CST8177

Scripting 3: How?

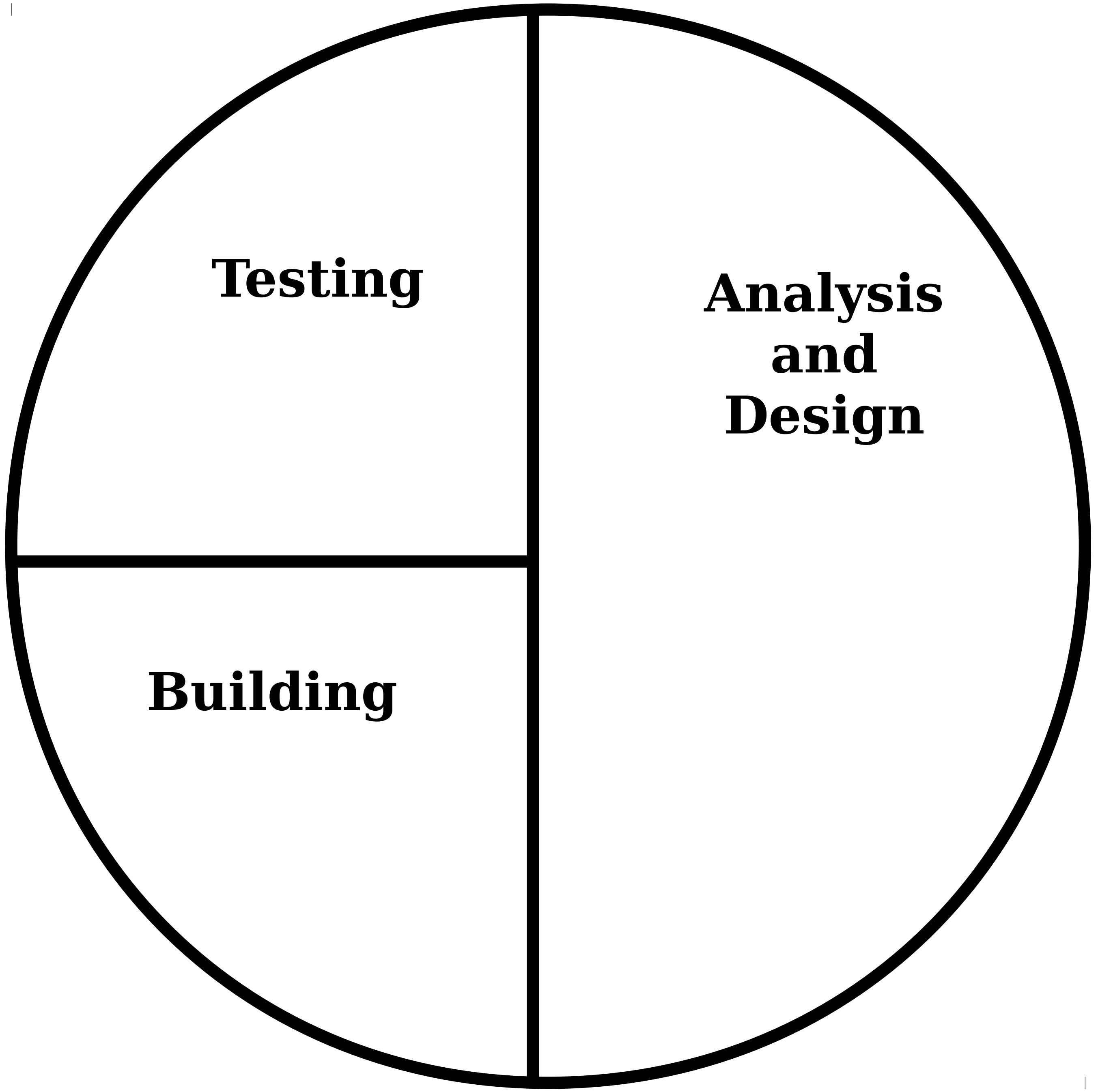
But First ...

How do you approach a scripting problem?

There are several steps to the process. In simplified form, they include:

1. Analysis
2. Design
3. Building
4. Testing

Let us first have a good look at these steps.


Testing
Analysis and Design
Building

1. The Analysis Process

First of all, decide what the actual problem is that is to be solved. Sounds simple, right? It may be, if it's a program you need, and you have a good idea of what it needs to do.

It's not so simple if you're writing it for someone else, like your boss (or your professor). You will have to carefully read whatever you've been given, perhaps checking back to clarify parts, until you have a full understanding.

In either situation, lay down the full requirements in writing. Include in this document considerations you have carefully gone over, including:

- What inputs are needed? Be sure to consider both your own files, system or user file, the command-line arguments, and **`stdin`**. If you need a config file, decide what will go in it.
- What kind of normal output will there be? Consider **`stdout`**, log files, and system and user files. Define the format if you can.

1. The Analysis Process

- What kind of errors can be expected? What about unexpected errors, like missing files? Consider every possible (and some impossible) eventuality. Write down the conditions for the error and decide what debug information you will need to see. If it's also going to a system log file, make sure you follow the local style standard.
- Rough out, perhaps a list of bullet points, what steps need to be done, in the sequence you expect they will be needed. Consider decisions points and loops, but only in an informal manner.

Write a how-to-use document for anyone who might be using your script. Show the command line, its options and arguments, and any prompts the script will make to the keyboard user.

2. The Design Process

You may need to have your Analysis approved, or at least reviewed by a class-mate. This is the time to make sure you are proceeding in the correct direction.

Now you can expand the roughed-out steps into full PDL (Problem Description Language). This describes <u>**WHAT**</u> the script is doing and <u>**WHY**</u> it's doing it. You do <u>not</u> need to describe <u>**HOW**</u> it's to be done -- that comes later.

Define the all input data the script will need, if any. Choose a file name if it's your own file, otherwise write down the name of the system or user file. Specify both the content and the layout (the format of the content). Don't overlook command-line options and arguments.

Do the same for all the output and the error messages. Actually write the text for each one. If you will be using the system logging facility, state the facility and priority to be used. If this is an addition to the logging rules, specify also the name and location of the (new?) log file.

2. The Design Process

You will now have a well thought-out plan for your script. The next step is to walk through walk through the PDL, step by step, with reference to your input and output. You can update the PDL at this time as you uncover steps you have missed, or glossed over too simply.

Finally, create and document a Test Plan. The goal of your Test Plan is to ensure that you can turn over the Plan, the final script, and the how-to document to a class-mate and expect them to be able to produce correctly-formatted test results.

To make this possible (and to simplify your own execution of the Test Plan), design at this time any special files or tools, even other scripts, that will make testing simple and repeatable. Reduce the amount of typing on **`stdin`** that's required by redirecting a file as input, and also redirect **`stdout`** and **`stderr`** into separate files for automatically checking the results.

3. The Building Process

It's time to write the script. By now you have developed a solid understanding of what problem is being solved and what the script is going to do to solve it.

As a first step, construct any files that are needed, such as a configuration file. You're going to test each chunk of script as you write it, so have this in place at the start.

Now convert the PDL to script. This is the **HOW** of the script, where the WHAT and WHY of the PDL become actual scripting statements. You do it a few lines at a time; many people turn the PDL into comments, and leave it throughout the script to describe what the next bit of script does.

You may have to turn to one side, so to speak, to write a tiny test script. This will help you determine how you can correctly build some of the complex parts of the script. On the other hand, you can keep useful script fragments to cut-and-paste into a script you are building.

3. The Building Process

Convert the PDL to script by repeating the following steps until it's all done:

- Write a few lines of script based on the PDL;
- Save your script file; you may wish to use a method that allows you to keep the last-good-version (save the current working copy under a test name; copy the 2 generations of backups only when this one has tested clean and error-free);
- Run the script in debug mode, carefully making sure each chunk of script behaves as expected before moving to the next;
- Make full and complete fixes of any script errors. If necessary, go back to your Design material and fix it as well.

Once you have completed this translation from PDL, run through the whole script as you mean it to be used.

4. Test process

Now it's the time to run the Test Plan, fixing errors until your script passes all the tests. This is often called the Alpha test, where the developer(s) of the script establish that what they have built operates as expected, even in error cases.

If you can (unlikely, I expect), get a class-mate who unfamiliar with your script to run the Test Plan from the how-to-use document. This is a lost art, it appears, that of the Beta test. In times gone by, companies would have quality assurance people who did this for a living.

The next step is to distribute the script, any required files, and the how-to document to a small set of live users. This may still happen in the Real World, since I still see references to Release Candidates.

Assuming all of the above proceeds smoothly, you can now release your script to the world, if that was the goal.

An example

This might be a casual or informal description from the Analysis of one of the script requirements:

kick user out unless root

Notice the brevity and simplicity. In the Design, turn this into good PDL - - describing **WHAT** and **WHY**, not HOW:

```
IF user is not root
   DISPLAY error message
   EXIT
ENDIF
```

There are a few things to note here. The keywords that make this PDL are shown in UPPER CASE, in an effort to increase their visibility. For much the same reason, the PDL is indented for the contents of the IF-block. And each control structure, the IF-ENDIF here, is paired to mark both its beginning and the end.

An example

Here, then, is the actual code -- the **HOW**:

```
#! /bin/bash
# kick user out unless root
if [ $UID != 0 ]; then
  echo Must be root user
  exit 1
fi
```

In some ways it's not too unlike the PDL. It looks almost like a one-to-one replacement, but that's not always the case.

An example

You might wonder why it use **UID** and not **USER**?

Surely comparing **$USER** to **"root"** can't be all that different from comparing **$UID** and **0**. It's because **UID** has been set to be read-only, but the **USER** variable can be changed by any user.

```
Prompt$ echo UID is $UID and USER is \"$USER\"
UID is 500 and USER is "allisor"
Prompt$ USER="root"
Prompt$ echo UID is $UID and USER is \"$USER\"
UID is 500 and USER is "root"
Prompt$ UID=0
bash: UID: readonly variable
Prompt$ echo UID is $UID and USER is \"$USER\"
UID is 500 and USER is "root"
```

An example

That was fairly painless, I hope. You have just learned 5 components of PDL:

1. **`IF -- ENDIF`** is a control structure, using the result of a comparison to make a choice. It comes in two more flavours that we'll look at soon. It, like all control structures, is paired to mark both the start and end of the block of statements.
2. Statements inside control structures (often called blocks) are indented for visibility.
3. There are 6 comparison operators for numbers. They are: **`== != > >= < <=`**
4. **`DISPLAY`** (or **`PRINT`** or **`SHOW`**) are used as output statements.
5. **`EXIT`** leaves the script.

An example

You have even learned 8 components of scripts:

1. The hash-bang line: it must be the very first line in the script, since the **`#!`** serves to tell the shell that this is a script. The rest of that line is the absolute path to the script program, **`/bin/bash`** for us.
2. There can be blank lines anywhere you need them for readability. Don't be afraid to use a few.
3. A comment is marked by a **`#`**. You can use a full line comment as here, or put a comment on a statement line after the script part:

   ```
   exit 1   # leave now
   ```
4. This is a specific form of the **`if`**-statement. We'll look at the general form later. The **`fi`** (yes, that's **`if`** spelled backwards. Sigh) is the end of the **`if`**-block, just like the PDL.

An example

4. The indenting is also just like the PDL, and for the same reason: to improve the readability for us humans. Be sure to indent properly.
5. There are 6 comparison operators for numbers. They are: `== != > >= < <=` Actually there is a whole other set for scripts in addition to these, but we'll address that later.
6. You're already familiar with the echo(1) built-in, although it has features you haven't used. Have a look at the printf(1) command for formatted output.
7. The **`exit`** statement terminates the current shell. You may already use it to return after an **`su`**. It takes one number(or numeric variable) as an argument to return to the caller for status.

   By convention, **`0`** is OK and any other number (it should be **`>0`**) shows some sort of error condition.

A Test Script

Let's take the code snippet we've been looking at and turn it into a test script. The only real change is the addition of a couple of echo statements:

```
#! /bin/bash
echo current UID is $UID
if [ $UID != 0 ]; then
   echo Must be root user
   exit 1
fi
echo UID is root
exit 0
```

I have saved this in a test directory so I can run a few tests, both as a user and as root. Since I've already decided to use **UID** and not **USER**, since **UID** is protected, there's no reference to the global **USER** variable.

A Test Script

Here a short test run:

```
[allisor@mycroft tmp]$ ./test-uid
current UID is 500
Must be root user
[allisor@mycroft tmp]$ su
Password:
[root@mycroft tmp]# ./test-uid
current UID is 0
UID is root
[root@mycroft tmp]# exit
exit
[allisor@mycroft tmp]$
```

Did it work correctly?

Without an Analysis of the requirements and a Test Plan, how can I tell?