PAGEUNIT: A “LITTLE LANGUAGE” FOR UNIT TESTING OF WEB APPLICATIONS

IAN F. DARWIN

A dissertation submitted in partial fulfillment of the requirements of

Staffordshire University for the degree of M.Sc.

Diploma Module

The development of the software accompanying

this research programme was primarily funded by the

Toronto Centre for Phenogenomics.

January 2006

Table of Content

PageUnit, a "Little Language" for Web Application Testing

Ian Darwin
M. Sc. Candidate
Staffordshire University

Abstract

To facilitate testing of web applications, a “little language” in the sense of [Bentley1986b] was designed and developed. While the framework was written in Java and one extension was made to facilitate the testing of J2EE Web Applications [Darwin2003], the testing framework is quite general and has been used to test web applications written in Java, PHP, and others.

As a module in the M. Sc. Program in Professional Computing, this work has consisted of research into the areas of “Little Languages” and of the theory of software testing; the practical portion has consisted of the design and implementation of the PageUnit framework.

1.0 Introduction

The need for automated testing grows as fast as the complexity of web sites grows. But the cost of writing test suites grows as well. While some web frameworks [Darwin2004a] such as Ruby on Rails [Hansson2004] provide their own testing tools, some of the most widely-used packages including Sun's J2EE do not. The Java developer is left with a choice of using a commercial tool such as Empirix, or using open source. A "recording" based open source package such as Solex may suffer from the disadvantage that entire web pages including large JavaScript entries are saved in their entirety. A “code based”' open source package such as HtmlUnit or HTTPUnit requires too much coding to be useful. Experience with HttpUnit at the

Toronto Centre for Phenogenomics indicates that a reasonably complete test for a page with multiple paths through a form can easily require upwards of 150 lines of Java code (not including comments). Here is one portion of such a Java file – the “before picture”:

```
public void testSave() throws Exception {

WebResponse resp = gotoMrisubjectDetailPage(wc);

WebForm form = resp.getFormWithName("pageForm");

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow0_table1TDRow0_box1_displaybox1_ExperimentN
ame", “2”);

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow4_table1TDRow3_box4_displaybox4_MouseID",
“3”);

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow0_table1TDRow0_box1_displaybox1_OrganKey",
“Brain”);

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow0_table1TDRow0_box1_displaybox1_personName"
, “22”);

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow0_table1TDRow0_box1_displaybox1_ScanType",
“T1”);

form.setParameter("htmlPageTopContainer_pageForm_table1
_table1TRRow0_table1TDRow0_box1_displaybox1_Status",
“Living”);

SubmitButton button =
form.getSubmitButton("htmlPageTopContainer_pageForm_table1_table1TRRow0_table1TDRow0_box1_displaybox1_save");

resp = form.submit(button);

assertNotNull("Save and Return to MRI Scan list page", resp);

assertTrue("Save and Return to MRI Scan list page", (resp.getText().indexOf("/Jsp/experiment/mriSubject_list.jsp")>-1));

}
```

To provide a more useful alternative, I went back to the notion of "little languages" and back to the origins of HtmlUnit and HttpUnit; these both use the Apache Jakarta Commons HttpClient library. I designed a “little language” specifically for use in web testing. In PageUnit, the above Java code can be replaced by the following lines of PageUnit script, the “after picture”:

```
// Assume we have gotten to the MRI Subject Detail Page

F pageForm

R ExperimentName 2

R MouseID 3

R OrganKey Brain

R personName 22

R ScanType T1

R Status Living

B save

S
```

```
    M Save and Return to MRI Scan list page
    M /Jsp/experiment/mriSubject_list.jsp
```

As can be seen, the input language is much simpler and easier to use, and also provides a better fit to the "level of abstraction" that is appropriate for writing web tests.

The remainder of this report discusses:

-the notion of "little languages" with emphasis on the UNIX environment – the context in which they arose and in which Bentley (see below) described them originally;

-the background of software testing;

-the design and implementation of PageUnit; and further directions.

A copy of the User Guide is attached to this report as Appendix A.

2.0 Little Languages

Introduced in his *"Programming Pearls"* column in the August 1986 issue of *Communications of the ACM* , Jon Bentley's paper on this topic [Bentley1986b][1] has been seminal to an entire literature of "little languages". As Bentley starts off:

> When you say "language", most programmers think of the big ones... In fact, a language is any mechanism to express intent, and the input to many programs can be viewed profitably as statements in a language. This column is about those "little languages"

Also called "Domain Specific Languages" (DSLs), these languages are intentionally kept simple to facilitate ease-of-use over generality.

Little Languages are often contrasted with general-purpose programming languages such as Java and its relatives, based upon the presence or absence of programming language features required for generality. All general-purpose programming languages are believed to be Turing-equivalent, meaning (a simplification for purposes of this discussion) that you can translate a

1 In a footnote at the end of his paper, Bentley credits Mary Shaw of Carnegie-Mellon University as the person who introduced *him* to the importance of "little languages".

program from any general-purpose programming language (GPL) into any other GPL. This is not the case for DSLs.

The smaller Little Languages also lack features such as general-purpose input-output; whatever input/output they perform will be done as one of their predefined tasks. They often also often lack scripting capabilities such as the capability for boolean logic.

An analogy may help clarify the distinction between DSLs and GPLs. The English language can be used in a tremendous variety of ways. One can write in various literary forms such as poetry, drama, fiction. One can write factual accounts ranging from computer software user manuals to news reports to M.Sc. theses to books on politics, psychology, the environment, and all the other subjects that fascinate book-lovers everywhere and keep authors, book-sellers and librarians employed. These are examples of general-purpose uses of the English language – words combine in almost every imaginable order - though forms such as poetry and drama have slightly more specific rules.

Or, one can write a last name, first name, street address, some dots, and a telephone number. And repeat this pattern a hundred thousand times with only minor variations, and call it a telephone directory ("white pages"). That is a little language, quite domain-specific, the grammar of which is spelled out almost in full in the first sentence of this paragraph.

Some Little Languages

Bentley's examples include scripting languages such as the UNIX shell [Bourne1978], the Awk language, and the batch text-formatting languages PIC, SCATTER and CHEM. The latter three were written at Bell Laboratories in the 1980's as front-ends to the powerful but somewhat cryptic *troff* text formatting engine [Ossanna1976] which is a featured part of all UNIX-like operating systems distributions. In fact, Scatter and Chem were written as front-ends to *pic*, which is itself a front-end to *troff*[2]. The notion of front-end-to-front-end is essentially the same as the original UNIX notion of pipes and filters [Ritchie1977], in which a number of small programs is combined using the "pipe" operator (denoted by the "|" character in the UNIX shell and elsewhere); each of these programs is supposed to "do one thing well" [Gancarz1995]. The

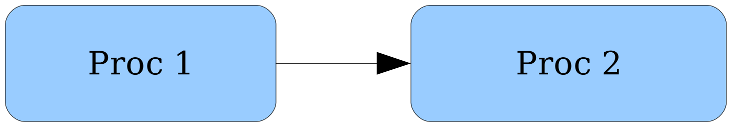
2 Troff itself is a little language (with a very big implementation) for text formatting; in common with HTML, a troff file both contains the content and specifies the formatting (but in much more detail than HTML!).

standard output of one program becomes the standard input of the next program in the pipeline. A common example is the following:

```
grep -i 'ian.*darwin' somedatafile | sort | uniq | more
```

The grep (file content search) program itself implements a little language, known as Regular Expressions [Friedl2004], to specify the pattern to match, in this case, “ian” followed by any number of arbitrary characters, followed by “darwin” (all on the same line)[3]. Lines matching this are copied from the file “somedatafile” and written to sort, which puts them in alphabetical order. This ordering is to satisfy the requirement of the next program, uniq, that identical lines be contiguous in the file (although the exact order does not matter for this purpose). Uniq, as its name implies, prints out only lines that are unique[4].

Pic is a simple language for drawing box diagrams like this:



A drawing like the above could be produced using an input file similar to the following:

```
.PS
box "Proc 1"
arrow right
box "Proc 2"
.PE
```

3 The “regex” syntax originally came from the Kleene Algebra (invented by Stephen C. Kleene; see http://en.wikipedia.org/wiki/Stephen_Kleene); they were first applied to text, as far as can be determined, by UNIX co-author Ken Thompson.

4 Uniq does not, as its name also implies, truncate text by leaving off trailing vowels; the propensity of UNIX folk to shorten command names to minimal recognizability/uniqueness is an artifact of the system's early development in a world in which “high speed terminal” often mean “1200 baud” (see [Ritchie1974]). The propensity to make program inputs and outputs very simple, however, derives from their common and expected use in pipelines [Ritchie1977]).

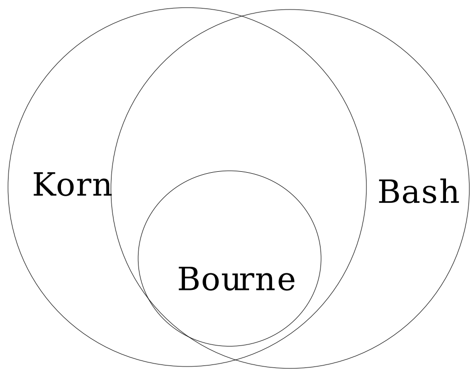
It may seem odd to write a language to *describe* drawings rather than just drawing them, but remember that PIC was developed in the late 1970's, and Scatter and GRAP in the early 1980's, before the IBM PC and the Mac brought low-cost drawing devices to every desktop. Bentley does mention interactive drawing programs, and admits that there are situations where they are the right tool; I'd expand this in today's context by saying that no sensible person would try to use PIC and Grap to design a magazine cover. As well, there are cases where the input to such processors will be generated from another data source, and one does not want to manually update the drawing file each time the data change.

The major "typesetting little languages" (troff preprocessors) invented at Bell Laboratories include:

Name	***Reference***	***Function***	***Implementation***	***Output***
tbl[5]	[Lesk1979]	Tables	C, yacc, lex	troff
eqn	[Kernighan1975]	Equations	C, yacc, lex	troff
Refer	[Lesk1978]	Bibliographical references	C	Troff with macros
Pic	[Kernighan1984]	Pictures	C	troff
chem	[Bentley1986a]	Chemical formulae	Awk (500 lines)	pic
grap	[Bentley1984]	Graphs, directed and otherwise	?	pic
scatter		Scatter plots	Awk (24 lines)	pic
Graphviz	[GraphViz]	Graphs, directed and other	Various	Direct graphic output (PostScript, displays, e

Scripting with Shells

5 Tbl is a reasonably small language but the implementation is large and complex due to the very low level drawing primites in troff, and full of undocumented depencies thereupon. I spent most of one summer in the 1980's trying, with David Slocombe of SoftQuad, to adapt it to our revised implementation of troff; at the end we agreed that a rewrite of the innarts might have been simpler.



The UNIX "shells" or command interpreter is among the oldest widely-used scripting languages. The Bourne shell "sh" [Bourn1977], Korn shell "ksh" [Korn1993] and "Bourne Again Shell[6]" bash [Ramey2003] use essentially the same syntax for variables, flow control, and so on (see diagram). Another family of shells (csh, tcsh) uses a different syntax that is somewhat reminiscent of the C programming language, hence the name. In practice the sh-family shells have proven superior to the csh-family shells for scripting purposes.

These shells offer a "glue" for combining other UNIX programs (including other shell scripts). To take an example from the UNIX typesetting domain, suppose I have a file that I change often and which uses several of the pre-processors mentioned above. I might type

```
tbl report.t | eqn | troff -Tps | dps | lpr
```

Tbl, *eqn* and *troff* have been discussed earlier. *Troff* produced a generic output format[7] which required device-specific reformatting; *dps* produced files for a PostScript™ printer, and *lpr* sent the file to the system's line printer.

6 Arguably one of the worst puns in UNIX history.

7 There are actually several versions of troff. The eldest, described in [Ossanna1976], produced only output for the Graphic Systems Inc. C/A/T wet-process phototypesetter; the University of Toronto produced an emulator for the Varian roll printer. The second major version was "device independent troff" or ditroff [Kernighan1982], the version described here. SoftQuad Inc. (my employer at the time) built a new version based on ditroff but featuring easier-to-parse output and some additions to the input language to ease the job of writing pre-processors. Modern Linux and BSD systems ship with groff, a fourth version, a complete re-implementation by James Clarke, incorporating features from the other versions (I was instrumental in helping him at the time to obtain permission to copy SoftQuad's input language extensions).

A user might type this incantation once or twice, but after that, matters of efficiency suggest that it be automated. Putting this exact invocation into a text file called, say, *printreport*, would often mark a user's entry into the world of shell scripting[8].

When one has a second text file that requires similar programming, an experienced computer user will want to abstract out the functionality, so the shell script becomes

```
tbl $1 | eqn | troff -Tps | dps | lpr
```

The $1 is substituted by the first positional parameter on the command line used to invoke the script, so if I say

```
printreport report42.t
```

then the system will behave as though I had typed

```
tbl report42.t | eqn | troff -Tps | dps | lpr
```

Refer to [Bourne1977] for more information on shell programming. A detailed summary of the UNIX formatting languages appears in [Akkerhuis1998].

Make [Feldman1979] is a build tool that is widely used by C programmers; its input is a very little language comprised primarily of file names that need to be built (targets), files that they are built from (dependencies), and the shell commands (as in a shell script) needed to ensure that the targets are up-to-date. For example to ensure that a program named *report*, generated from C language files report.c and report.h, is up to date, one only need run the *make* command in a directory where the source files and this *Makefile* are stored:

```
report:   report.c report.h
     cc report.c -o report
```

If the report program is up-to-date (file is newer than its dependencies) then *make* does nothing, else it runs the cc (C Compiler) step to build the program.

There are a number of “little languages” that have grown to be full-function languages. Awk, named in the mid-1970's for the initials of its three authors Aho, Weinberger and Kernighan, is the best known. The “new awk” of 1979 introduced a variety of new features, and the GNU

8 Also called shell programming; the term “shell scripting” sounds less threatening to the non-programmer.

version added even more. Awk has been used to write document format converters, accounting systems and even text formatters. Awk's original motivation was to try to create a language that would incorporate the functionality of many simpler utilities such as the stream editor *sed,* with a scripting functionality reminiscent of C's syntax.

Larry Wall's scripting language Perl took this one step further, incorporating all the functionality of awk along with such UNIX tools as sort. Perl today is used for a wide range of tasks, including systems administration on both UNIX and MS-Windows, and for building web sites, where it competes head-on with Java, PHP and other web-site packages.

Python, Ruby and Groovy are some recent scripting languages; all of them (with Perl and probably Awk) believed to be Turing-complete. Groovy builds on the success of Java to make an easier-to-use scripting language that both runs on the Java Virtual Machine; Groovy scripts can also be compiled in to Java “class files” to get most of the efficiency of compiled Java.

There are DSL's from many other domains, too many to list them all. The following section lists some domain languages from the mathematical/scientific/statistical computing area.

DSL's for Math/Science/Statistics

The programming world provides numerous examples of specialized languages for mathematical, scientific and statistical processing, including the following. While these are “domain specific languages” they are for the most part no longer “little” languages

Name	***Origin***	***Domain***	***Reference***
Maple	Maple Inc.	Mathematics	http://www.maplesoft.com/
Mathematica	Wolfram Research	Mathematics	http://www.wolfram.com/
S, R	Bell Labs/ R Project	Statistics, graphing	http://cm.bell-labs.com/cm/ms/departments/sia /history.html http://www.r-project.org/
SAS	SAS Inc	Statistical analysis	http://www.sas.com/

Name	*Origin*	*Domain*	*Reference*
Speakeasy	Stan Cohen, Argonne Labs	General mathematical/statistical analysis	[Cohen1977]; http://www.speakeasy.com/
SPSS	SPSS, Inc	Statistical analysis, originally aimed primarily at the Social Sciences	http://www.spss.com/

XML is not a little language

One technology that is sometimes offered as a little language is XML [Bray1998], the eXtensible Markup Language. It was plausible to describe XML as a little language in its early days; it was, after all, created as a reaction to the complexity of its parent standard SGML [ISO8879]. However, the intervening years of feature-accretion mean that XML is no longer a "little" language. Consider the XML Schema Standard and the Web Services Description Language as non-little languages (though not necessarily large in functionality). As well, XML was not intended to be used as a human-generated input language; it is designed to be read and written by computer software. This has not stopped it from being used in this way, however; it is used for many configuration file such as those of the Java Enterprise Edition. This is acceptable if the configuration files are generated by GUI, but places a burden on the administrators if they are expected manually to edit such files. XML is used as the input language to the Canoo Web Test package, which in my opinion makes it too complex to write tests in a reasonable time. This observation was one factor that lead to the initial planning for PageUnit.

Implementing Little Languages

There are two major ways of implementing little languages: as standalone programs or by extending an existing language.

The majority of existing "little languages" are implemented as standalone programs – I call this

“Plan A” - written in a general-purpose programming language such as C or Java. As we have seen with the text formatting languages *tbl* and *eqn*, the implementation is often assisted by use of lexical and grammar generators such as YACC (Yet Another Compiler Compiler, [Johnson1975]) and lex ([Lesk1975]).

Recently there has been attention given to implementing little languages by adding simplification or extension mechanisms to existing languages ([Hudak1998], [Stevenson2005], [Theodoru2006]). I call this “Plan B”. The advantage is that one can reuse a considerable amount of language extrastructure such as the base language's I/O system. The downside – and I think it is a major one - is that anybody wanting to use the new “little language” must first learn some or all of the syntax of the base language, or at least, be exposed to its linguistic quirks.

A third approach, which I shall call “Plan C for Compromise”) is to write in some GPL a standalone interpreter for your simple language (as in Plan A), and allow extensibility by programmers using the same base language. This is in some sense similar to the extension mechanism provided by Java Applets [Sun1995]; these are usually written using a fairly small subset of the Java API, but have most of Java's capabilities available. On the server side, JavaServer Pages™ use HTML and some special syntax but allow insertion of arbitrary Java code for special effects. Further back in time, the SpeakEasy system [Cohen1976], written in Fortran, allowed developers to write “Linkules” in Fortran.

The advantage of this approach is that the normal user of the little language has only to learn the basics thereof, yet all of the features of the “main” language, such as I/O, logic, etc. are available to programmers writing extensions to the new “little” language.

Summary of Little Languages

We have now discussed the general idea of little languages in some detail, illustrated with a variety of little languages and domain-specific languages. An overview of various “little languages” is in Volume III of the *Handbook of Programming Languages* [Salus1998]. The first chapter is a reprint of Bentley's classic paper (*op cit*).

We now turn to the general notion of software testing.

3.0 Software Testing

History & types of Testing

Software has needed testing since the earliest days of computer programming. In the early days this was left to the programmer to do in his/her “spare time” and, as a result, there was never enough testing performed[9]. An early classic in this field, recently reissued, is [Myers1979]. The basic approaches to testing include “black box” testing – in which you test a piece of software in isolation, either not looking inside or pretending not to know its internals – and “white box” testing in which it is permissible to look inside the software being tested[10]. The basic purposes of testing include Unit Testing and Functional Testing. Unit Testing tests individual pieces of a system; functional testing tests the overall system, and is often a manual task. PageUnit focuses on unit testing, which is discussed in more detail.

Unit testing

Regression Testing means running a piece of software and comparing its outputs with the expected outputs to ensure that no “regressions” (reversions to non-working state) have been introduced into the code. Unit Testing is the best-known form of Regression Testing. In Unit Testing, each “unit” of code is tested separately. For Java code, this normally means testing each significant class as a unit (see example below).

Test Driven Development or TDD ([Beck2002], [Ambler2003]) has become popular lately. Also known as “Test First Development”, this involves writing the test before you write the code that it tests. This at first seems a curious inversion; old-time developers first hearing this often ask “How can you test code that isn't written yet?” But TDD is ready with the answer: “How can you write code when you're not sure how it should perform?” In other words, only when you sit down and design tests for a method do you really have the detailed understanding of its

9 There probably still isn't. Jonathan Fuerth, at my consulting client SQLPower.ca, noted (company meeting, January 2006) that in the “good old days” managers begged programmers to write tests; now programmers must beg managers for the time to do so.

10 Probably “open box” or “clear box” would have been a better term, but it's too late to change the terminology now.

behavior necessary to implement it correctly. See [Darwin2005] for a brief outline of using TDD in conjunction with JUnit (see below) and Eclipse.

One of the thornier aspects of unit testing is knowing *what* tests to write for a given method. As it is a non-halting problem to verify all possible inputs and outputs, one must choose a “reasonable” set of “reasonable” input values and verify that they produce correct outputs (which are known in advance, by some means other than running the program). One must also verify that some non-reasonable values are correctly detected for error handling, if appropriate[11].

Test coverage is the measure of how much of a given code base is actually tested by a given test suite. There are tools that can measure this for common languages and common testing approaches. One doesn't normally require 100% test coverage, however. Simple setter/getter methods, for example, are usually written by an IDE and, if they don't contain any validation logic, not only do not need testing but there are usually so many of them that testing them would lead to a noticeable slowdown of the overall testing process.

It is also important to know *when* to test. The best general advice is the old adage “Test early and often”, by which is meant that one should begin testing while the software is being developed, and run all the tests as often as possible (see [McConnell1996]). In fact, *Continuous Integration* [Fowler2000] recommends dedicating a computer to the purpose of building and testing the software; this machine would periodically obtain the current source code from the source code librarian (such as CVS [Berliner1990] or Subversion [Collins-Sussman2004]; regular and consistent use of such a librarian is a requirement for this process), compile everything together, and run all the unit tests against all the code. This should happen as frequently as every hour, to ensure that all changes made by all developers can be integrated (hence the term Continuous Integration). This technique has been used by Microsoft (with at least daily builds) [McConnell1996] to reduce the number of bugs in software such as Microsoft Windows™ and Microsoft Word™, and is in regular use by many smaller organizations.

11 Obviously not applicable to continuous cyclical math functions such as sin(), cos() for which any input angle produces a valid return that can be tested against known mathematical tables, but such functions are the minority.

Unit Testing with the JUnit Framework

Unit testing has become very popular in recent years. A family of xUnit frameworks has been developed for programmers in various languages: CPPUnit for C++, JUnit for Java, and so on (see http://www.xprogramming.com/software.htm. part way down the page, for a list of xUnit frameworks, and especially http://www.junit.org/ for JUnit). JUnit has quickly become the industry standard for Java programmers, and most open source Java programmers rely on it. The popular Eclipse IDE has both a “wizard” for generating the skeleton of a test program (and another for maintaining a runnable list of test programs), and a “test runner” that shows graphically the status of the tests: a gauge control that shows green as long as all tests pass, and turns red as soon as one test fails. The JUnit package provides similar “test runners” for standalone use; their web site features the slogan “Keep the bar green to keep the code clean”. The Eclipse JUnit test runner also presents a back trace for each failure, and lets you double-click on any entry in the back trace to jump directly to the line of code which contributed to the failure. Eclipse and JUnit have thus contributed to each other's success symbiotically.

Suppose that we were modifying, and therefore wanted to test, the *decode* method in the standard class java.lang.Integer[12]:

```
public static Integer decode(String value)
 throws NumberFormatException;
```

To test this one should, as mentioned, test that it performs correctly for a valid inputs but also that it fails gracefully for invalid inputs. A JUnit test might look like the following (but probably with more tests, possibly loaded from an array):

```
public class IntegerTest extends TestCase {
     public void testDecode() throws Exception {
          int ret;
          ret = Integer.decode("-42").intValue();
```

12 This method converts strings containing a number in a variety of formats (such as decimal, hexadecimal) to integers. Of course only Sun has permission to modify this class and distribute the changed version, but it is publicly available so that any reader can easily verify that my tests do test some of its functionality.

```
                assertEquals(-42, ret);
                ret = Integer.decode("-0x42").intValue();
                assertEquals(-66, ret);
                try {
                     Integer.decode("one two three");
                     fail(
                     "Didn't throw expected exception");
                } catch (NumberFormatException e) {
                     System.out.println(
                     "Caught expected exception");
                }
           }
      }
```

Once this method has been written, it becomes a permanent part of the source code of the project. One can run this test as part of a test suite to ensure that no regressions have been introduced into the code. With Eclipse, you just press the Run button again to re-run all the tests.

Web Testing Modalities

Web testing can be approached from many perspectives, including HTML syntax checking and browser testing, link checking, interaction functionality (regression or unit testing), responsiveness under load, and even spell checking.

HTML Syntax checking means that a given page conforms to a standard for HTML syntax, e.g., conformance to XML syntax “XHTML” or the XML-like syntax of HTML4.x, without regard for the actual content. This is a necessity to ensure that the page has a good chance of rendering correctly in the abstract, and on most browsers. Browser Testing means actually testing a page on five or ten different browsers, including some on different platforms. The latter is necessary because of the different character sets used by different operating systems, as well as because some browser vendors such as Microsoft put out very different versions of the same browser on, say, Microsoft Windows versus Mac OS X (Microsoft have, in fact, given up on trying to support Mac OS). Syntax checking can be done easily using an HTML parser, whereas browser testing requires a more complex "robot" setup with various computers (physical or virtual) driven from a test program which either validates the results (regression testing) and/or obtains screen captures for a remote user to verify the correct visual results. An interesting "torture test" variant on browser testing is in [Zalewski2004].

Link checking means, as the name implies, verifying that all page links on a given page actually exist. Link checking is a necessity due to the ongoing maintenance of the user's own site as well as the fact that external web sites frequently get re-organized and not infrequently either get renamed or even disappear altogether. Link checking is among the easiest of these modalities to automate because one only needs to extract '<a>' and "<form>" elements from a page, and extract and open each link URL. Java code to implement a simple link checker appears in Chapter 18 of my *Java Cookbook* [Darwin2004b]; a simplification of this code has recently been added into PageUnit.

Measuring responsiveness under load is undertaken to ensure that the web site will perform well once a significant number of users begins accessing the site. In earlier eras of both mainframes with attached terminals, and dial-up timesharing, it was easy to predict the maximum load on a computational system, because the maximum number of users was determined by hardware. In the age of the web, there is no feasible absolute maximum number of users, which raises an interesting question for IT managers: is a sudden increase in web activity a sign of being noticed by consumers due to mention on a prominent website (the "slashdotted" phenomenon[13]) or is it a sign of a "distributed denial of service" attack?

13 SlashDot.org is a UNIX geeks' web site, named after "/.", a roundabout reference to the root of a filesystem; filenames or URLs with /. in them (or /..) are often used to bypass simple security checks. SlashDot has tens of thousands of subscribers and its news section is updated frequently during the day, so when an interesting site is mentioned therein, huge numbers of readers may visit it in a matter of minutes, resulting in the site being "slashdotted".

Interaction functionality testing (also known as regression testing or unit testing) must be undertaken to ensure that a site's functions work correctly. For example, to test a page that registers a new customer's name and address, we must pretend to be a user at a browser: obtain the HTML form, provide the parameters to the form, and submit the form. We must then retrieve the results and verify that a “successfully registered” message appears; we should also verify that the data were actually inserted into the database. The former can be automated and is the domain of my research, PageUnit. Requirements for this type of testing are discussed next. Data insertion verification would require external access to the database, and is being considered as an additional functionality to the program.

Requirements for Web Unit Testing

In order to unit test web pages, one needs basically a scripting language and the following capabilities:

Fetch an HTML page containing a form;

Match title, or arbitrary element, or text anywhere;

Find a form (by name or position; page can have multiple forms);

Set form inputs/parameters by name and value;

Submit the form (optionally with a button name; forms can have multiple buttons)

Then use match/find to verify the page submitted correctly; maybe continue with next form!

This implies a need for the following underlying mechanisms:

- run a web-based login process if required by the site, so it can be tested in a realistic mode;
- navigate the complexities of the HTTP protocol, including sessions and cookies;
- process the vagaries of HTML pages.

How PageUnit accomplishes these goals is discussed in the following pages.

Design & Implementation of PageUnit

Rationale

The most obvious question is "why another web test framework"? There are many to choose from already (one of many lists of same is [Hower1996]). The simplest answers I can give at this point are: because I thought I could do better, and because I thought it would be a good research project.

The explicit design goals of PageUnit are to provide an easy to use (with reference to, e. g., the section "XML is not a little language" above) facility for unit testing of web pages. That is, it does not perform spell checking, XHTML validation (in fact, it uses rather a lax parser, because not all HTML is well formed by any means), nor load testing. That said, it must be easy to:

- Create new tests or modify existing ones;

- Change the server host name, HTTP port, or "context" (directory prefix) for an entire run with only a single change; these configuration sets should be able to be stored in a simple text file for re-use;

- Run the program repeatedly, both manually and automatically, and

- Interpret the output.

These goals have guided the design and implementation of the framework.

Design of the Input Language

The input language to PageUnit consists of single-letter commands, one of which must be the first character on each input line. For example, to go to a given "unprotected" (no login required) **p**age, say xyz.html, the command is

```
P /xyz.html
```

(assuming you had already specified the server and host in one of several ways). To focus on the HTML form named CUSTOMER in the resulting page, you'd use

```
F CUSTOMER
```

To set the parameter “CUST_ID” to 12345, the command would be

```
R CUST_ID=12345
```

Then the command

```
S
```

would submit the form.

It it were desirable to check for a given string or pattern in the page, the Match (M) command is used. Patterns are specified as regular expressions. To match the string “Toronto Centre for Phenogenomics” in the current page, one would use

```
M Toronto Centre for Phenogenomics
```

The choice of whether to match in the page loaded by P or the results page fetched by S is determined simply by placing the M directly after the P command or after the S command.

Conspicuous by its absence is any form of traditional boolean logic (“if statement”). Such a mechanism is not only not needed, it is a complication that goes against the spirit of “keep it simple”. Clearly if loading a page fails, the framework needs to skip until the next page is loaded; this is not implemented today (so cascading failures do appear), but this “skip to next page” logic will be implemented shortly.

A complete list of the commands is shown in Appendix A of this report.

Sample Inputs are in the source distribution under the “demos” directory.

Organization of the code

The source distribution consists of one directory with the following structure. This directory can be treated as an Eclipse project, which will build all the files to run under Eclipse. There is also an Ant build.xml[14] file that can be used to build all the files and also can build a Jar file of the framework itself for use in other projects.

Name	***Content***
demos	Samples and demonstrations

14 Not necessarily as up-to-date as the Eclipse project structure.

Name	*Content*
docs	documentation
lib	Directory for JAR archive files.
scripts	Contains one shell script to run the framework (may not be up to date)
selftest	Top of self-test (JUnit unit tests)
src	Top level of Framework source code

The source itself is organized into several Java packages:

Name	*Functionality*	*Classes*
pageunit	Main classes	PageUnit (main), ScriptTestCase (JUnit) TestFilter, TestUtils
pageunit.html	Web Pages	HTMLParser, plus an interface and a cla for each of a dozen HTML elements
pageunit.http	Web Server Interaction	Session, WebResponse, ...
Play	-	Test or demo code that is not in use at present.

Parsing User Input and Running Tests

One of the benefits of the input syntax described earlier is its ease of parsing. In ScriptTestCase, as lines are read from a test file, empty lines are discarded[15]. Then the first character of the line is stored in a variable called 'c', which is used to control a switch statement. Actually there are two switch statements. The first is used to look after simple administrative requests such as file

15 This is a simplification; in fact it makes one pass reading all lines into a List, for purposes of the JUnit method countTestCases, and walks the List during the run() method.

inclusions, adding and removing test filters (see “Extensibility” below), and setting general parameters such as user name and password.

The second switch statement, enclosed in a try/catch, is used to run one test operation such as getting an HTML page, looking for a form, setting forms parameters, submitting the form, and looking for text in the current page. This part is inside a Java try/catch so that any errors that occur will only abort the a single test command rather than the entire run; errors caught here are simply reported (see below).

Many of the tests are written using the various assert() methods from the JUnit unit test framework. Here they are being used as convenience routines, since the assert() methods will throw an exception including the text of a message if the assertion fails. For example, the JUnit statement

```
assertNotNull(“finding form”, currentForm);
```

is short for

```
if (currentForm == null) {
throw new Exception(“Failure occurred in “ + “finding
form”);
}
```

There is more about ScriptTestCase in the discussion of output reporting, below.

Parsing HTML Pages

Any web testing framework must be able to parse HTML files in order to ascertain the presence or absence of selected form elements; this is in fact one of the most important implementation pieces of a web testing package. An early implementation of PageUnit used HtmlUnit [Bowler] which provided this service, but I dispensed with that API because it both did too much on its own and provided too little direct access to the information I most needed for testing.

If it could be known that all the HTML will be valid or at least well-formed[16], one would use a

16 Well-formedness indicates conformance to the XML syntax described in [Bray1998] – correct use of angle brackets, tag nesting, etc. - whereas validity indicates conformance to a grammar such as the W3C XHTML or

full-function XML parser such as that included in the Java Standard Edition[17] (in the packages javax.xml.parsers, org.w3.dom and org.xml.sax packages). Unfortunately, in the general case one can make no such guarantee: many web sites that one might want to test consist largely of hand-made HTML that is full of syntax errors. Indeed, code generated by many web application frameworks may contain HTML syntax and/or semantic errors[18]. Accordingly, in parsing HTML in a web test framework, one simply has to be prepared for the worst.

HTML/XML parsers can be categorized as "strict" or "lax" depending upon how rigidly they adhere to the specifications; in other words, how they will respond to invalid syntax or semantics. To be able to parse arbitrarily bad HTML, of course, one wants a very lax parser. The choice then is whether to use an existing parser, or to build one's own for this purpose. Most of the existing HTML parsers either are strict (like Java's built-in XML parsers) or, like *HTML Tidy* [Raggett] are designed to "clean up" poorly-formed HTML. For my purposes in PageUnit, these tools were excessive.

Thus I was down to updating my simple tag extractor (Section 18.9 in [Darwin2004b]) or finding a very lax parser. Fortunately, I did a bit more research before coding and found that there is an HTML parser hidden inside the Swing GUI functionality of the Java Standard Edition. The javax.swing.text packages contain an HTML parser that is very forgiving; it can be used for formatting simple HTML pages for display, and is also used internally for parsing HTML used in labels and other Swing GUI components.

The Swing "HTMLEditorKit Parser" provides an easy-to-use interface and, because it's part of Java SE, requires no additional API ("Jar files") to be shipped along with Pageunit. To use the parser, one extends the class HTMLEditorKit.ParserCallback and overriding whichever of its methods one wants, and pass this ParserCallback to the ParserDelegator's static parse() method. ParserCallback methods include:

```
    public void handleText(char[] data, int pos)
```

HTML4.0 standards

17 Previously known as the Java 2 Standard Edition, or J2SE for short. The "2" has been dropped with the release of Java 5 / Java Development Kit 1.5.

18 As well, sites that make extensibive use of any web framework's "include" mechanism will often have more than one <title> element, which is also a syntax error.

```
    public void handleComment(char[] data, int pos)
    public void handleStartTag(HTML.Tag t, MutableAttributeSet a, int pos)
    public void handleEndTag(HTML.Tag t, int pos)
    public void handleSimpleTag(HTML.Tag t, MutableAttributeSet a, int pos)
    public void handleError(String errorMsg, int pos)
```

The class HTML.Tag implements an extensible type-safe enumeration ([Bloch2003]) of all known standard HTML tags. The first tag you see in a complete HTML file will be represented by the singleton HTML.Tag.HTML, the second by the singleton HTML.Tag.HEAD, and so on.

For example, to print just the names of all the tags in an HTML document, you would only need to override handleStartTag() and handleSimpleTag(), and have them both print their HTML.Tag argument. The file ParseSimple in the “play” package of the source distribution demonstrates this:

```
    new ParserDelegator().parse(reader,
    new HTMLEditorKit.ParserCallback() {


    public void handleSimpleTag(Tag t, MutableAttributeSet
    a, int pos) {
        System.out.println(t);
    }


    public void handleStartTag(Tag t, MutableAttributeSet
    a, int pos) {
        System.out.println(t);
    }
    }, false);
```

This simple HTML file is included in the ParseSimple source code:

```
    <html><head><title>Hello</title></head><body>
    <h1>Hi</h1><p><b>I</b> am <a>here</a>
    <br/><p>Text</body></html>
```

Running the program produces the expected list of tags:

```
    html head title body h1 p b a br p
```

Of course in a real parsing application there is more than just printing elements; the HTML elements must be stored in some sort of structure. I use the classes HTMLComponent and HtmlContainer to implement a “composite” pattern similar to that used for Component and Container in the Java AWT windowing system. There are subclasses of these to represent the HTML elements I am interested in, such as Title, Form, Input, and so on.

In my HTMLParser class, methods pushContainer, curContainer.addChild, and popContainer are used to create a simple tree representing the containment structure of the HTML elements in the input page. This tree is consulted by code in the ScriptTestCase class.

There is also a “last resort” method, WebResponse.getAsString(), which makes the entire content of the current page available as a String. This is used by some PageUnit commands such as Match, as described in the next section.

Variables

Whilst initially I thought I would not use variables in my “little language”, they have proven useful in two contexts so I have been persuaded to add a general variable mechanism, and to revise some of the code to use them in a general and consistent fashion. Variables are set with the '=' command, that is,

= siteName Toronto Centre for Phenogenomics

creates a variable called siteName with the organization's name as its value. Because variable substitution is done very early in the parsing, it can be applied to any command. For example, the Match command given earlier can now be simplified to

M ${siteName}

The Match command also sets one or more variables; the entire matched string is place in the variable M0, and if the regular expression contains “capture groups”, the matching substrings will be placed in variables M1, M2, ... M*n*. These can be used in subsequent commands such as R to set the matched primary key of one submission as the search parameter of the next. This has proven useful in writing tests for multi-page HTML form dialogs.

Initiation of each file pre-sets several “global” variables such as HOST, PORT, USER and PASSWORD from the configuration file; these can be overridden by various commands in the script.

Reporting

The previous section has discussed how HTML elements are parsed and stored as tests are being run. This section briefly discusses the handling of errors that occur during these tests. In the terminology of the JUnit framework, there are two main categories of problems that can occur, errors and (web-specific) failures. Runtime Errors (generally, *RuntimeException* and its subclasses in Java; errors detected by the Java Virtual Machine, such as *NullPointerException* and *InvalidArgumentException*) are usually caused by errors in PageUnit itself, that is, incomplete validation of user input or bugs in some supporting class. These must be found, ferreted out, and fixed. There is a growing package of JUnit tests that tests the PageUnit framework itself, in the *selftest* package, that tries to check for these.

The other major category - failure - is actual errors caused by web pages. These in turn divide into I/O errors and web site failures. I/O errors include sites that time out and missing pages (caused by broken links and by typographical errors in tests), but also includes missing or mis-named test files. Web site failures – generally, actual test failures - are reported when the code in *ScriptTestCase* detects that a user test has failed and throws an *AssertionFailedError* (from the JUnit library).

All these errors are caught in the framework. At present they are reported on the standard output (along with a still-too-large volume of debugging output) and a summary of the different types of errors is reported at the end.

Errors and Failures are also reported back to JUnit by use of the Junit-provided TestResults class. This allows a graphical test-runner such as the JUnit GUI ScriptTestCase or the in-built Eclipse test runner to display graphically -in gauge format - the progress through the testing and – by color – the status of the overall testing. Green means all tests have passed so far; Red means there has been one or more failures.

Because the test class pageunit.ScriptTestCase behaves like a “Test Suite” style of JUnit test, it can be integrated into JUnit tests along with those for the Java code (if any, such as Servlets) in the overall web site development system.

Extensibility

Java makes it easy to provide an extensible framework using “Plan C” discussed above. The decision made early on by Java's originators to make Java's “reflection” API [Darwin2004, Chapter 25] available to the average programmer must be seen as one of the most important API decisions from the early years. Developers in languages such as C and C++ had needed this capability for years; Java not only delivered it, but did so portably and in a standardized way. The Reflection API makes it possible to write tools as diverse as reverse engineering tools, API cross-reference programs, and implementations of the Strategy or Plug-In design patterns.

The PageUnit framework therefore happily offers a simple plug-in mechanism. A plug-in need only implement the *pageunit.TestFilter* interface, which has one method in it:

```
public void filterPage(HtmlPage thePage, WebResponse
theResult) throws Exception;
```

The plug-in is invoked early in the list of tests; if the plug-in throws a Java Exception, the listed test will not be invoked for the given page, the Exception will be printed (but not a stack trace!), and the page will count as a failure. As an example, the provided filter *sample.CvsIdFilter*, which implements a sample policy that all pages must contain a CVS identifier, looks like the following:

```
public void filterPage(HtmlPage thePage,
     WebResponse result) throws Exception {
```

```
     String contentAsString =
result.getContentAsString();
     if (!(contentAsString.indexOf("$Source") > -1) &&
          !(contentAsString.indexOf("$Id") > -1))  {
          throw new RuntimeException(
               "Page does not have a CVS Identifier");
          }
}
```

You can use more than one plug-in; they will be invoked in the order encountered.

The code that, using the Reflection API, instantiates the user's plug-in (which is assumed to be somewhere on the CLASSPATH at run time), given the class name, is about like this:

```
Object o = Class.forName(className).newInstance();
if (!(o instanceof TestFilter)) {
   throw new IllegalArgumentException(
     className + " does not implement TestFilter");
}
TestFilter usersFilter = (TestFilter)o;
filterList.add(usersFilter);
```

5.0 Issues, Conclusions and Further Development

Work So Far

The PageUnit framework as it stands in January, 2006 is usable for testing a wide variety of web pages, and is being used regularly to provide regression testing on sites written using PHP [PHP], the Java frameworks SOFIA and Struts [Darwin 2004a], and plain HTML. However there are some limitations to the present design and implementation which leave room for additional effort in order to make it as useful as desired. The minor irritants are listed in the source code under the *TODO* file in the *docs* directory; major functionality steps are shown below.

It is hoped that PageUnit can be developed into an ongoing open-source project, and permission to do so has been obtained from the Toronto Centre for Phenogenomics; the reader interested in the latest work should check the website http://www.pageunit.org/.

JavaScript Interaction

Web pages of moderate complexity often use JavaScript-based code for one of these purposes:

- to update the remainder of a form based on choices made in e. g., a drop-down list (like setting the name of the “Postal Code” field to “Zip Code” if the Country choice is set to “United States”;

- to automatically submit a form once certain fields are filled in, so that the server-side component can re-display the form with changes made as above;

- to validate forms input.

For these reasons, an implementation of some or part of JavaScript ([JS98], itself a “little language”) will be required for PageUnit to be able fully to test web sites.

Output Formatting

The current version of PageUnit simply displays errors in plain text. It is desirable to provide a summary report, which will probably occur in several stages, each involving re-use of existing software:

1) Use the JUnit textual report formatter;

2) Use the JUnit graphical (Swing/AWT) runner;

3) Provide an Eclipse [Eclipse] plug-in [Gamma2003] to provide push-button invocation and graphical pass/fail notification.

Intelligent Dumping

It would be useful to be able to get a display of the request and response headers on a single request or series of request.

Database Verification

Although it is not formally part of the domain for which PageUnit was originally designed, it is useful in some circumstances to verify that a given value has been inserted or updated in a relational database as a result of invoking a web operation. I have some unpublished code at my disposal that is designed to simplify direct SQL access to such databases, so it should be possible in future to add this capability to PageUnit.

Reflection

In a sense my whole life as a programmer has been about “little languages”, although I didn't know it at the time. One of my very first steps in programming was a utility I co-wrote with Joel Troster at the University of Toronto Computer Centre to replace the cumbersome file-system utility programs provided by IBM's mainframe operating system MVT. Like PageUnit this program used single-character commands for operations such as file renames, deletions, and copying. I like to think that I've learned a few things about software design and development in the intervening decades, but I still like single-letter command names: they're easy to remember (e. g., you don't have to worry about “centre” vs. “center”[19]), they're easy to implement (a switch statement in C or Java), and, as in the original UNIX line editor ed, provide a limit of about 26 commands, which imposes some discipline on the design and implementation. As well, one not need to use external tools such as YACC and LEX; these and their Java counterparts require extra processing steps and extra libraries at compile time and/or at run-time.

In this project I have had to go back and examine the notion of “little languages” in more detail, and refine the boundaries between true “little languages”, “domain specific languages” which may be larger, and the languages such as Perl and Ruby that have grown into full-fledged programming environments in their own right. I've also had to formalize my sometimes-informal categorization of testing methodologies, such as unit testing vs regression testing.

Acknowledgments

The development of PageUnit and some of the accompanying research was funded by the Toronto Centre for Phenogenomics. The following applies to this portion of the funding:

> “This work is part of the Mouse Imaging Center (MICe) at the Hospital for Sick Children and the University of Toronto. The infrastructure has been funded by the Canada Foundation for Innovation (CFI) and the Ontario Innovation Trust (OIT). The research has been funded by an Ontario Research and Development Challenge Fund (ORDCF) grant to the Ontario Consortium for Small Animal Imaging (OCSAI).”

19 James Gosling, inventor of Java, is a Canadian, but the early versions of the BorderLayout component layout manager required the string “center” rather than “centre”. Later versions provided a series of compile-time-constants (such as BorderLayout.CENTER) to move the spell-checking from run-time to compile time.

Thanks to my manager John Cargill for believing that there would eventually be light at the end of this particular tunnel. Evan Despault provided invaluable feedback while serving as the program's first “real” user. Jonathan Fuerth of SQL Power Group and David Saff of MIT provided information on the design & operation of JUnit. Finally, I would like to express my gratitude to my academic supervisor Dr. Cathy French of Staffordshire FEUT for her continuing support in many discussions relating to this research.

Bibliography

Aho1985 Alfred Aho, Brian W. Kernighan and Peter J. Weinberger, *Awk – A Pattern Scanning and Processing Language: Programmer's Manual*. Bell Laboratories Computing Science Technical Report No. 118. Bell Laboratories, Murray Hill, NJ USA: June, 1985. The first public description of the Awk Language; when Awk was new enough and “little” enough to describe in 32 pages.

Akkerhuis1998 Akkerhuis, Jaap. *Essential Features of UNIX Formatting Languages*, Chapter 4 of [Salus1998].

Ambler2003 Ambler, Scott W. Essay on Test Driven Development. Unpublished; online at http://www.agiledata.org/essays/tdd.html.

Beck2002 Beck, Kent. *Test Driven Development By Example*. Addison-Wesley Professional, November, 2002. ISBN 0321146530.

Bentley1984 Jon L. Bentley and Brian W. Kernighan. *GRAP – A Language for Typesetting Graphs: Tutorial and User Manual*. Bell Laboratories Computing Science Technical Report Number 114. Bell Laboratories, Murray Hill, NJ USA: December, 1984.

Bentley1986a Jon L. Bentley, Lynn W. Jelinski, Brian W. Kernighan. *CHEM – A Program for Typesetting Chemical Equations: User Manual*. Bell Laboratories Computing Science Technical Report Number 122. Bell Laboratories, Murray Hill, NJ USA: April, 1986. The source code for the chem little language can be downloaded from http://www.netlib.org/typesetting/chem

Bentley1986b	Jon L. Bentley, *Little languages, Communications of the ACM*, 29(8):711--21, August 1986. This is the original "little languages" paper which is cited throughout the literature; available on-line (requires ACM membership or equivalent) at http://portal.acm.org/citation.cfm?id=315691.
Bentley1986c	Bentley, Jon and Kernighan, Brian. *Tools for Printing Indexes*. Bell Laboratories Computing Science Technical Report 128, October, 1986. The source code for this little language can be downloaded from http://www.netlib.org/typesetting/indexing.tools.
Bentley1988	Bentley, Jon L. *DFORMAT – A Program for Typesetting Data Formats*. Bell Laboratories Computing Science Technical Report Number 142, April, 1988.
Bloch2003	Bloch, Joshua. *Effective Java*. Addison-Wesley, ISBN 0201310058. I consider this to be among the very best books on designing object-oriented classes using the mechanisms that Java provides. The Typesafe Enumeration pattern described herein was implemented by Bloch at Sun and became the basis of Java 5's new class-like **enum** feature.
Bray1998	Bray, Tim *et al*. *Extensible Markup Language (XML) 1.0: W3C Recommendation*, February, 1998. First edition online at http://www.w3.org/TR/1998/REC-xml-19980210; latest version always online at http://www.w3.org/TR/REC-xml
Bourne1978	Bourne, Steven R. *An Introduction to the UNIX Shell*. November 12, 1978. Included with some UNIX distributions, and included in [V7Vol2A].
Bowler	Bowler, Mike. The HTMLUnit web site at http://htmlunit.sourceforge.net/ appears to be the only publication on this project.
Cohen1977	Cohen, Stan. The SpeakEasy Language. Argonne National Laboratories: 1977.

Collins-Sussman2004 Collins-Sussman, Benjamin; Pilato, and Fitzpatrick. *Version Control with Subversion* (also known as "the red bean book"); O'Reilly Media, Inc., June 22, 2004 ISBN 0596004486.

Berliner1990 Brian Berliner, *CVS II: Parallelizing Software. Development*, Proc. USENIX Winter 1990. Conf., January 22-26, 1990, Washington, DC,. Pp 341-352.

Darwin2004a Darwin, Ian. *Java Web Development Frameworks*, Certificate module in the M. Sc. Professional Computing Programme at Staffordshire.

Darwin2004b Darwin, Ian. *Java Cookbook*, second edition. O'Reilly Media: 2004.

Darwin2005 Darwin, Ian, *Test Driven Development with Eclipse,* talk presented informally to several small groups, online at http://www.darwinsys.com/training/.

Feldman1979 Feldman, Stuart. *Make--A computer program for maintaining computer programs*. Software-Practice and Experience, 9(4):255--265, Apr. 1979.

Fowler2000 Fowler, Martin. Continuous Integration. Undated paper online at http://www.martinfowler.com/articles/continuousIntegration.html. My citation of 2000 is based on its oldest appearance in The Internet Archive (http://web.archive.org/)

Gamma2003 Gamma, Erich and Beck, Kent. *Contributing to Eclipse: Principles, Patterns and Plug-Ins*. Addison-Wesley, October, 2003. ISBN 0321205758. Rather dated due to the rapid evolution of Eclipse itself, but it provides a good overview of the process and covers the principles of becoming an effective Eclipse developer.

Gancarz1995 Mike Gancarz. *The UNIX Philosophy.* Digital Press, Newton, MA 1995. ISBN 1-55558-123-4. Gancarz was not associated with the original UNIX developers but appears to have extracted a good summary of the "official" UNIX originators' thoughts on UNIX philosophy. Summarized at http://c2.com/cgi/wiki?UnixDesignPhilosophy.

GraphViz GraphViz, a package of graphical tools; "dot" interprets a little language for creating graphs. See http://www.graphviz.org/. See also my contributed demonstration at http://www.graphviz.org/Gallery/directed/unix.html

Hansson2004 *Ruby on Rails* (web site http://www.rubyonrails.org/)is a free, open-source web application development framework initially created by David H. Hansson out of several web applications he had coded in the Ruby language.

Hower1996 Hower, Rick. *Web Site Test Tools and Site Management Tools*. Online at http://www.softwareqatest.com/qatweb1.html.

Hudak1998 Hudak, Paul. *Domain Specific Languages*, appears as Chapter 3 of [Salus1998].

ISO8879:1986 ISO, The Standard Generalized Markup Language, ISO 8879:1986. A detailed timeline on the history of SGML is at http://www.sgmlsource.com/history/sgmlhist.htm.

Johnson1975 Johnson, Steven C. Yacc: Yet Another Compiler Compiler, Bell Laboratories Computing Science Technical Report Number 32.

Kernighan1975 Kernighan, Brian W, Cherry, Linda L. A System for Typesetting Mathematics. CACM, March 1975. Revised version appears in [V7Vol2A].

Kernighan1982 Kernighan, Brian W. A Typesetter-Independent TROFF. Bell Laboratories Computing Science Technical Report Number 97, 1982.

Kernighan1984 Kernighan, Brian W. *PIC – A Graphics Language for Typesetting: Revised User Manual*. Bell Laboratories Computing Science Technical Report No. 116. Bell Laboratories, Murray Hill, NJ USA: December, 1984

Lesk1975 Lesk, Michael A. Lex: A Lexical Analyzer Generator. Bell Laboratories Computing Science Technical Report Number 39.

Lesk1978 Lesk, Michael A. *(Refer:) Some Applications of Inverted Indexes on the UNIX System*. June 1978. Appears in [V7Vol2A]. This paper describes the bibliographical preprocessor refer, and an additional program lookall, which is clearly recognizable as the inspiration for the modern UNIX/Linux locate command. As well, the “inverted indexes” described here provided (via some refactoring by Ken Thompson) the origin of what is now known as “Berkeley DB” and its clones such as ndbm and GNU DB (gdbm).

Lesk1979	Lesk, Michael A. *Tbl – A Program to Format Tables*. January, 1979. Appears in [V7Vol2A].
McConnell1996	McConnell, Steve. *Best Practices: Daily Build and Smoke Test*. Appeared in *IEEE Software*, Vol. 13, No. 4, July 1996; online at http://www.stevemcconnell.com/bp04.htm.
Myers1979	Myers, Glenford, *The Art of Software Testing*, John Wiley & Sons; First Edition February, 1979, Second edition June, 2004. ISBN 0471043281, 0471469122.
Ossanna1976	Ossanna, Joseph P. Nroff/Troff Users' Manual. October, 1976. Reprinted in [V7Vol2A]. Ossanna was a long-term contributor to early UNIX but passed away in 1977.
Raggett	Ragget, David. HTML Tidy, a tool for cleaning up HTML. Published on the web by the W3C at http://www.w3.org/People/Raggett/tidy/ and now maintained at SourceForge.net.
Ramey2003	Ramey, Chet and Fox, Brian: GNU Bash Reference Manual, ISBN 0-9541617-7-7.
Ritchie1974	Ritchie, Dennis M and Thompson, Ken. *The UNIX Time-sharing System*. CACM 17 No. 7 (July, 1974). Reprinted with revisions in [V7Vol2A].
Ritchie1977	Ritchie, Dennis M. *The UNIX Time-sharing System-A Retrospective*. Tenth Annual Hawaii International Conference on the System Sciences, 1977; reprinted in the UNIX special issue of the *Bell System Technical Journal*, Vol 57, N 6, Part 2, July-August 1978.
Solex	The Solex web testing plug-in for Eclipse can be found at http://solex.sourceforge.net/

Salus1998	Salus, Peter. *Handbook of Programming Languages, Volume III: Little Languages and Tools*, Macmillan Technical Publishing, 1998. ISBN 1-578-70134-1.
Stevenson2006	Stevensen, Chris. Describes a commercial tool that uses Ruby to build a web testing framework that is otherwise similar to PageUnit. Presentation online at http://www.skizz.biz/wp-content/AcceptanceTesting.pdf
Sun1995	Sun Microsystems. The original Applet announcement is no longer available, but the Applet mechanism is summarized at http://java.sun.com/applets/.
Theodoru2006	Theodorou, Jochen. Posting to the *groovy-jsr* mailing list January 2, 2006., archived as http://article.gmane.org/gmane.comp.lang.groovy.jsr/1304.
V7Vol2A	Volume 2A of the *UNIX Seventh Edition Programmer's Manual* (1979) by AT&T Bell Laboratories. Was reprinted commercially (Fort Worth, TX, U.S.A. : Harcourt College Publishers, 1983) but has been out of print for years. Available on-line at http://netlib.bell-labs.com/7thEdMan/v7vol2a.pdf.
VanDeursen2000	van Duersen, Arie *et al*. Domain-Specific Languages: An Annotated Bibliography. 2000. Unpublished, but on-line at http://homepages.cwi.nl/~arie/papers/dslbib/.
Violet	Violet, Scott. *The Swing HTML Parser*. Undated article online at http://java.sun.com/products/jfc/tsc/articles/bookmarks/
Zalewski2004	Zalewski, Michael. *The HTML Manglizer* (sic), online at the author's rather informal web site http://lcamtuf.coredump.cx/, described at Security Focus: http://www.securityfocus.com/archive/1/378632.

Appendix A: PageUnit User Guide

Please see the following pages.