Engineering Quality Goes Bananas

How a 'Dumb Monkey' Helped One Company Automate Function Testing

By Rex Black, Daniel Derr and Michael Tyszkiewicz

Arrowhead Electronic Healthcare has been creating eDiarys on handheld devices since 1999. With the devices, Arrowhead helps pharmaceutical research and marketing organizations document information about how their products are being used in patients' homes.

Arrowhead's third-generation eDiary product is called ePRO-LOG. Its primary design goal was to be able to rapidly deploy dairies used for data collection in clinical trails and disease management programs. A typical diary might include 100 forms translated into 15 or more languages, and used in several locales. To handle the large number of software builds and configurations that resulted. the team needed an automated test tool to address potential risks and to automate common tasks.

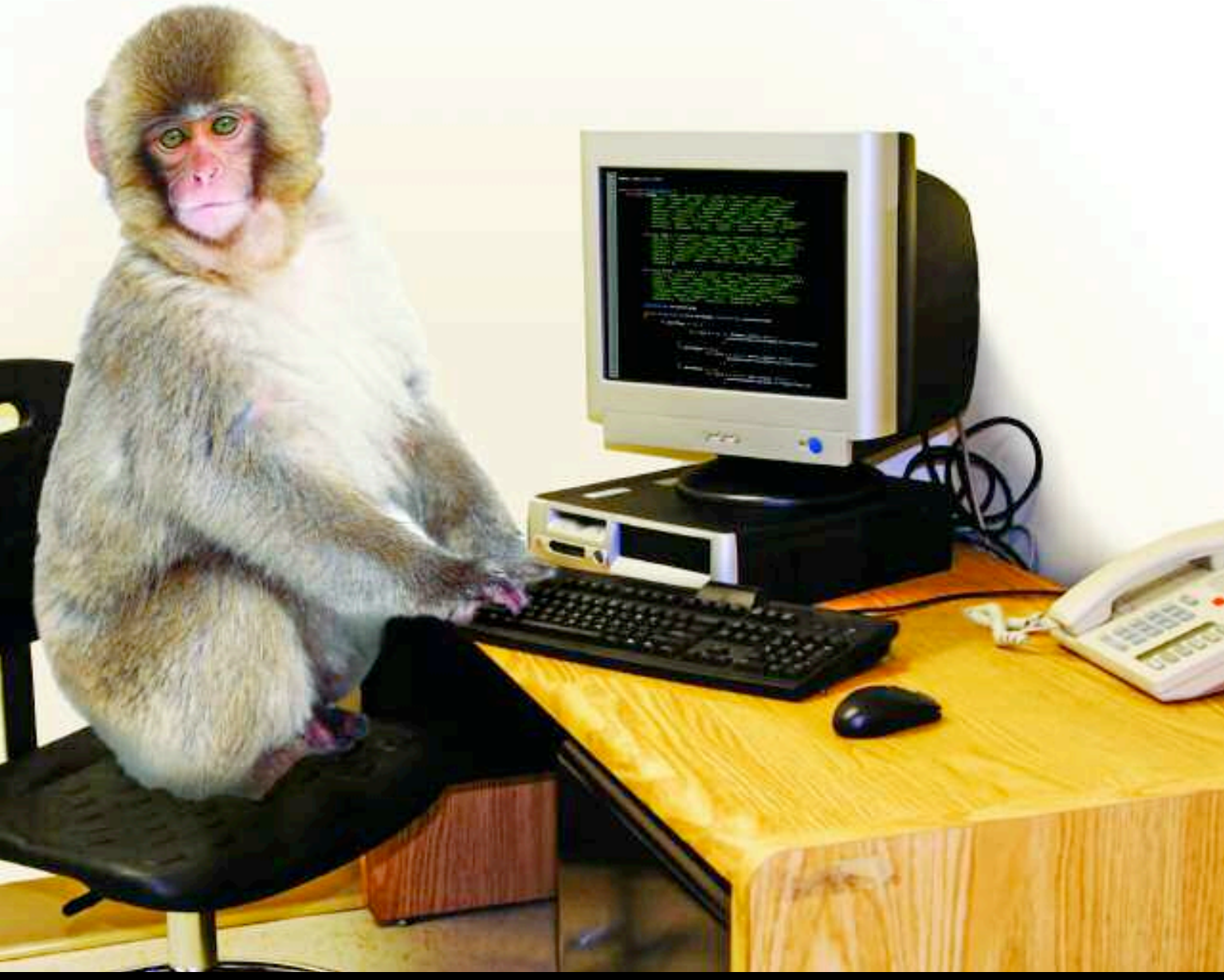
The most important quality risks we wanted to address were:

- Reliability
- Translation completeness
- Functionality of UI
- Input error checking
- Verification of requirements

The automation tool needed to do the following:

- Address defined risks
- Produce accurate form-flow diagrams
- Reduce tedium and opportunity for error in manual testing
- Save effort associated with manual testing for these risks
- Improve time-to-market by reducing test cycle duration through 24x7 testing
- Provide auditable documentation
- Handle any screen flow or translation without custom test scripts (i.e., be trial-independent)
- Be easy to implement
- Be cost effective

This is a case study in how we reduced our risks and achieved our test automation objectives in just a few months on a total outlay of $0 for tools. Now, it wasn't as if we started with zero cost as a target. Often, buying tools is the most cost-effective solution, so we evaluated test automation tools as a potential solution. Since we were developing custom software on a hand-held device, we found the commercial options limited. ePRO-LOG is highly configurable and optimized to make diaries easy to produce. The drawback of



our approach was that our widgets were non standard, and are therefore not handled gracefully by common testing tools. We also needed an easy way to generate screen flows and compare those with our requirements.

We had hit a dead end. We couldn't find a commercial tool to meet our needs and human labor was cost prohibitive. That's when the monkey came into the picture; a Dumb Monkey to be precise.

Why is the Monkey dumb? Because the architecture is so simple. The Monkey is an unscripted automated test tool that provides input at random. To minimize cost, effort, and time required for development, we implemented the Monkey in Perl under Cygwin. We also took advantage of our application's cross-platform functionality and performed the bulk of our testing on a Windows PC. This allowed us to test more rapidly.

Every test automation tool tends to have its own terminology, so let's start by introducing some terms, shown in Table 1 (next page).

The Monkey's Talents

The Monkey improves reliability in our application by randomly walking through the diary while trying different input combinations. The use of random events allows the Monkey to be diary-independent and generally does not require any customization (some customization was required to successfully login, otherwise the device would lock us out after too many attempts. Other special situations may also require customization).

During the Monkey's walk, it is constantly looking for broken links, missing images and input validation errors. The Monkey also can perform long-term reliability tests, which allow us to accumulate as many hours of testing as time and CPU cycles permit. By continuously stressing the application, potential defects are more likely to be discovered.

Such long-term reliability tests are ideal for testing after deployment, and require little human intervention. This allows our products to be continually tested while testing staff focuses on new development.

The Monkey tests more input combinations than a reasonably-sized manual test team could, thus increasing confidence and decreasing the likelihood of undiscovered defects. In addition, the screenshots,

Both at Arrowhead Electronic Healthcare, **Daniel Derr** is VP of software development, and **Michael Tyszkiewicz** is manager of QA. **Rex Black** is president of RBCS, a software test and development consultancy.

TABLE 1: MONKEY-PEDIA

Chef	Testability features added to the application to make the Monkey Chow
Monkey Chow	Human readable description of the screen produced by the application in real-time
Eat Monkey	Reads in the Monkey Chow and creates a data structure suitable for the Think Monkey
Think Monkey	Takes in the data structure from the Eat Monkey and decides what action to take next
Watch Monkey	Captures screen shots of ePRO-LOG as the Monkey operates
Push Monkey	Interacts with the hand-held device's user interface. The Push Monkey creates custom Windows messages and sends them to the ePRO-LOG application. (Postmesg from http://xda-developers.com/ was used to send messages, however any method of sending a Windows message should work. XDA tool chain was chosen since it works for both Windows Mobile and a Windows PC.)
Monkey Droppings	Screen shots and human readable log files produced by the monkey to keep track of where it's been, what it's done, and what it's seen
Chunky Monkey	Encapsulates the Eat Monkey, Think Monkey, Watch Monkey and Push Monkey, and produces the Monkey Droppings
Presentation Monkey	Transforms monkey droppings into graphical flow charts
dot file	A human readable data file used by the GraphViz dot application to generate abstract graphs (http://www.graphviz.org/)

FIG. 1: THE MONKEY'S BUSINESS

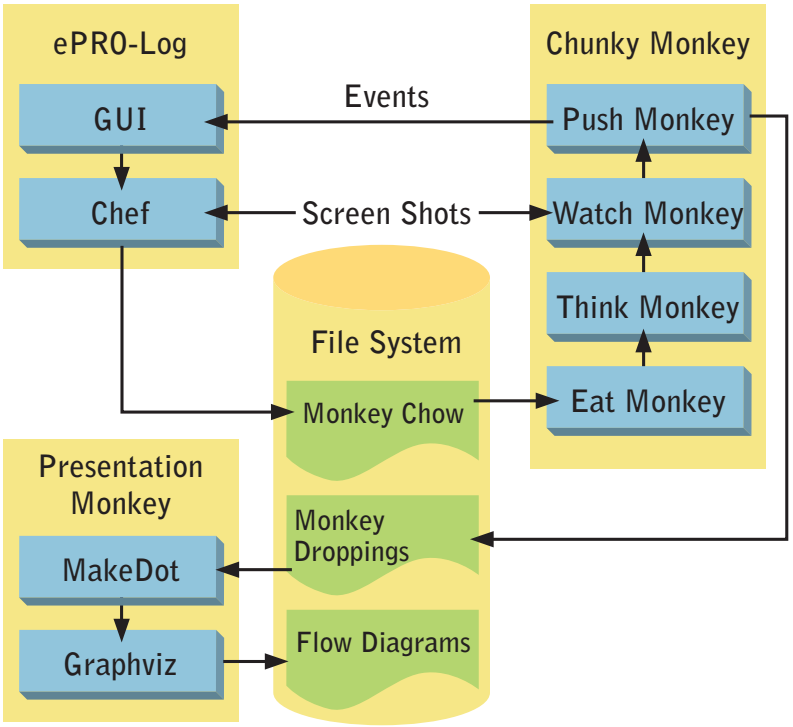


TABLE 2: APE ROI

	Manual	Automated
Test Plan Preparation Time (hrs)	225	42
Test Execution per cycle (human hrs)	7	3
Number of Cycles	35	35
Total Effort (hrs)	470	147
Savings (hrs)		323

Monkey Chow and Monkey Droppings created during the test process are saved in an auditable format. Auditable test results are important in environments that are subject to FDA regulations.

Diaries are typically translated into many languages. For each language, a translation tester must verify all screens. Screenshots captured by the Monkey are automatically inserted into a Word-formatted translation verification document. This document allows translation testers to verify the content and completeness of the screens. This approach is more efficient and less error-prone than navigating to the ePRO-LOG screens manually on a device.

Gifts of the Monkey

While using the Monkey over a four-month period, we noticed significant time savings, mainly in the areas of diary testing, screenshot capturing, and translation verification. We also enjoyed the benefits of long term reliability testing and faster cycle times.

The initial development of the Monkey took approximately 120 hours of a programmer's time over a three week period. This is an upfront cost and does not have to be repeated for each diary.

The Monkey allows the compression of two calendar days of functional testing into a single half-day. This allows for flexibility and changes during the test period.

The time saved doing translation verification for a single diary created in 14 different languages was approximately 323 hours (see Table 2), obviously surpassing the 120 hours required to develop the Monkey. Since the Monkey is diary independent, our return on investment will continue to grow the more we use the Monkey.

Anatomy of the Monkey

The monkey is a collection of Perl and other scripts (see Table 3), open source tools and minor testability enhancements to ePRO-LOG.

TABLE 3: SIMIAN SCRIPTS

chunkyMonkey.pl	A Perl script that implements the Eat, Think, Watch, and Push Monkeys. This script also creates the Monkey Droppings.
launchMonkey.sh	A Bash script used to invoke chunykMonkey.pl using the monkeyChow.txt as input, and redirecting output to monkeyDroppings.txt
makeDotFile.pl	A Perl script which processes monkeyDroppings.txt, and creates a GraphViz dot file
makeDot.sh	A Bash script which calls GraphViz (dot.exe) to convert a dot file into a BMP, GIF, JPEG, PDF, PNG, or SVG file

Collectively, the ePRO-LOG application and the scripts described in Table 3 implement the system described in Figure 1.

Let's take a look at examples of the three main types of documents that make up the Monkey's anatomy.

Monkey Chow describes the form and all of the widgets belonging to the form. Figure 2 shows some examples of Monkey Chow corresponding to FormHome. White space was added to make the data more readable.

To use this data to hit ButtonTools, we would pass in the form handle= "0x001502FA", lparam="0x001304A8", and controlId="0x5" to the Push Monkey. Additional data is used to provide insight to the Think Monkey and to make the Monkey Droppings more descriptive.

The subroutine in Listing 1 was extracted from the Push monkey. The print statement at the end will become a single entry in the Monkey Droppings.

Monkey Droppings record the output from the Chunky Monkey. The output consists of the current form, whether a screen shot was taken, and any actions taken by the Think Monkey. In the example in Listing 2, we started on the login screen, pressed Button1 four times, hit ButtonOkay, then selected ButtonTools on FormHome. Screens shot where also taken along the way.

This data can also be used to create a dot file for the Presentation Monkey. FormTools was added to the dot file for purpose of illustration. Listing 3 shows a sample GraphViz dot file.

LISTING 1

```
sub hitGraphicButton
{
 my $formContainer = shift;
 my $widgetParams = shift;

 my $handle = $formContainer->{"params"}->{"handle"};#
form:handle="0x001502FA":
 my $message= "0x000111";          #
WM_COMMAND message
 my $wParam = $widgetParams->{"controlId"};
       # controlId="0x5"
 my $lParam = $widgetParams->{"lparam"};      #
       lparam="0x001304A8":

 my $result = `postmsg.exe -p -h $handle $message
$wParam $lParam`;

 print "event" .
 ':name="'.$formContainer->{"params"}->{"name"} .'"'.
 ':type="'.'GraphicButton'.'"'.':name="'.$widgetParams-
>{"name"}."\"\n";
 #
event:name="FormHome":type="GraphicButton":
       name="ButtonTools"
}
```

LISTING 2

```
Storing image as: ../images/FormLogin.png
event::name="FormLogin":type="GraphicButton":
name="Button1"
event::name="FormLogin":type="GraphicButton":
name="Button1"
event::name="FormLogin":type="GraphicButton":
name="Button1"
event::name="FormLogin":type="GraphicButton":
name="Button1"
event::name="FormLogin":type="GraphicButton":
name="ButtonOkay"
Storing image as: ../images/FormHome.png
event::name="FormHome":type="GraphicButton":
name="ButtonTools"
Storing image as: ../images/FormTools.png
```

LISTING 3

```
digraph studyFlow
{
 FormLogin [label = "", shapefile =
"images/FormLogin.png"];
 FormHome [label = "", shapefile =
"images/FormHome.png"];
 FormTools [label = "", shapefile =
"images/FormTools.png"];

 FormLogin -> FormTools;
 FormLogin -> FormHome;
}
```

FIG. 2: THE MONKEY'S GUTS


```
ready:
form:handle="0x001502FA":objectGuid="21":type="1":
  name="FormHome":x="0":y="0":width="320":height="320":

widget:lparam="0x001A0496":controlId="0x1":objectGuid="22":type="6":
  name="ButtonExit":x="0":y="232":width="75":height="34":formGUId="21":

widget:controlId="0x2"

widget:lparam="0x001804A4":controlId="0x3":objectGuid="24":type="6":
  name="ButtonMainMenu":x="20":y="105":width="200":height="30":formGUId="21":

widget:lparam="0x000B0408":controlId="0x4":objectGuid="25":type="6":
  name="ButtonSendData":x="20":y="140":width="200":height="30":formGUId="21":

widget:lparam="0x001304A8":controlId="0x5":objectGuid="26":type="6":
  name="ButtonTools":x="20":y="175":width="200":height="30":formGUId="21":

formEnd:
```

Figure 3 displays the result of Presentation Monkey using GraphViz to render the dot file into a screen flow image.

The Monkey's Hidden Powers

The monkey has a latent capability that we have not yet used—the ability to verify the actual screen flows against the requirements specification. This is particularly important in an FDA-regulated environment where complete coverage of requirements are mandated by 21 CFR and other regulations. For companies that are operating in regulated environment, maintaining the required level of documentation can be a significant operating cost.

Figure 4 shows a comparison of specifications with screens and test screen flow. Let's work our way around this figure, starting with the sequence originating on the right side.

The Presentation Monkey can produce a screen flow diagram from the Monkey Droppings file as shown previously in Figure 3. This diagram shows what screens where observed during Monkey testing.

However, we can also produce a screen flow diagram using our requirements specification instead of the Monkey Droppings file. Our testers can use this diagram to show the expected functional flow of the application.

Now, that capability alone would be exciting enough, but would still leave the tedious and error prone task of comparing the two screen flows. However, we also have a comparator that can compare the test-based screen flow with the spec-based screen flow. The output is fed to the Presentation Monkey, which produces a comparison like that shown in Figure 5.

Figure 5 highlights the differences between the screen flow described in the specification and what was observed during testing. For example, the requirements specification called for the screen flow to proceed from FormT4 to FormT5 prior to entering FormSave, but instead we went straight from FormT4 to FormSave. In addition, the requirements specification called for the screen flow to proceed from FormI2 directly to FormSave, but instead we went from FormI2 to FormI3 before proceeding to FormSave. This capability greatly reduces the risk of releasing a product which does not adhere to customer requirements.

FIG. 3: THE MONKEY SHINES

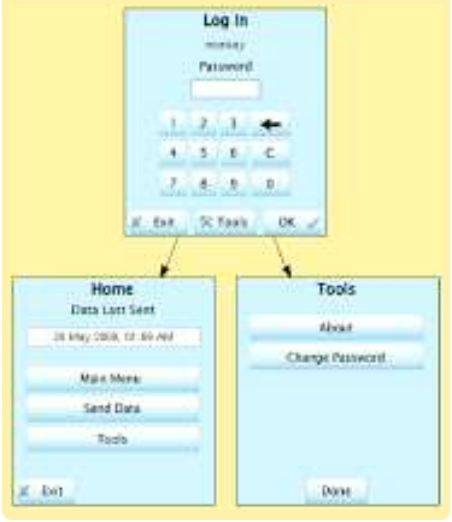


FIG. 4: CHIMP CHOICES

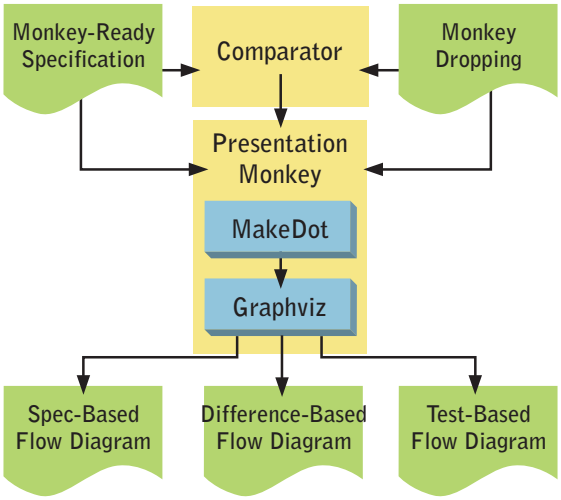
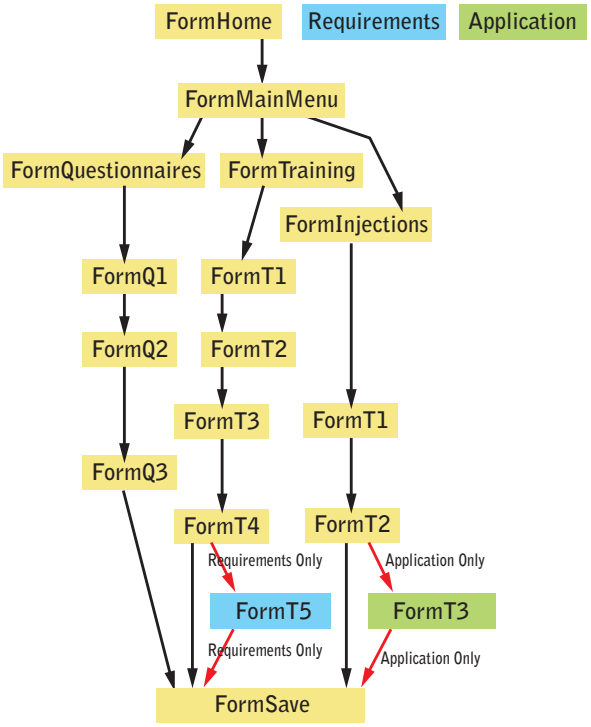


FIG. 5: PRIMATE PATHS



What's Next for the Monkey?

We plan to scale our usage of Monkey labor to perform long term software reliability testing. By using a large number of PCs or a Monkey cloud, we could simulate tens or even hundreds of thousands of hours of operation in as little as a week. This will allow us to produce statistically valid software reliability estimates for the ePRO-LOG.

We also intend to introduce scripting capabilities into the Monkey. This will allow for a pre-determined decision about screen flows (rather than a random decision) during scripted tests.

Creating a Monkey with simple architecture allowed us to address our risks while saving time and money. Using open source components and minimal software development effort, we created a custom testing application that provides far greater benefits than existing commercial products. The Monkey has already paid for itself many times over in time saved, and gives the company a competitive advantage by improving our documentation and testing, and allowing for faster turnaround time.

Also, it should be noted that no monkeys where harmed during the development of this application.