and Script

The reasons illustrated for scripting and the types of scripts you can produce have only barely scratch the surface of all the possibilities. But even these I feel are compelling reasons to start integrating a scripting language and the power and opportunities it presents into your testing activities. By providing sample scripts that I have used successfully on many projects and tips about how to create your own I hope to have further lowered the barriers-to-entry around scripting for testers.

Listing 1 - DeveloperNotes.py

```
import os, os.path

def do_search(starting_path, res):
    # walk through our source looking for interesting files
    for root, dirs, files in os.walk(starting_path):
        for f in files:
            if is_interesting(f) == "yes":
                # since it is, we now want to process it
                process(os.path.join(root, f), res)
    print_results(res)

def is_interesting(f):
    # set which type of file we care about
    interesting = [".java", ".cpp", ".html", ".rb"]

    # check if the extension of our current file is interesting
    if os.path.splitext(f)[1] in interesting:
        return "yes"
    else:
        return "no"

def process(to_process, res):
    # make a list of things we are looking for (all lowercase)
    notes = ["todo", "to-do", "fixme", "fix-me"]
    # open our file in "read" mode
    r_f = open(to_process, "r")
    # make a counter
    line_num = 0
    # read our file one line at a time
    for line in r_f.readlines():
        # circle through each of the things we are looking for
        for note in notes:
            # check if our line contains a developer note
            # note we a lower()ing the line to avoid issues of
            # upper vs lower case
            # note also the find() function; if the thing it is
            # looking for is not
            #   found, it returns -1 else it returns the index
            if line.lower().find(note) != -1:
                # initialize our results to have a key of the
                #file name we are on
                if not res.has_key(to_process):
                    # each value will be a list
                    res[to_process] = []
```

```
                # add our information
                res[to_process].append({"line_num": line_num,
"line_value": line})
        # increment our counter
        line_num += 1
    r_f.close()

def print_results(res):
    # check if there was any developer notes found
    if len(res) > 0:
        # asking for a dictionary's keys gives you a list, so we
can loop through it
        # rememeber, we used the file name as the key
        for f in res.keys():
            # the %s syntax says "put a string here", %d is the
same for a number
            print "File %s has %s developer notes. They are:" %
(f, len(res[f]))
            # our value for the key here is a list, so again, we
can loop through it
            # (see, for loops are way too handy)
            for note in res[f]:
                # embed a tab for cleanliness
                print "\tLine %d: %s" % (note["line_num"],
note["line_value"])
    else:
        print "No developer notes found."

# set our base for our source code
source_base = "path\to\your\code"

# create a dictionary which will hold our results
results = {}

# go!
do_search(source_base, results)
```

Listing 2 - LOUD.py

```
import os

my_class = """ public class MyResources extends
ListResourceBundle {
      public Object[][] getContents() {
              return contents;
      }
      static final Object[][] contents = {
      // LOCALIZE THIS
              {"OkKey", "OK"},
              {"CancelKey", "Cancel"},
      // END OF MATERIAL TO LOCALIZE
      };
 }
"""

# normally you would open a file and iterate over it's contents
# but this is an example...
my_class_lines = my_class.split("\n")

# process each line
for line_num in range(0, len(my_class_lines) -1):
    # sure, a regex is likely the way to do this, but...
    key_sep = '", "'
    key_sep_at = my_class_lines[line_num].find(key_sep)
    end_point = '"}'
    end_point_at = my_class_lines[line_num].rfind(end_point)
    if key_sep_at is not -1:
        # break the line into chunks
        beginning = my_class_lines[line_num][0:key_sep_at +
len(key_sep)]
        middle = my_class_lines[line_num]
[len(beginning):end_point_at]
        end = my_class_lines[line_num][end_point_at:]
        # LOUD the middle
        middle = "^%s$" % middle.upper()
        # put it back into our copy of the class in memory
        my_class_lines[line_num] = "%s%s%s" % (beginning,
middle, end)

# write the file out to disk... okay, to the screen in this case
print "\n".join(my_class_lines)
```

Listing 3 - Browsers.rb

```
#!/usr/bin/ruby

# buckets for totals
browsers = {"ie6" => 0, "ie7" => 0, "ie8" => 0, "ff2" => 0,
"ff3" => 0, "s3" => 0, "s2" => 0, "opera" => 0}
bots = {"msn" => 0, "slurp" => 0, "googlebot" => 0, "become" =>
0}

# put the gateway ip's for your office here
office = ["x.y.z.a"]

total = 0
File.open("/Users/adam/work/parser/adam.log", "r").each do |
line|
    # one crazy regex -- token[9] is the agent string
    token = /^(.*?)\s+(.*?)\s+(.*?)\s+(\[.*?\])\s+(\".*?\")\s+
(\d+)\s+([\d-]+)\s+(\".*?\")\s+(\".*?\")$/.match(line)

    # remove the office gateway
    if office.index(token[1])
        next
    end

    # counter
    total += 1

    # grab the user agents
    # ie6
    if token[9].index("MSIE 6")
        browsers["ie6"] += 1
    # ie7
    elsif token[9].index("MSIE 7")
        browsers["ie7"] += 1
    # ie8
    elsif token[9].index("MSIE 8")
        browsers["ie8"] += 1
    # ff2
    elsif token[9].index('Firefox/2')
        browsers["ff2"] += 1
    # ff3
    elsif token[9].index('Firefox/3')
        browsers["ff3"] += 1
    # safari
    elsif token[9].index('Safari')
```

```
        # 3.x
        if token[9].index('Version')
            browsers["s3"] += 1
        # 2.x
        else
            browsers["s2"] += 1
        end
    # opera
    elsif token[9].index('Opera')
        browsers["opera"] += 1
    # msn bot
    elsif token[9].index('msnbot')
        bots["msn"] += 1
    # yahoo slurp
    elsif token[9].index('Slurp')
        bots["slurp"] += 1
    # googlebot
    elsif token[9].index('Googlebot')
        bots["googlebot"] += 1
    # becomebot
    elsif token[9].index('BecomeBot')
        bots["become"] += 1
    else
        #puts line
    end
end

puts "Total hits: " + String(total)
puts "\tInternet Explorer 6: #{((Float(browsers["ie6"]) / total)
* 100).round}"
puts "\tInternet Explorer 7: #{((Float(browsers["ie7"]) / total)
* 100).round}"
begin
    puts "\tInternet Explorer 8: #{((Float(browsers["ie8"]) /
total) * 100).round}"
rescue ZeroDivisionError
    puts "\tInternet Explorer 8: 0"
end
puts "\tFirefox 2.x:         #{((Float(browsers["ff2"]) / total)
* 100).round}"
puts "\tFirefox 3.x:         #{((Float(browsers["ff3"]) / total)
* 100).round}"
puts "\tSafari 2.x:          #{((Float(browsers["s2"]) / total)
* 100).round}"
puts "\tSafari 3.x:          #{((Float(browsers["s3"]) / total)
* 100).round}"
```

```
puts "\tOpera:                #{((Float(browsers["opera"]) /
total) * 100).round}"
```

Listing 4 - Loyalty.py

```
import random, sys, string

def verify_number(to_verify):
    # length check
    if len(to_verify) != 9:
        # raise an exception with some context
        msg = "Invalid account length"
        raise SystemError, msg # yes, I know, SystemError isn't
really the right error

    # only numbers
    for char in to_verify:
        if char not in string.digits:
            print "Accounts can only have numbers"
            sys.exit(1)

    # cant start with zero
    if int(to_verify[0]) == 0:
        print "Accounts cannot start with 0"
        sys.exit(1)

    # checkdigit
    run_total = 0
    for run in to_verify[:-1]:
        run_total += int(run)
    mod = run_total % 10
    if int(to_verify[-1]) != mod:
        print "Check-digit failed"
        sys.exit(1)

    print "Account is valid"

def generate_number():
    account_num = []
    account_num.append(str(random.randint(1,9)))
    for ix in range(0,7):
        account_num.append(str(random.randint(0,9)))
    run_total = 0
    for run in account_num:
        run_total += int(run)
    mod = run_total % 10
    account_num.append(str(mod))
    print "Your new account number is %s " %
"".join(account_num)
```

```
if __name__ == "__main__":
    if len(sys.argv) == 2:
        # show a try/except block in a real example
        try:
            # verify provided number
            verify_number(sys.argv[1])
        except SystemError:
            print caught
            sys.exit(1)
    elif len(sys.argv) == 1:
        # generate number
        generate_number()
    else:
        print "usage: my_script.py [account number]"
```