TestPoint in Practice

Ward Cunningham
Cunningham & Cunningham, Inc.

Submitted to
Fourth Annual Austin Workshop on Test Automation
January 25-26, 2003

Abstract

A web browser is a wonderful GUI testing tool. To make GUI programs browser compatible one must include a web server in the address space of the program under test. Web programs running under a multi-threaded application server such as Tomcat [1], Websphere [2] or Dynamo [3] already have this testing capability available through servlets. In this report we describe including TestPoint [4], an extremely simple HTTP 0.9 [5] web server, in two graphical simulations and applying a testing approach developed on application servers using servlets.

Introduction

James Bach describes Exploratory Testing [6] as a powerful alternative to scripted testing. His expectation is that a skilled tester will operate a program using the same interface as the intended user, but with a critical attention that will lead to valuable discovery faster than casual use by all but a large number of actual users. We add to Bach's approach the visibility of internal state made possible by custom interfaces to the program under test and suggest that the conventional web browser provides excellent means for the skilled test engineer to access such an interface while maintaining critical attention to the testing task.

In this paper we describe the exploratory testing of two graphical applications. Both are simulators. As such, they both exist to explore the properties of the systems they simulate. That is, the end user of the simulator is expected to be an "explorer" too. But the end user must trust that the simulation employs sufficiently accurate models and that these have been coded correctly. For this purpose the simulator itself must be tested. One kind of test is the duplication of known situations on the simulator and checking that the predicted results match reality. When there is no extant reality, as is the case in both simulators discussed in this paper, one must at least validate that the chosen models are implemented as intended and are capable of the accuracy required.

We will describe each simulator in turn. In each case we will provide just enough overview to understand the graphical nature of the applications and the sort of internal state exposed by the web server testing interface. Then we will report our first hand experience exploring the behavior of each simulator and reflect on strengths and weaknesses of the approach.

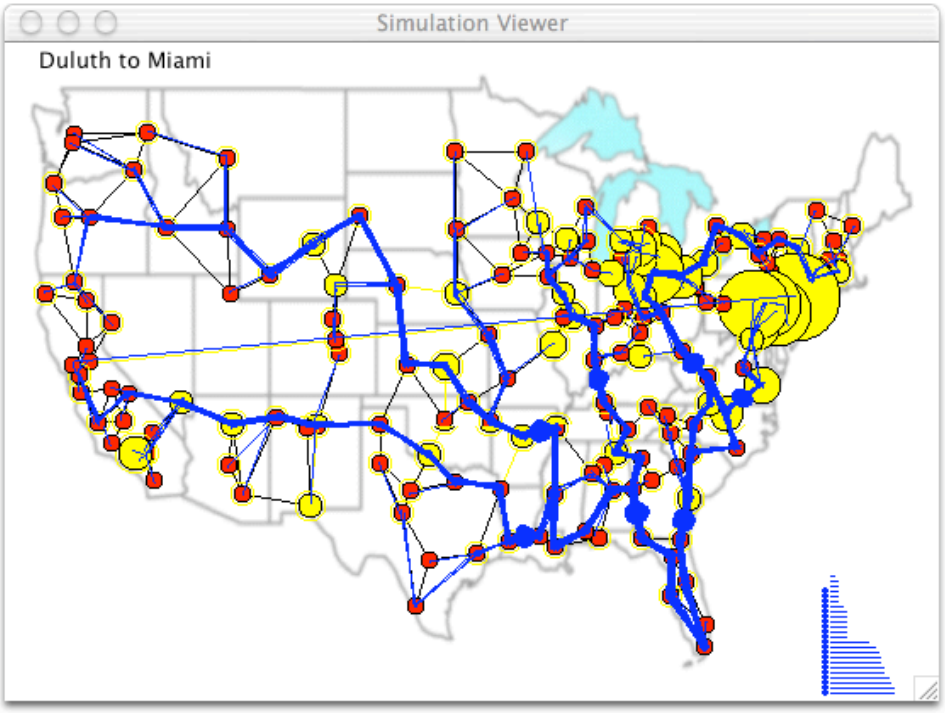
Radio Network Simulator

The SimNet [7] application simulates 173 computer controlled radios providing a backbone electronic message service imagined as a nation wide amateur radio project to be undertaken in the 1970's, the dawn of the microprocessor era. The program was initially written in Pascal for a CDC 6000 series super computer. Of a dozen pages of source code, half were devoted to collecting and plotting data. The other half, six pages, comprised the simulated radio controlled computers and was most specifically dedicated to dynamic hierarchal routing, a process that exhibited complex and emergent behavior that defied the analysis available at the time.

The SimNet program was revived in 2001 as a test case for refactoring for understanding [8]. The program was hand translated to Java and debugged on 30 year old data sets then aggressively refactored using intelliJ IDEA [9]. To aid in exploration three threads were added to the previously single threaded program:

- Animated Map Display
- Routing Query Mouse Handler
- TestPoint Web Server

This is a snapshot of the map display. A drag from Duluth to Miami would render a route trace as a blue line over the base map. Here the route rendering has been expanded to show routes from all possible locations to the destination, Miami.



This "all sources" route display changes as individual nodes choose to route around congestion. On occasion unusual routing decisions lead to unusual or irrational paths. These can be easy to spot visually, but hard to describe precisely enough to trigger debugging output. This is where the third new thread, the TestPoint web server, helps out. I modified the simulator report "what's happened recently" as a web page. Then, when I see something fishy, I tap "refresh" on the browser to see what has been going on inside the simulation. Here is how I did it.

I instrumented important routines with calls to **trace(String, Object)** which would add a trace record to a circular buffer [10]. Sometimes the calls would be specific to a line of inquiry. These would be added as needed and then removed when interest moves elsewhere. I've found making this editing simple gives me much more control over what traces look like than using a fancy trace package. Some trace calls provide a general gestalt and are left in place indefinitely. Here is the call that records event dispatching, the heart of a discrete event simulator.

```
void dispatch() {
    if (eventQueue.isEmpty()) return;
    thisEvent = dequeue();
    clock = thisEvent.time;
    trace("dispatch", thisEvent);
    thisEvent.dispatch();
    eventCount++;
}
```

I've also found that having a low overhead in the trace handler allows me to trace inner loops and critical sections that would be difficult to observe with heavyweight trace packages. Here is the implementation of trace I've used in this case.

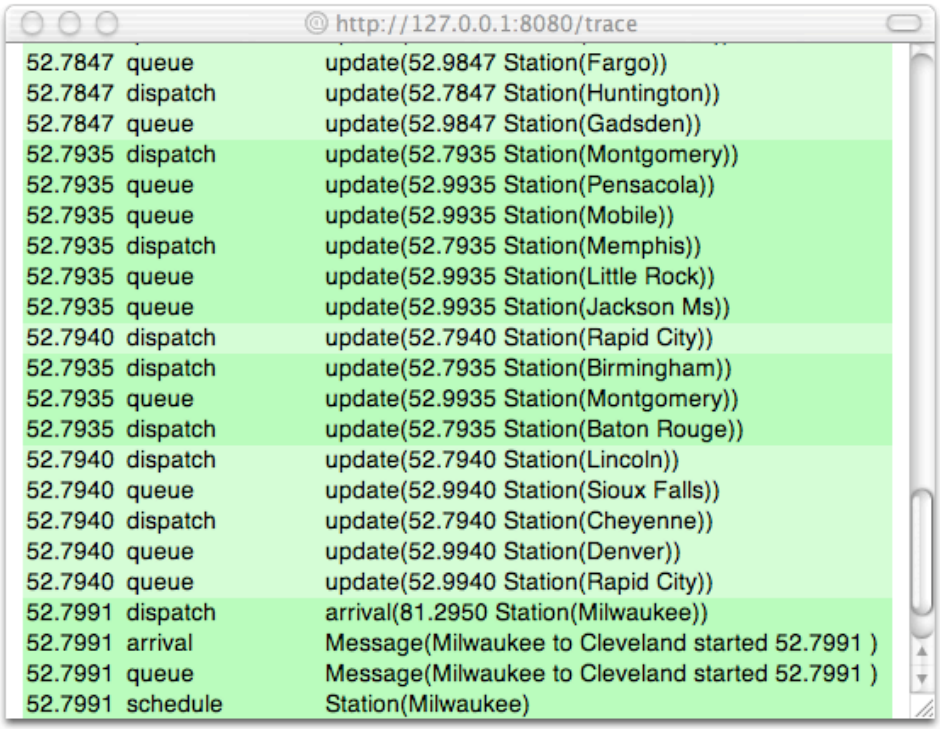
```
static void trace (String label, Object argument) {
    Trace newTrace = traces[nextTrace];
    if (newTrace == null) {
        newTrace = traces[nextTrace] = new Trace();
    }
    nextTrace=(nextTrace+1)%traces.length;
    newTrace.time = clock;
    newTrace.label = label;
    newTrace.argument = argument;
}
```

The **Trace** class is really just a struct that holds **trace**'s parameters and the simulator's clock. Notice that, except for the lazy initialization of the **traces** array, this routine does not allocate memory and neither does its caller. This "pointer pushing only" approach means that tracing can be left in place even in production programs. This routine can be synchronized with some additional overhead should calls to trace come from more than one thread.

The burden of formatting trace output can be delayed until there is a request to see that output. Such requests are rare in comparison to calls on **trace**. A top level exception handler would be a great place to print a trace. It's output is much more semantic than the runtime stack that one usually prints. And, of course, we will print a trace on request from a web browser. Here is the formatting routine we use to do so.

```
static void printTrace(PrintWriter out) {
    out.println("<table cellspacing=0 cellpadding=2>");
    double clock=0;
    int tick=0;
    for (int i=nextTrace+1; (i%traces.length)!=nextTrace; i=(i+1)%traces.length) {
        Trace t = traces[i];
        if (t==null) continue;
        if (t.time != clock) {
            clock = t.time;
            tick++;
        }
        String color = tick%2==0 ? "#cfffcf" : "#afffaf";
        out.println("<tr bgcolor=" + color + ">" + t + "</tr>");
    }
    out.println("</table>");
    out.flush();
}
```

This is sample trace output as viewed with Microsoft Internet Explorer.



The TestPoint web server expects one "microservlet" method per page. This is the method used to call the formatter.

```
void trace() {
    Simulator.trace("TestPoint", new java.util.Date());
    Simulator.printTrace(out);
}
```

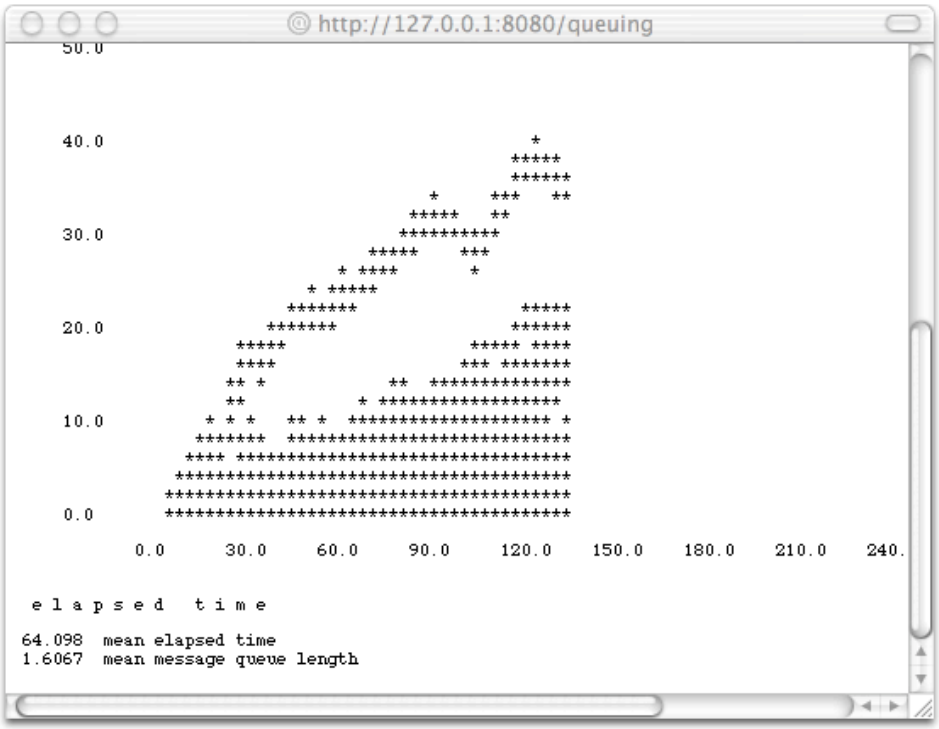
When the http request for the trace page arrives, the server calls this method by reflection. The method marks its own running in the trace and then prints the entire buffer as the http response. The servlet marks the trace so that one can see where new information begins when rapidly refreshing the trace page.

Since domain objects are used to represent trace parameters it is important that they not change state in significant ways between the time that they are recorded in the trace and the trace is printed. One approach is to only record immutable objects as parameters. Here I have used the slightly less restrictive convention of letting domain objects print themselves in the trace and being careful to only report immutable state from such prints. When this proves inconvenient one can always **toString** the object and record the resulting string in the trace, though this will allocate dynamic storage.

The browser offers a convenient way to make asynchronous requests of the program under test. The browser also offers many alternatives for formatting output from requests. The '70s version of the simulator ended every run by printing a batch of line-printer plots. It turns out that the plot routines could just as easily print partial results in response to asynchronous requests from a browser. Here is a TestPoint handler that prints the aggregate queue lengths over time using a small enough font to get the line-printer effect.

```
void queuing () {
   out.println("<font size=-1><pre>");
   Simulator.queuing.report(out);
}
```

This is a sample of "classic" line-printer output. This is data for all 173 nodes plotted on top of each other. This makes for a bit of a mess except that the few nodes with huge backlogs isolate themselves from that mess in this graph.

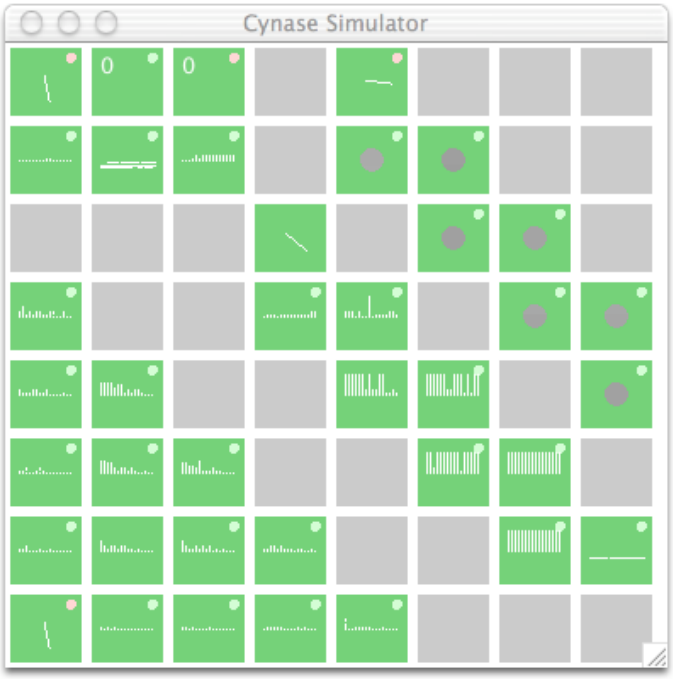


Line-printer plotting is a hack with parallels in the modern world. Simple graphs are easily drawn by coloring squares of a table or varying the dimensions of an image. Sometimes flows of simple character codes can stimulate our inate human pattern recognition abilities as I have done with browser based "signature surveys" [11] reported elsewhere.

Biologically Inspired Computing

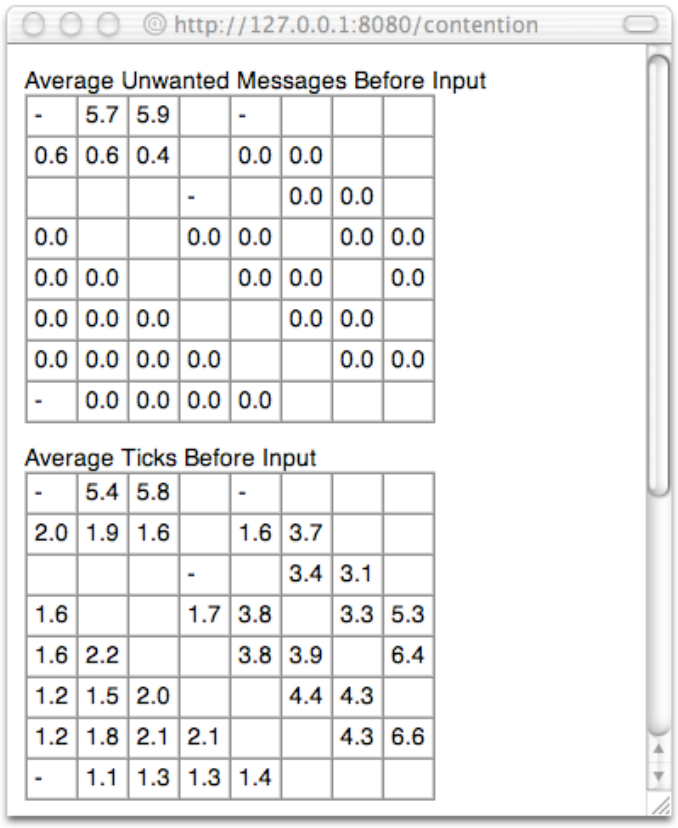
The second simulator also tracks the flow of messages through a network. Cybords are small circuit boards that communicate with neighboring boards using Cynase, a network flooding style communication protocol that is partially inspired by protein kinase signal transduction in the cell. The boards are designed to self-organize. The simulator exists to test organization principles in the presence of frequent reconfiguration. This work is an outgrowth of the workshop on Biological Framings of Problems in Computing [12].

The simulator is 1000 lines of Java, 200 lines of which do graphical user interface and another 200 lines, diagnostic web output. The GUI offers an external view of boards and can reconfigure them with mouse drags. The web interface offers an internal view of processes running on each board. This is a snapshot of the simulator running with four independent experimental configurations assembled and operating at once.



All of the boards animate their current state in one way or another. The physical boards will be less visual, but far more tactile in that they will interface with a wide variety of sensors and actuators so as to interact with the world.

The following snapshot is of Microsoft Internet Explorer viewing one of eight different pages offered by the simulator. The page shows three different signal contention statistics, one per board in each of three tables (two visible here). The web pages comprise 15 different tables, all but one of which is organized to reflect the current configuration of boards.



The following snippet of code is typical of that which produces each table. There are two parts: first a **println** that labels the table to follow, then a call to the custom **table** method that prints either transmit (tx) related or receive (rx) related statistics depending on the nature of the anonymous inner class passed as an argument. In this case the argument defines **rxPrint** so the average of **unwantedStats** is printed for each receiving node.

```
out.println("Average Unwanted Messages Before Input");
table(new Datum() {
   void rxPrint() {
      out.print(node.unwantedStats.ave());
   }
});
```

The **table** method is implemented with the expected doubly-nested loop running over rows and columns of nodes and emitting html <table>, <tr> and <td> tags as it goes.

```
private void table(Datum d) {
   d.out = out;
   out.println("<table border cellspacing=0 cellpadding=4>");
   for (int i=0; i<Simulator.matrix.length; i++) {
      out.println("<tr>");
      Node m[] = Simulator.matrix[i];
      for (int j=0; j<m.length; j++) {
         out.print("<td>");
         Node n=m[j];
         if (n!=null) {
            d.at = new Point(j,i);
            d.node(n);
            d.print();
         } else {
            out.print(" ");
         }
         out.println("</td>");
      }
```

```
            out.println("</tr>");
        }
        out.println("</table><br>");
    }
```

A **Datum** is an adapter in the Design Patterns [13] sense. It does only enough to make it easy for a Java method to pass a fragment of code to another.

```
    class Datum {
        Node node;
        Point at;
        HorzBus horz;
        VertBus vert;
        PrintWriter out;

        void node(Node n) {
            node = n;
            horz = n.horz;
            vert = n.vert;
        }

        void print() {
            if (node.rx) rxPrint();
            else txPrint();
        }

        void rxPrint()      {out.print(" - ");}
        void txPrint()      {out.print(" - ");}
        double round(double v)  {return Math.round(100*v)/100.0;}

    }
```

Experience

As mentioned in the introduction, both of these programs were written to support exploration. As such, both had graphical user interfaces that visualized primary domain data in original ways. However, in both cases there was considerable hidden data, as there always is in any large program. I have found the notion of "browsing" to be a good metaphor for the activity of exploring this additional state. Also, not surprisingly, I've found the modern web broswer an excellent tool for supervising the browsing activity.

The browser cooperates with web servers. The TestPoint web server and the servlets built upon it serve my browsing activity well for three reasons:

- queries are about right now
- answered using programming language
- formatted to enhance gestalt

I will consider each in turn.

The answers I get from the servlets are the answers about the state right now. This is a tremendous simplification from scanning log files or trying to write logic to trap conditions of interest. Instead I simply watch the program run and refresh the browser at the moment of interest. The effect is much like that of the photographer attempting to capture the "decisive moment" [14]. This may sound difficult but it is not because the developing is instantaneous and the film is free.

The answers are gotten using the same machinery as used to program the applications themselves, Java. This has been discusses at length in regards to unit testing. I feel the most important benefit is that the tests and the program are much more likely to co-evolve gracefully in the presence of rapid change.

The answers are easily formatted using position, alignment, contrast, color and size in ways that advance rapid recognition of patterns. So often I find myself looking at the data to just see what is there. The browser is plenty fast. The limiting factor is my ability to absorb the information it presents. With well formatted information I can often simply refresh the page over and over and watch for unexpected results.

The TestPoint server used in these two cases was written to bring servlet style remote access specifically to GUI based programs. I have already had positive experience with the approach on large scale, three tier, commercial applications.

These two cases differ in that there is no a priori known good behavior for which I am looking. This says more about the kind of programs that I write than it does about any limitation of the technique.

References

[1] http://jakarta.apache.org/tomcat/
[2] http://www-3.ibm.com/software/info1/websphere/
[3] http://www.atg.com/en/products/overview.jhtml
[4] http://c2.com/doc/TestPoint/
[5] http://www.w3.org/Protocols/HTTP/AsImplemented.html
[6] http://www.satisfice.com/articles/what_is_et.htm
[7] http://c2.com/~ward/morse/SimNet/
[8] http://www.visibleworkings.com/archeology/hendrickson.htm
[9] http://www.intellij.com/idea/
[10] http://c2.com/cgi/wiki?CircularBuffer
[11] http://c2.com/doc/SignatureSurvey/
[12] http://www.dreamsongs.com/BiologicalFramings.html
[13] http://c2.com/cgi/wiki?DesignPatternsBook
[14] http://www.ip.pt/photoceania/cartier.htm
[15] http://junit.sourceforge.net/doc/testinfected/testing.htm