The Empire Infrastructure for Testing Experimentation

by
Brian Davia

a research paper
submitted to
Oregon State University

in partial fulfillment
of the requirements
for the degree of
Master of Science

April, 2001

Abstract

It is widely believed that there is not enough experimentation in the field of computer science. One area in particular in which additional experimentation is needed involves regression testing of large, real software. Regression testing is the expensive process of validating changes made to previously tested software. One reason why there is not enough experimentation in this area is the substantial infrastructure required for such experimentation. We have developed an infrastructure designed for performing various testing experiments on a large, real test subject. This paper describes the requirements, design, and implementation issues associated with the development of our infrastructure. We also performed an experiment to validate the infrastructure, in which we investigated the affects test granularity has on regression test selection techniques. The experiment shows the usefulness of our infrastructure. We believe that our infrastructure can serve as a prototype for future development of similar infrastructures involving other large test subjects.

Contents

1. Introduction

For most important discoveries in science, there have been experiments that have either validated or produced those discoveries. There have been simple experiments, such as Galileo dropping two cannon balls of different weight off of the Leaning Tower of Pisa to show that objects of different weight fall at the same rate. There have also been more complicated experiments, such as Robert Milikan's oil drop experiment, which led to the calculations of the mass of an electron in 1910. There are even accidental discoveries, such as that of penecillin by Alexander Fleming in 1928, that have been products of experimentation. Experimentation is important, because it leads to innovation and a better understanding of the world around us.

Experimentation is part of the foundation of science, and in this, computer science should not be an exception. In his paper *Should Computer Scientists Experiment More?*, Walter Tichy states, "To understand the nature of information processes, computer scientists must observe phenomena, formulate explanations and theories, and test them" [12]. Unfortunately, it seems as if this belief is not acted on widely enough. Tichy also cites findings that of a random sampling of papers published by ACM in 1993 that claim the need for empirical analysis, 40 percent of those papers provided none. It is clear that there is room for growth in experimentation in computer science.

One particular area of interest in experimentation within computer science is that of regression testing. Regression testing is an expensive testing process that attempts to validate changes made to previously tested software. One method of regression testing is to save the test suites used to validate the software, and rerun all of those tests after changes have been made. This method can prove to be expensive, particularly when the time needed to execute these tests can be on the order of days. One way to reduce this cost is to select and run only the tests that validate the modified or new portions of code.

Since effective regression testing techniques could save companies a great deal of time and money, regression testing is a popular area of research. However, as popular as it is in the field of research, it is widely acknowledged that more experimentation in regression testing is needed [2, 3, 4, 5, 6, 9, 13].

One reason why there hasn't been enough experimentation in regression testing is that such experimentation requires substantial infrastructure. First, it requires subject programs which have multiple versions. Second, it requires tests that can be executed on those programs and their versions. Lastly, it requires automation. As the number of tests needed to validate a piece of software increases, so does the need for automation. By automating tests, we can accomplish in a few hours what may otherwise take weeks by hand.

There have been previous efforts to produce infrastructure for regression testing that met the requirements outlined above [11]. However, this infrastructure was developed for relatively small programs (100 -1000 lines of code). To make continued progress in experimentation using regression testing techniques, there is a need for an infrastructure that can handle larger, real-world programs.

For this project, we have produced such an infrastructure. Our infrastructure is designed to be flexible, so that it can support many different types of testing experiments. The infrastructure focuses on a large (about 80,000 lines of code), real-world program, and several of its versions. It also consists of nearly 2,000 tests aimed at testing the functionality of that program. The infrastructure has automation capabilities that will execute a test suite on multiple versions of the program, save the results, and analyze them. Finally, although the infrastructure focuses on a single, specific program, we believe that it serves as a prototype for such infrastructures, and thus, can be used to support additional large experimental subjects in the future.

To validate the usability of our infrastructure, we conducted a case study investigating how test granularity affects the effectiveness and efficiency of regression test selection. As a result of this study, we report that there may be little

benefit to using regression test selection techniques with test sets of coarse granularity. Further, we find that regression test selection techniques applied to fine-grained test sets can be competitive with the retest-all method applied to coarser-grained test sets.

The remainder of this paper is organized as follows. The second section of this paper describes the requirements that the infrastructure must meet. The third section briefly describes the test subject that is used in our infrastructure. The fourth section describes the details involved in designing and implementing the infrastructure, as well as the tests for the subject. The fifth section summarizes a case study that was completed using the infrastructure, and the results of that study. Finally, the sixth section presents our conclusions and discusses further work.

2. Requirements

In this section we discuss the requirements that our infrastructure should satisfy. When specifying these requirements, we must be sensitive to how the infrastructure will be used. Our overall goal is to create an infrastructure that supports experimentation with testing and regression testing techniques on a large, multi-version software system.

There are several items that comprise such an infrastructure. First of all, there is the program that is to be tested and all of its versions. Second, there is a set of realistic tests and test suites that are used to test the program. Third, there should be a set of tools that can be used to accomplish various tasks involved in testing, such as installing the program and executing tests.

It should be made clear that we are not producing this infrastructure for the sole purpose of supporting a single experiment. Instead, we would like to produce a

system that is extensible and flexible enough so that it can be used in various experiments pertaining to testing the specified subject.

Initially, the system must be able to support the following experiments:

1. Evaluation and comparison of regression test selection techniques. Given a program, its test suites, and its version history, which tests should be selected to guarantee coverage of modified code between versions?
2. Test case prioritization. How should tests be prioritized in order to improve various test measures, such as rate of fault detection?
3. Test suite minimization. Given a program, and a test suite, which tests should be selected to guarantee coverage of desired code in that program while minimizing the number of tests selected?
4. Regression testability issues. How do factors in test process or test suite design affect the efficiency and/or effectiveness of regression testing?

These overall requirements imply a number of additional requirements for the infrastructure. These requirements will be explored in the following sections.

2.1 Test Repeatability

Since we are creating a system that assists in executing testing experiments, we also need to address issues that arise when executing any scientific experiment. We will be most concerned with controlling the environment in which we are testing. We want all tests to be repeatable. That is, suppose $f(x,v)$ represents the result (outputs) of some test x executed on some version v. Let $f'(x,v)$ be the results of executing the same test x on the same version v a second time. Then $f(x,v)$ must be equal to $f'(x,v)$.[1] Repeatability is an issue that we will encounter further while addressing the requirements of the infrastructure, as well as the design and implementation of the infrastructure.

[1] At least equal barring any extraordinary circumstances such as disk crashes or power failures during test execution.

2.2 Subject Requirements

There are certain qualities that we would like our test subject to possess in order to support experimentation such as that outlined previously.

We would like the experimentation subject to be relatively large (at least 10,000 lines of code), and available in multiple versions. The subject should also be "real". That is, it should be software that is used in the real world, rather than software that was developed specifically for experimentation. The subject should be written in C for reasons that will be discussed later in this section.

We would like our subject to be maintained in a version control system. By maintaining the code of the subject in such a manner, we can efficiently store the many versions of the subject, as well as ensure that we can retrieve any desired version.

Another quality that we require of our subject in order to make our experiments repeatable is determinism. As discussed in Section 2.1, we must attempt to ensure that, regardless of when or how many times we run a test on a single version, we receive the same results from that test every time. If the subject is not deterministic, our infrastructure may attempt to point out false bugs. For example, suppose the subject prints the date at some point during execution. If we run tests that extract this behavior on two separate days, and if our system works as it should, a bug will be reported because there were two different outputs for the same test. This information is useless to those who are running the experiment, since the program is simply behaving as it should. Another source of non-determinism is the use of random number generators. In order to remove non-determinism, we may need to modify the original code of the subject so that it behaves in a predictable manner.

When executing a test, it is often desirable to measure the effectiveness of that test. One possible measure of effectiveness is test coverage, in which we record the portions of code that were executed while running a test (or suite). Such coverage information includes the percentage of the code (functions, statements, or branches) that was covered, and exactly what code was executed during the tests. For the user of the system to be able to extract meaningful information about the coverage of the tests executed on the subject, the subject should be instrumented in some manner in order to calculate the test coverage. The instrumentation software that should be used by the infrastructure is Aristotle [7]. Since this software works only on C-based programs, the subject must be written in C.

2.3 Requirements of the Automated Testing Procedure

One general requirement for our infrastructure is that it provides a system that automatically runs selected tests on the subject. In this section, we describe a general outline of what we would like the system to do while running tests. This entails an explanation of the inputs and outputs for the system, as well as a description of the testing process.

Let's begin by addressing what the system needs to do in order to execute a test on the subject. Since the subject will be under version control, the system must first extract the subject from the version control system repository. Next, since only the source code of the subject will be stored in the repository, the system must take the steps necessary to build and install the subject.

We would like to import any data files that the subject may rely on. Depending on the test subject, there may be many areas of functionality of the program that we would like to test that are not accessible with a newly built instance of the subject. By importing pre-configured data files, we can effectively force the subject into a state in which we are able to test all desired functionality. Note that this should be done before each test, and not just after the subject has been installed. The reason is

that we would like a controlled environment when executing each test. Each test should be executed with the same start state; this is necessary for the testing process to be repeatable.

Once the subject is built and installed, and the data files have been imported, the system is ready to start executing tests. The structure of the tests is really a design issue and depends on the subject. However, we will require that the structure of the tests allow for flexibility in execution. It is very important that the system be flexible enough to handle tests of different size and test suites of different size, and be able to test any number of versions of the subject.

After each test is executed, the system should save the results of that test. Specifically, we can identify three items that we would like to save. The first item is the output of the subject; this is the primary output produced by the program, and is usually output sent to standard-out or standard-error (the computer screen). The second item concerns the state of the system after the test was executed. This involves data files that are maintained by the test subject. Last, we instrumented the subject in order to collect data on the effectiveness of the test executed (and the effectiveness of a suite). Thus, the third item we would like to save after each test is the trace file produced by the instrumentation.

We have now outlined the requirements that the system must meet when executing a test suite. These requirements allow us to define a process that our system must follow. The system must first build and install the subject. Then, for each test in the test suite for the subject, the system must import data files, execute the test, and save the data for that test. The result of executing the test suite will be a directory populated by test output, data directories, and trace files. All of these should be stored (named) in a manner in which it can be easily determined which test produced which results.

2.4 Requirements for Regression Testing

We have discussed the requirements of the system that must be met to support execution of a test suite. We now discuss the capabilities the system must have in order to execute regression tests in an automated fashion.

We would like the system to be able to run a test suite on one or more versions of the subject. There are several different types of tests that we would like the system to be able to run, so we need a flexible system that would be able to handle any of these possibilities. Specifically, we would like the system to be able to do the following:

1. run a suite of tests on a single version;
2. run a suite of tests on a pair of versions, and compare the results;
3. run a suite of tests on a sequence of pairs of versions, and compare the results.

In order for the system to do this, on any execution of the system, we must supply the system with (1) a test suite and (2) a list of versions to be tested.

Item 1 was addressed in the previous section. For items 2 and 3, we assume that anytime there is more than one version specified to be tested, the system is performing regression tests. In the case that only a pair of versions is specified, once the suite is executed on each version, the system should compare the results. The system should report any differences in either the outputs of each test or the contents of the resulting data directories. If there are more than two versions specified to be tested, then the results of testing each pair of versions in sequence should be compared. For example, if we specify versions v_1, v_2, v_3 and to be tested, then the results of pairs (v_1, v_2) and (v_2, v_3) will be compared, but not (v_1, v_3).

3. Case Study Subject

The infrastructure whose requirements were outlined in Section 2 will be used in a number of experiments focused on testing methodologies and issues. Although the requirements that have been outlined could be applied to any test subject, there are a number of issues in the design and implementation of the system infrastructure that are dependent on the test subject chosen for experimentation. Thus, before discussing the design and implementation, we first present the test subject that was chosen for this project.

The software for which we have built our infrastructure is a client/server game called Empire. Empire is a game of world domination where the setting of the game is a world in which each player has their own country. The goal of the game is for a player to develop his or her country's economy and military to the point of overpowering all other countries in the game. A game of Empire is setup, started, and maintained by the *deity* of the game who installs and activates a server. The players of the game dial up using a client and play from their own terminal, issuing commands via the command line. The functionality available to the player consists of 182 commands (most with multiple parameters and options). A sampling of these commands is provided in Figure 3.1. We have chosen the *server* of this system to be our testing subject.

Figure 3.1: Some Sample Commands

Command	*Description of functionality*
move	Moves people or commodities between sectors.
commodity	Displays information on the commodities in each sector.
map	Displays a map of the land visible to the user.
telegram	Write a telegram to another player in the game.
build	Builds a plane, ship, land unit, bridge, or nuclear warhead.
load	Load a plane, a land unit, or commodities onto a ship.
offer	Allows user to offer a loan or treaty to another country.
bye	Logs the user out of the game.

There are several reasons why we chose Empire as a test subject. First, it met all of the requirements addressed in the previous section: it is a large piece of software (around 80,000 lines of code) written in C, it is used in the real world, and it has multiple versions (29 versions to date) available. The second reason why Empire was chosen was the availability of source code and documentation for the system. Empire is distributed under the GNU General Public License Agreement, and can be downloaded at no cost. Along with the software comes a large amount of documentation on how to play the game. This documentation is crucial in designing tests for use in experiments.

4. Infrastructure Design and Implementation

We now discuss the design and implementation of our infrastructure. The section is split into four parts. The first part addresses the steps that were taken to prepare the source code of the subject for the infrastructure. The second part discusses the format of a test, and why we chose this format. The third part of this section describes the

different components that make up the automated regression test system. The last section describes the contents and purposes of the different directories within the infrastructure that supports the system.

4.1 Source Code and Version Preparation

Section 2.2 outlined specific requirements that must be met by the test subject. Before a version of the subject could be included in our infrastructure for testing, there were three tasks that needed to be completed in order to meet these requirements. First, all nondeterministic portions of code were modified in order to produce a deterministic version of the subject, including instances where date and time were considered, as well as instances where random numbers were used. Second, the Makefiles used for building and installing the subject were modified to allow for building both regular and instrumented instances of the subject. The last task was to insert the version into the CVS repository. Once these steps were completed for each version, the subject was ready for test development.

4.2 The Composition of a Test

The major issue that needs to be addressed when determining the format of a test is flexibility. As stated in the requirements, the system must be flexible enough to handle test suites of different sizes, and allow different granularities of tests to be constructed. Let's start by looking at how we created tests and then look at how we implemented those tests.

4.2.1 Creating Tests with the Category Partition Method

For this project, we chose to create tests adequate for functional testing. The goal of functional testing is to locate faults within the software; that is, to find discrepancies between the specifications of the software and the actual behavior of the software.

We used a semi-formal method for creating functional tests for our test subject called The Category Partition Method [1]. This method required us to use a Test Specification Language (TSL) to represent the specification for each command. This allowed us to partially automate the process of creating tests. Figure 4.1 shows a sample command specification provided with our subject and Figure 4.2 shows a test specification created for that command by applying the Category Partition Method and using TSL.

Figure 4.1: Functional Specification for *toggle* Command

Command : TOGGLE

NAME
toggle - Toggle client flags

Level: Expert

SYNTAX
[##:##]% toggle [inform|flash|beep|coastwatch|sonar|techlists] [on|off]

You use the toggle command to set various user-interface flags for your country. The server will remember which flags you have set between sessions. If you do not specify the second argument ("on" or "off"), then the server will just flip the switch from on to off or vice versa.

[##:##] Command: toggle inform
Will toggle the "inform" flag (default off) for your country. When inform is on, the server will inform you immediately whenever you receive a telegram.

[##:##] Command: toggle flash
Will toggle the "flash" flag (default on) for your country. When flash is on, you can receive instantaneous messages from your allies when they use the "flash" and "wall" commands.

[##:##] Command: toggle beep
Will toggle the "beep" flag (default on) for your country. When beep is on, your terminal will beep whenever there's a big explosion in the game (shells, bombs, torpedoes...). You might want to turn beep off if you don't want to disturb others.

[##:##] Command: toggle coastwatch
Will toggle the "coastwatch" flag (default on) for your country. When coastwatch is on, then you will receive telegrams informing you of any new ship sightings.

[##:##] Command: toggle sonar
Will toggle the "sonar" flag (default on) for your country. When sonar is on, then you will receive telegrams whenever any of your sonar-equipped ships detect a sonar ping.

[##:##] Command: toggle techlists
Will toggle the "techlists" flag (default off) for your country. When techlists is on, when you do a "show" command, the lists shows will be in order of tech instead of grouping order.

[##:##] Command: toggle
Will list all your flags and show whether they are on or off.

SEE ALSO
telegram, flash, wall, Communication

Figure 4.2: Test Specification for *toggle* Command

```
# Test specification for toggle
# [##:##]% toggle [inform|flash|beep|coastwatch|sonar|techlists] [on|off]

Parameters:
        what to toggle:
                'inform'.                       [single]
                'flash'.                        [single]
                'beep'.                         [single]
                'coastwatch'.                   [single]
                'sonar'.                        [single]
                'techlists'.
                blank.                          [property blank]
                something else.                 [error]

        on or off:
                'on'.                           [if !blank]
                'off'.                          [if !blank]
                blank.                          [if blank]
                something else.                 [error] [if !blank]

Environment:
        property was on before call:
                yes.                            [if !blank]
                no.                             [if !blank]
```

To apply the Category Partition Method, we start by analyzing the specifications of the program, function, or in our case, the command that we wish to test. We identify input parameters as well as environmental influences (factors other than parameters that affect the function of the command); these are our *categories*. In our example, the categories are 'what to toggle', 'on or off', and 'property was on before call'. Once we have specified these, we identify specific *choices* for each category. For example, in category 'on or off', we identified four choices: 'on', 'off', blank, or something else. Tests are generated by combining choices for each category. Finally, since the full set of combinations may yield too large a set of tests, we can associate constraints with individual choices to limit the number of combinations. For example, consider the 'what to toggle' parameter in Figure 4.2. Since we associate [single] with it, the 'inform' choice will be tested only once. We also added [property blank] to the 'blank' choice for 'what to toggle'. Once this property is declared, we can use it to apply constraints to the remaining parameters. We have effectively cut our number of tests down from 64 (8*4*2) to 12 by using these constraints.

After running a TSL tool on our specification, we are left with the test cases specified in Figure 4.3. It is then left to the tester to implement the specified tests.

Figure 4.3: Test Cases for *toggle* Command

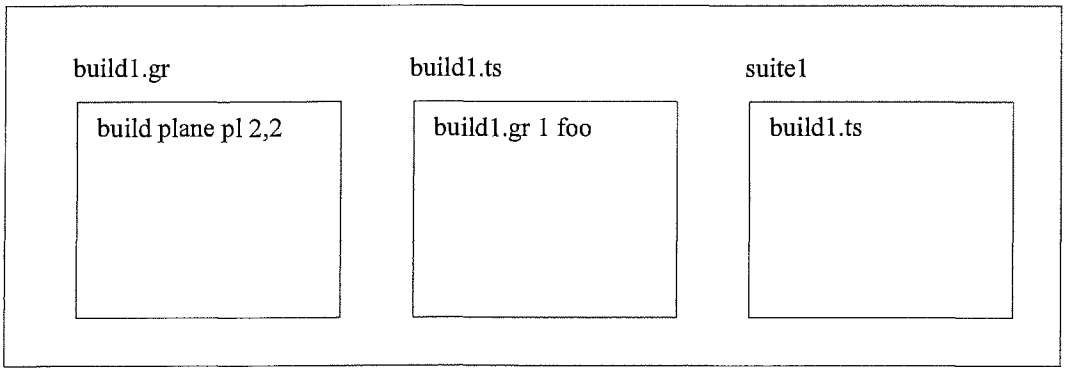
```
Test Case 1                 <single>
  what to toggle :  'inform'

Test Case 2                 <single>
  what to toggle :  'flash'

Test Case 3                 <single>
  what to toggle :  'beep'

Test Case 4                 <single>
  what to toggle :  'coastwatch'

Test Case 5                 <single>
  what to toggle :  'sonar'

Test Case 6                 <error>
  what to toggle :  other fluff

Test Case 7                 <error>
  on or off :  other fluff

Test Case 8                 (Key = 6.1.1.)
  what to toggle                  : 'techlists'
  on or off                       : 'on'
  property was on before call     : yes

Test Case 9                 (Key = 6.1.2.)
  what to toggle                  : 'techlists'
  on or off                       : 'on'
  property was on before call :  no

Test Case 10                (Key = 6.2.1.)
  what to toggle                  : 'techlists'
  on or off                       : 'off'
  property was on before call     : yes

Test Case 11                (Key = 6.2.2.)
  what to toggle                  : 'techlists'
  on or off                       : 'off'
  property was on before call :  no

Test Case 12                (Key = 7.3.0.)
  what to toggle                  : blank
  on or off                       : blank
  property was on before call     : <n/a>
```

4.2.2 Handling Different Test Granularities

As stated in the requirements, the only real requirement for how a test is structured is that it be flexible. We must design the infrastructure so that it can handle tests of different granularities, as well as test suites of different sizes. It may be beneficial to define some terminology at this time. A *test* is a set of inputs that can be applied to a program, which yield some outputs. A *test suite* is a collection of one or more tests. The term *granularity* refers to the size of a test. A test of the finest granularity is a test of the smallest possible size, where size is some measure appropriate to the system being tested. To look at these issues in our context, consider the test subject for which we are designing the system; Empire. Empire is a game in which the user issues commands via a command line when prompted. Therefore, for Empire, a test of the finest granularity (granularity-1) is a test that executes only one command. A test of granularity-4 involves four commands.

We decided that the best means of storage for a test would be a file. There are a few levels of abstraction to discuss in this design, so we will start at the bottom and work our way up.

Figure 4.4



The foundation of our test structure, the building block with which we build all of our tests, is a *test grain.* Each test grain is composed of one command that can be issued to the test subject. For example, we could have a file (a test grain) build1.gr that contains a command to build a plane (Figure 4.4, left). With this test grain, we can create a test. Empire is a game with a number of players. So, when creating a test,

we must specify the player (and password) for which the grain should be executed, as well as the test grain itself. So, to complete our example, we can represent a test that executes build1.gr for player 1 with password 'foo' as 'build1.gr 1 foo' (Figure 4.4, center).

Each test is represented in a file. The test that we created above, for example, is contained in the file build1.ts. Continuing to build out, we can create a test suite by listing a number of tests. Figure 4.4 (right) summarizes the contents of an example suite.

By structuring our tests into grains, tests, and suites, we have effectively modularized our test structure. A test suite is built using tests. A test is built using test grains. This modularization gives us a great deal of flexibility. Of course, there can be any number of tests in a test suite. Also, we can use any number of test grains within each test. This allows us to meet the requirement that the system handle tests of any size. Figure 4.5 shows what a test of granularity-4 may look like.

Figure 4.5: build2.ts

```
build1.gr 1 foo
build2.gr 1 foo
build2.gr 3 bar
build5.gr 3 bar
```

4.3 Components of the Infrastructure

There are many repetitive steps involved in testing a piece of software. One purpose of our infrastructure is to identify those repetitive steps, and automate them. By automating the test process, we can decrease the time it takes to execute tests, as well as lessen the chance of human error while executing tests. In the following paragraphs, while we discuss the key steps in the process of testing our subject, we identify the major components of our automated system.

Let's start by taking a look at the big picture. We would like to run regression tests on the subject. The requirements state that the system should be able to run a suite of tests on any number of specified versions of the subject. This suggests an outermost loop of executing a suite of tests for each version listed.

So what needs to be done in order to run a suite of tests on a single version of the subject? A complete summary of the algorithm that we used can be found in Figure 4.6.

Figure 4.6: Algorithm for run_test.pl

```
For each version specified
   1. build & install subject
   2. for each test in suite
   3.       import data files
   4.       start server
   5.       for each grain in test
   6.            execute test grain
   7.       stop server
   8.       store, tar, and zip output and data files
   9. remove subject
```

The first component of our infrastructure is a Perl script called run_test.pl, following the algorithm outlined in Figure 4.6. The first step in the algorithm is to extract the specified version of the subject from the CVS repository, build it, and install it (line 1 of Figure 4.6). To automate this, we created make_install.sh, a simple shell script that extracts the specified version of the subject from CVS, builds the subject, and takes any other steps necessary to enable running of the subject. (This is the second component of our infrastructure.)

Next, for each test in the suite, we must initialize the state of the subject so that the test will execute as designed. Once the server has been started, we then execute the test and save the appropriate output and data files. We group all of these steps into a second loop (lines 3-8 of Figure 4.6).

Finally, each grain within the test needs to be executed (lines 5-6 of Figure 4.6). Up to this point, every step can be accomplished by using simple shell and Perl scripts. However, the task of executing a test grain is not so simple. Remember that we are dealing with an interactive program, and simply executing the program with some command line parameters will not suffice. This problem is handled in our third component, called test_emp.ex. test_emp.ex is a script written in Expect[10], which is an extension of Tcl. Expect gives us the capability of running a program (like the Empire client) and interacting with that program. Using Expect, we are able to specify prompts to look for from the program, and input to feed to the program when we see those prompts. For example, consider the steps that are taken to log a user in to the server using the client. Figure 4.7 shows the steps that a user would take to login by hand (italics distinguish user inputs), while Figure 4.8 shows the section of test_emp.ex that allows the infrastructure to login to the server.

Figure 4.7

```
1% emp_client
Country name? POGO
Your name?  *****

[0:640] Command :
```

Figure 4.8

```
1.      spawn csh
          . . .

2.      expect "%"
3.      send "emp_client\r"

4.      expect {
5.        "% $"                 { exit }
6.        "Country name?"             { }
7.      }
8.      send "$name\r"

9.      expect {
10.       "% $"                 { exit }
11.       "Your name?"          { }
12.     }
13.     send "$pswd\r"

14.     expect {
15.       "% $"                 { exit }
16.       "Command : "          { }
17.     }

          . . .
```

The example in Figure 4.8 starts out by spawning a process in line 1, using the *spawn* command. In this case, the process is a shell. The *expect* command waits for a string from that process: in line 2, Expect waits for the shell prompt. Upon receiving that prompt, Expect uses the *send* command to send a string to the spawned process: in line 3, Expect starts the client of empire by sending "emp_client\r" to the spawned process. Lines 4-7 of Figure 4.8 illustrate Expect's ability to handle multiple possibilities within the spawned process. In line 5, if Expect sees the shell prompt ('$' anchors the end of a string), then the client was unable to connect to the server, and thus the test should exit. However, in line 6, if the spawned process prompts for

"Country Name:", Expect continues to the next line where it sends the country name to the spawned process. Similarly, in lines 9-13, Expect waits for the prompt for a password, and then sends the password to the spawned process. If the login failed, Expect will receive the shell prompt from the spawned process and will consequently exit, as specified in line 15. However, if the spawned process produces the command prompt, then the Expect script has successfully logged-in to the server, and continues on to the next statement.

Empire uses 22 different prompts in total. Since Expect allows us to handle multiple possible prompts, test_emp.ex handles all 22 prompts, and is used to execute all test grains.

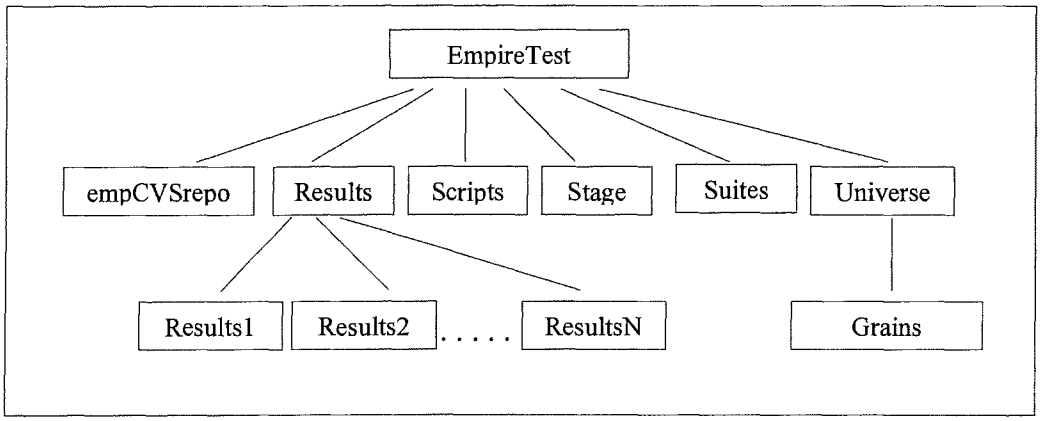
Once each test in the suite has been executed, we remove the subject that we have been testing (line 9 of Figure 4.6), and prepare to run tests on the next specified version of the subject.

In summary, the three major components of the automated system are run_test.pl (the driver of the system), make_install.sh, and test_emp.ex. There are a few smaller components used to assist with minor details of the system operation, such as a script that compares the results. However, the components described make up the framework of the system.

4.4 The Directory Tree

The infrastructure that we have designed to this point requires several script files. The infrastructure could also provide large numbers of test suites and thousands of tests, and produce multiple instances of test results. To maintain this large collection of files in an orderly manner, we have designed a directory tree structure. The structure of this tree is shown in Figure 4.9. A description of the contents of each directory in this tree follows.

Figure 4.9: Tree Structure



empCVSrepo: This directory is maintained by CVS and contains the source code of all versions of the test subject.

Results[1,2,3,...]: The directories Results1, Results2, ... ResultsN contain tarred and zipped test results for each of the N versions that are tested during a test run.

Results: This is the parent directory for Results1, Results2, ... ResultsN.

Scripts: This directory contains all the scripts used by the infrastructure. Such scripts include test_emp.ex and make_install.sh, as well as other minor utility scripts used in running tests and analyzing results.

Stage: This is the location where all tests are run. A version of the subject is extracted to this directory to be tested.

Suites: This directory holds all suites that can be used in running tests.

Universe: The Universe directory holds all tests. Included in the Universe directory are all .ts files, as well as the Grains directory.

Grains: This directory contains all test grains (.gr files) used for testing.

5. A Case Study in Regression Test Selection Techniques

The primary goal of our infrastructure is to support experiments on regression testing. To assess whether we have accomplished this goal, we used our infrastructure to perform a prototype case study. The objective of the case study is to investigate the effects that different test case granularities have on regression test selection (RTS) techniques.

5.1 Subject

The subject that we will be testing is the subject which we developed the infrastructure for; Empire. We chose five versions of Empire to work with. These versions are summarized in Table 5.1.

Table 5.1

Version	**Number of Functions**	*Number of Functions Changed from Previous Version*
4.2.0	1159	-
4.2.1	1159	51
4.2.3	1171	300
4.2.4	1172	9
4.2.5	1173	100

We created several sets of tests aimed at testing the functionality of the server of Empire. The first set of tests is the granularity-1 tests created by using the techniques described in section 4.2.1. There are 1,985 tests in the granularity-1 set. Using this set of tests, we then created sets of granularity-4, granularity-16, and granularity-64 tests. As described in section 4.2.2, a test of granularity-n is just n test grains selected to make a single coarser-grained test.

In the sets of granularity-n tests, n test grains were each used exactly once. In creating a set of granularity-n tests, the first step is to create a list of granularity-1

tests. The next step is to randomly remove n granularity-1 tests from this list and use them to create a single test of granularity-n. This step is repeated until there are at most n granularity-1 tests remaining in the list. The final granularity-n test is composed of the tests remaining in the list of granularity-1 tests. The test sets we created for Empire by this process are summarized in Table 5.2.

Table 5.2

Test Set	Number of Tests
granularity-1	1985
granularity-4	497
granularity-16	125
granularity-64	32

5.2 Techniques

There are two regression testing techniques that are used in this study. The first technique, called *retest-all*, takes all tests that were created and executed on a version, v_k, and executes them all on the succeeding version, v_{k+1}.

The second technique uses a selection technique to select a test set that is *safe* [8], and runs only those tests selected. In this study, we use a selection technique that is safe at the function level; that is, we select tests that execute either changed or missing functions between versions v_k and v_{k+1}. The first step in this process is to execute all tests on v_k and collect coverage information for each test. The second step is to find all functions that either differ in v_{k+1} from v_k, or exist in v_k, but not in v_{k+1}. The third step is to cross-reference the list of changed and deleted functions with the coverage information collected in the first step. Every test that executes, on v_k, a changed function, or used to execute, on v_k, a function that has been deleted, is selected. The result is a list of tests that, when run on v_k, execute functions that have changed in, or

deleted from v_{k+1}. The fourth, and final step is to run the tests that resulted from the third step on v_{k+1}. A summary of these steps is listed in Figure 5.1.

Figure 5.1 Regression test selection technique.

1. Run all tests on v_k and collect coverage information.
2. Find all differences between v_k and v_{k+1} at the functional level.
3. Cross-reference functions found in step 2 with coverage information found in step 1.
4. Run the tests resulting from step 3 on v_{k+1}.

5.3 Procedure

The main objective of this study is to apply the two different regression testing techniques just described using the four different granularity-n test sets, and see how the two techniques compare under different test granularities. Figure 5.2 summarizes the algorithm for our procedure.

Figure 5.2

```
1.    For each test granularity level G
2.       For each pair of sequential versions vk, vk+1
3.          Run TG on vk and collect test results
4.          Measure the time to run TG on vk+1 (retest all)
5.          Gather the coverage results from running TG on vk
6.          Cross-reference coverage results with the changed
            functions between vk, vk+1 to determine set of
            selected tests
7.           Measure time to run selected tests from step 7 on
            vk+1
```

There are two metrics that are recorded during this procedure. The first is how long it takes to run a suite of tests on a version (step 4), and also the time it takes to run a selected subset of tests on that version (step 7). The second metric is the number of tests that are selected to be safe at the function level (step 6).

5.4 Results

Table 5.3 summarizes the number of tests selected by our test selection technique (step 3 in Figure 5.1) by listing the percentage of tests selected from the complete granularity-n test set, for each granularity level.

Table 5.3 Percentage of tests selected to be functionally safe.

Version	Granularity-1	Granularity-4	Granularity-16	Granularity-64
4.2.0	29.9%	74.4%	97.6%	100%
4.2.1	85.1%	100%	100%	100%
4.2.3	6.7%	23.6%	60.8%	93.8%
4.2.4	59.9%	98.0%	100%	100%

In general, as the granularity of the test set increased, so did the percentage of tests selected. Therefore, we can say that as the granularity of test sets increased, the effectiveness of the selection technique used decreased. This result was consistent across all versions.

The other metric that we want to look at is the time it takes to re-test all and run selected tests for each version. These results are displayed in Figures 5.3 – 5.6.

Figure 5.3 Suite execution time for version 4.2.1

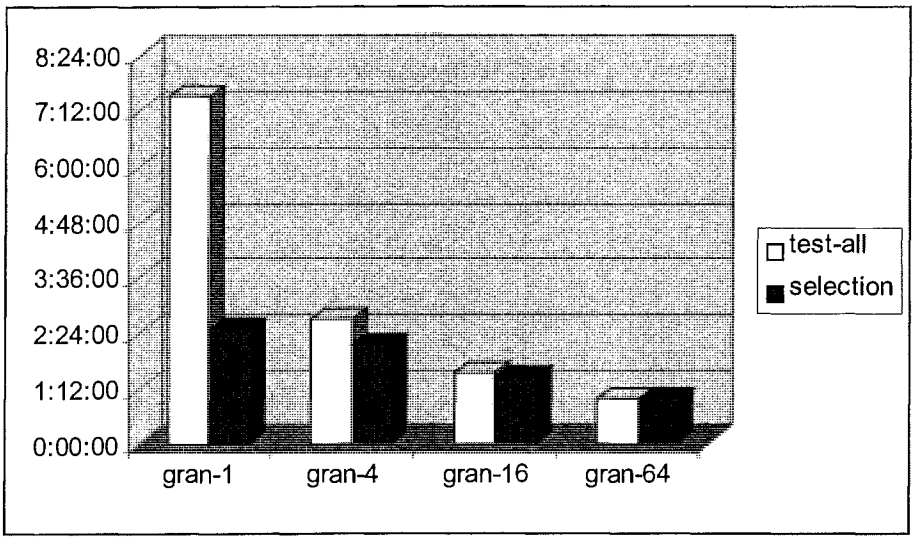


Figure 5.4 Suite execution time for version 4.2.3

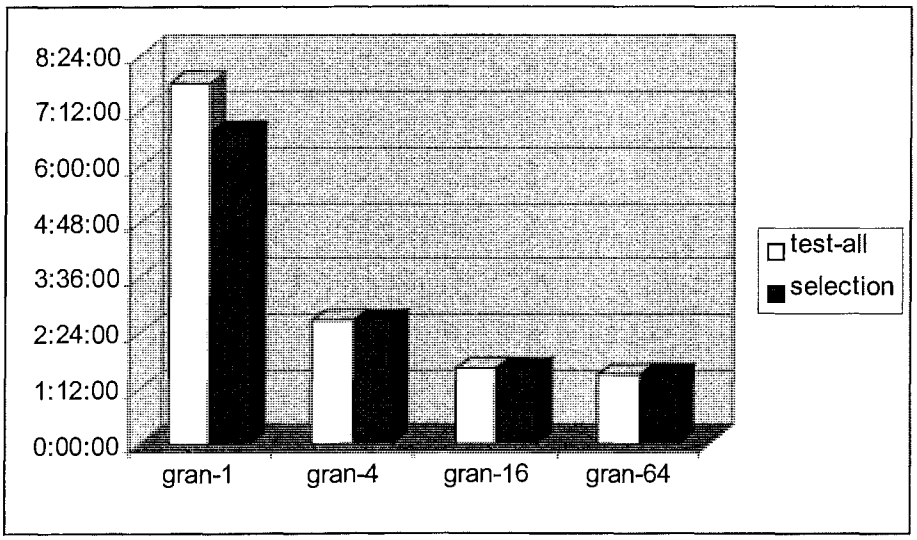


Figure 5.5 Suite execution time for version 4.2.4

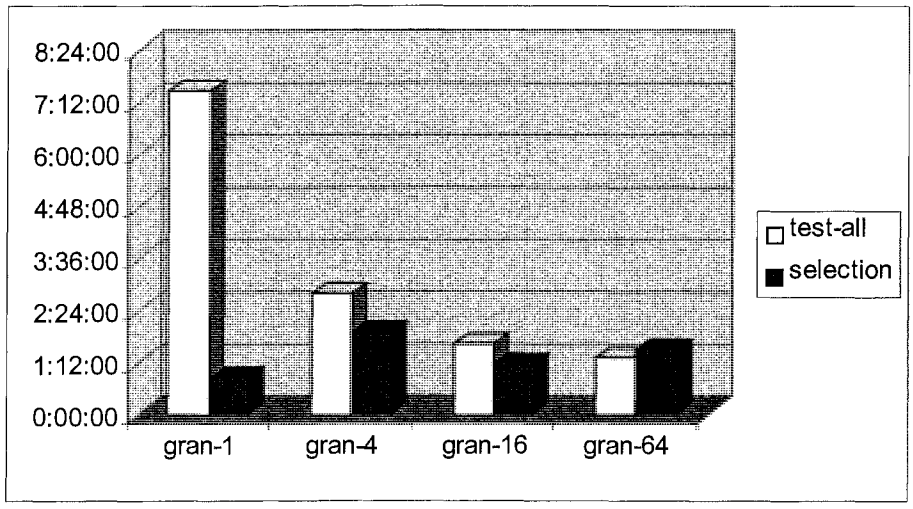
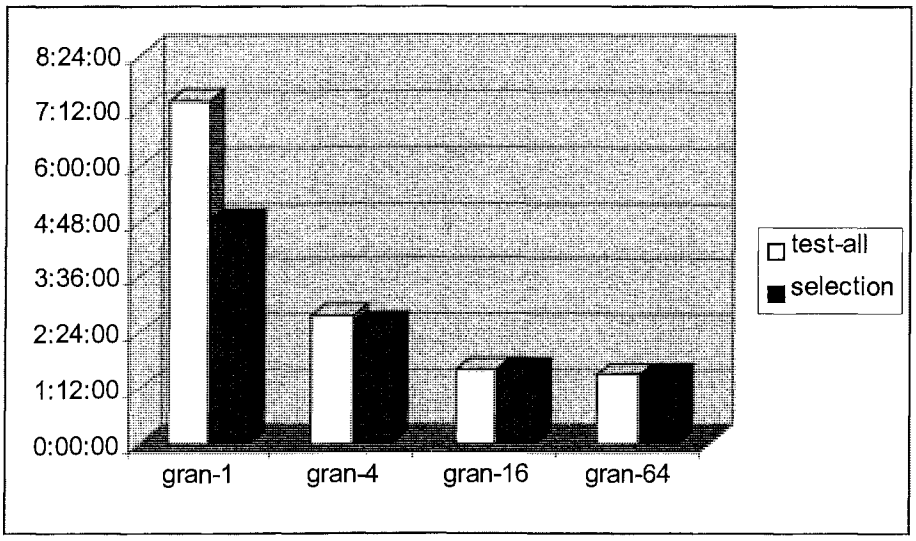


Figure 5.6 Suite execution time for version 4.2.5



The first thing that we notice in the results depicted above is that as the granularity increased, the time to re-test all decreased. Recall that there is overhead incurred each time a test is executed; mainly the time it takes to reset the server, and to save the results. Since there are actually fewer tests executed in the larger granularity sets, this overhead does not occur as much in those sets as it does in smaller granularity sets.

We also notice that there is a much greater benefit for using the regression test selection technique for the granularity-1 set of tests than there is for the others. This is in direct correlation with the number of tests selected shown in Table 5.3.

The results suggest that with very coarse-grained test sets, there may be little to gain by using a selection technique. With granularity-16 and granularity-64 test sets, there was little or no time saved by using the test selection technique.

Figures 5.3 and 5.5 also show, however, that using the regression test selection technique with the granularity-1 set of tests can be competitive with the retest-all technique applied to the other sets of tests. This is important, because there is at least one other reason to use tests of fine granularity, and that is that fine granularity tests can be more accurate in pinpointing faults. For example, consider a granularity-64 test that consists of 64 commands. If the test fails catastrophically (the server shuts down) on the fourth command, the remaining 60 commands will not be executed. This leads to inefficiency since it is possible that somewhere in the remaining 60 commands, additional faults are uncovered. Yet, to actually uncover these faults, the problem uncovered by the fourth test must first be handled, and then the test can be rerun. Only at this time will the next fault appear. This can all be avoided when using fine granularity tests, where at most one fault can be uncovered by each test.

In summary, we see that in the case studied, the benefits of using the regression test selection technique were more prevalent when applied to test sets of fine granularity than when applied to tests sets of coarser granularity.

6. Conclusion and Future Work

In this project, we have identified the requirements for, designed, and implemented an infrastructure to be used for regression testing experiments, where the test subject for these experiments is a large, real program. Though the design and implementation of our infrastructure are in some ways dependent on the test subject, we believe our infrastructure can serve as a prototype for additional infrastructures built for other large, real test subjects.

Our infrastructure was designed to support multiple types of regression testing experiments. To validate our infrastructure, we performed a case study that utilized many of the features of that infrastructure. This case study explored the effects that test granularity has on the effectiveness of regression test selection (RTS) techniques. We found that RTS techniques applied to fine-grained test sets can be competitive with the retest-all method applied to coarser-grained test sets. We also found that there may be little benefit to using RTS techniques on coarser-grained test sets.

For future work, there are many more experiments that can be implemented by using our infrastructure. One experiment could take our case study a step further by including investigation into the effectiveness of finding faults. Other experiments, for example, could focus on test suite minimization, and test case prioritization.

In addition to further experimentation, there is further work to be done on the infrastructure itself. First, faults within the various versions of the subject need to be identified and categorized. Second, by running a real game, we can collect operational profiles on user usage, which could be used to investigate issues in reliability. Finally, the test subject chosen for our infrastructure, Empire, is a continuous work in progress. As new versions of Empire are released, they should be included within the infrastructure.

Appendix A: run_test.pl

```
#!/usr/bin/perl

# Title: run_test.pl

# Author: Brian Davia
# Project: Empire Regression Testing

# This script is designed to perform regression tests on Empire.
# As inputs, it takes a list of versions, and a list of tests to
# be run on each of those versions.  The script builds and
# installs the specified version, runs the specified tests on the
# version, and saves the results of each test.  When more than one
# version is specified, the script analyzes the results between
# each sequential pair, looking for descepancies in their output files.

#inputs:
#     1 - file containing a list of all versions to run tests on
#           (one version per line)
#     2 - file containing a list of tests to be run on each version
#           (one test per line)

($versList, $testList, $flag) = @ARGV;
$scriptsDir = '/nfs/phantom/u8/davia2/EmpireTest/scripts';
$stageDir = '/nfs/phantom/u8/davia2/EmpireTest/stage';
#$testDir = './';
$testDir = '/nfs/phantom/u8/davia2/EmpireTest/test_suites';
$resultsDir = '/nfs/phantom/u8/davia2/EmpireTest/results';
$univDir = '/nfs/phantom/u8/davia2/EmpireTest/universe';
$dataDir = '/nfs/phantom/u8/davia2/EmpireTest/stage/data';
$begin = `date`;
$AristotDir = '/nfs/phantom/u8/davia2/AristotleDB';


print "versList = $versList\n";
print "testList = $testList\n";

# open the file containing the list of versions
open(VFILES,"$versList") || die "cannot open $versList: $!";
$vers_num = 1;

while (<VFILES>) {
  chomp;
  $version = $_;

  # create directories for results
  system("mkdir $resultsDir/results$vers_num");
  system("mkdir $resultsDir/traces$vers_num");

  $vers_begin = `date`;

  #kill existing empire server....
  system("$scriptsDir/kill_emp.sh");
  system("sleep 3");
  print("removing old emp\n");
```

```
system("rm -fr emp4");
print("finished removing emp4\n");

print "now we will build $version\n";

$binDir = "$stageDir/emp4/bin";

#build version
system("make_install.sh emp4 $version")
          && die "installing emp4 failed";

$mid_time = `date`;
print("copying files to bin\n");
system("cp $scriptsDir/test_emp.ex $binDir")
          && die "cannot copy: $!";

$built = "no";
$test_num = 1;
open(TFILES, "$testDir/$testList")
          || die "cannot open $testList: $!";
while (<TFILES>){
  chomp;
  $test = $_;                # a string
  if (length($test) >0) {
    chdir("$binDir") || die "cannot move to $binDir directory";

    if ($built eq "no"){

    $built = "yes";
    #kill existing empire server
    print "\tkilling existing emp_server if one exists...\n";
    system("$scriptsDir/kill_emp.sh");

    #removing data files
    print "\tremoving data files....\n";
    system ("rm -fr ../data");

    #copying data files
    print "\tcopying data files....\n";
    system ("cp -r $dataDir ..") && die "connot copy data files\n";

    #start server
    print "\tstarting empire server...\n";
    system("emp_server") && die "could not start server\n";
    system("sleep 1");

    }

    print "\tnow running $test on $version\n";

    #open file containing the grains of the test
    open(GFILE, "$univDir/$test") || die "cannot open $test: $!";

    while (<GFILE>){
    chomp;
    $grain = $_;
    if (length($grain) > 0) {
```

```
        #splits the string into an array of strings
        @grain = split(/ /, $grain);
        ($comm, $country, $pswd) = @grain;

        print ("cnty is $country, comm is $comm, pswd is $pswd\n");


        #run expect script
        system("expect test_emp.ex $country $pswd $univDir/GRAINS/$comm
                  $resultsDir/$test") && die "failed running $comm:
            $!";
      }
      }

      system("mkdir $resultsDir/results$vers_num/test$test_num")
                  && die "could not create directory: $!";
      system("mv $resultsDir/$test
            $resultsDir/results$vers_num/test$test_num/$test_num.out")
            && die "failed mv: $!";
      system("mv $stageDir/emp4/data
            $resultsDir/results$vers_num/test$test_num/data_$test_num"
            && die "failed cp: $!";
      # need to kill server to get .tr files
      system("$scriptsDir/softkill_emp.sh");

      # allow time for trace files to be written
      system("sleep 3");
      system("cp $AristotDir/empire.c.tr
            $resultsDir/traces$vers_num/$test_num.tr") && die "failed
            copy: $!";
      system("mv $AristotDir/empire.c.tr
            $resultsDir/results$vers_num/test$test_num/$test_num.tr")
            && die "failed mv: $!";
      system("echo tarring and zipping files....");
      system("$scriptsDir/gzutil.sh
            $resultsDir/results$vers_num/test$test_num&");
      $test_num = $test_num + 1;
      chdir("$stageDir") || die "cannot move to $stageDir: $!";
    }
  }

  # print out summary of how long testing this version took
  $vers_end = `date`;
  open (SUMMARY, ">$resultsDir/results$vers_num/summary.txt");
  print SUMMARY ("test suite started at $vers_begin\n");
  print SUMMARY ("test suite finished at $vers_end\n");
  close(SUMMARY);

  # tar and zip the directory of trace files
  system("$scriptsDir/gzutil.sh $resultsDir/traces$vers_num&");
  $vers_num = $vers_num + 1;

}

$end = `date`;

# print the summary of how long the tests took
open(OUT, ">$resultsDir/summary.txt");
```

```
print OUT ("test started at $begin");
print OUT ("build completed at $mid_time");
print OUT ("test ended at $end\n");
close(OUT);
```

Appendix B: make_install.sh

```
#!/bin/sh

# Title: make_install.sh

# Author: Brian Davia
# Project: Empire Regression Testing

# This is a shell script that checks out the specified version of
# a piece of software, builds, and installs it.  There are pieces of
# code that are specific to the program in which this was designed
# for: Empire.

#parameters:
#1-the project you are checking out of cfe, ex: emp4
#2-the version of the project that you are checking out of cfe
#     ex: v4_2_5

PROJECT=$1
VERSION=$2
WRK_DIR="/nfs/phantom/u8/davia2/EmpireTest/stage"
SYSTEM="solaris"
SCRIPTS="/nfs/phantom/u8/davia2/EmpireTest/scripts"

if [ "$PROJECT" = "" ]; then
      echo "ERROR, Project name not specified"
      exit 1
fi
if [ "$VERSION" = "" ]; then
      echo "ERROR, Version number not specified, will use most recent"
fi


#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Things to do before checking out a version of the project go here.
#     For our project, we must make sure that there is not a version
#     of our project currently running.

echo "Stopping Empire Server (if running)"
PID=`ps -e | grep emp_serv | awk '{print $1}'`
kill $PID > /dev/null 2>&1

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Do the checkout
# if the working directory doesn't exist we must first get a baseline
# before we can get any specific rev

if [ ! -d $WRK_DIR ]; then
      echo "$WRK_DIR doesn't exist, it will be created"
      mkdir $WRK_DIR
fi

echo "Checking out version $VERSION of $PROJECT to $WRK_DIR"
```

```
cd $WRK_DIR
pwd

# now get specific revision
if [ "$VERSION" != "" ]; then
      cvs checkout -r $VERSION $PROJECT
else
      cvs checkout $PROJECT
fi

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Things to do after the checkout and before building the project....
#     There are a few files and directories that are necessary
#     for the build to be successful.  We create them here.  We also
#     must edit the build.conf file that was checked out.  The
#     build.conf file holds a lot of system specific info, along with
#     info concerning the type of world we would like to build.

echo version is $VERSION

cd $PROJECT

# the Empire build process is flawed, these are ugly workarounds
echo mkdir data
mkdir data
echo touch data/econfig
touch data/econfig
echo touch data/econfig.bak
touch data/econfig.bak
echo mkdir bin
mkdir bin
echo mkdir lib
mkdir lib

# build empire binaries
echo "Editing build.conf"
find . -name build.conf | $SCRIPTS/edit_buildfiles.pl

# There are two versions that have slightly different file structures.
# We take care of that here.

if [ "$VERSION" = "v4_0_12" ] || [ "$VERSION" = "v4_0_13" ]; then
      echo special version
      cd emp4
      echo mkdir data
      mkdir data
      echo touch data/econfig
      touch data/econfig
      echo touch data/econfig.bak
      touch data/econfig.bak
      echo mkdir bin
      mkdir bin
      echo mkdir lib
      mkdir lib
```

```
fi

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Now build the project.

echo "Building Empire"
#gmake $SYSTEM
gmake inst-bt
# gmake depend
echo "Empire build complete"

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Things to do after building the project....
#     for some reason, older versions looks in xyz for data, so
#     we'll just move those things to xyz so that it runs properly.

mkdir xyz
cp data/* xyz

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Finished.
echo "Empire build and instalation complete"
```

Appendix C: test_emp.ex

```
# Title: test_emp.ex

# Author: Brian Davia
# Project: Empire Regression Testing

# This is a simple expect script that will log in to the empire
# client as a specified player.  It will then execute a specified
# sequence of commands.  The script looks for any of the possible
# prompts that are issued by Empire, and submits the commands listed
# in the file specified by comm (the third input parameter).  The
# script also keeps a log of events between it and the spawned process.
# This log is what the user would see if the commands were fed to
# Empire by hand.

# The following parameters are read in from the command line in order:
#     1) login name
#     2) login pswd
#     3) command to execute
#     4) name of logfile

set name [lindex $argv 0]     # user name
set pswd [lindex $argv 1]     # user password
set comm [lindex $argv 2]     # file of commands to be issued
set cout [lindex $argv 3]     # the destination of the log file
set timeout 90


send "name: $name\n"
send "pswd: $pswd\n"
send "comm: $comm\n"

# open the input file
set exFile [open "$comm" "r"]

# spawn a shell to start Empire
spawn csh

#open log file
log_file "$cout"

expect "%"
send "emp_client\r"
expect {
      "% $"             { exit }
      "Country name?"         { }
}
send "$name\r"
expect {
      "% $"             { exit }
      "Your name?"            { }
}
send "$pswd\r"
expect {
      "% $"             { exit }
      "Command : "            { }
}
```

```
# While we have commands to issue (from input file), look for a prompt
# and submit the command.  If we don't see any of the specified prompts
# then expect will timeout, and the program will exit.
while {[gets $exFile line] != -1} {
      send "$line\r"
      expect {
            timeout           { exit }
            "% $"             { }
            "Command : $"           { }
            "\\? $"                 { }
            "left: $"         { }
            "value : $"       { }
            "lot to buy: $"         { }
            "do you bid: $"         { }
            "Bomb what\\?*) $"      { }
            "Number of mil *:$"     { }
            "How many mil*:$" { }
            "attack with * ]$"      { }
            "assault with * ]$"     { }
            "bidding on: $"         { }
            "per unit: $"           { }
            "destination sector : $"      { }
            "<*:*:*> $"       { }
            "(days) $"        { }
            "<*: *,*> $"            { }
            "This setup ok\\? $"    { }
            "No such file or directory" { }
            "Execute :"       { }
            "<FLASH> $"       { }
            "(max *) : $"           { }
            "ynYNq?] $"       { }
      }
}

# quit emp_client
send "bye\r"

# exit the shell
expect "%"
send "exit\r"

#close log file
log_file
```

References

1. M. J. Balcer, T. J. Ostrand, The Category-Partition Method for Specifying and Generating Functional Tests, *Communications of the ACM*, June 1988 (pages 676 – 686).
2. J. Bible, G. Rothermel, A Unifying Framework Supporting the Analysis and Development of Safe Regression Test Selection Techniques, *Technical Report 99-60-11, Computer Science Department, Oregon State University*, December 1999.
3. Y. Chen, D. Rosenblum, and K. Vo, TestTube: A System for Selective Regression Testing, *Proceedings of the 16th International Conference on Software Engineering*, May 1994 (pages 211–222).
4. S. Elbaum, D. Gable, G. Rothermel, Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites, *Proceedings of the 7th International Software Metrics Symposium*, April 2001.
5. S. Elbaum, A. Malishevsky, G. Rothermel, Prioritizing Test Cases for Regression Testing, *Technical Report 00-60-03, Computer Science Department, Oregon State University*, February 2000.
6. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, G. Rothermel, An Empirical Study of Regression Test Selection Techniques, *ACM Transactions on Software Engineering and Methodology*, April 2001 (pages 184–208).
7. M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, M. Smith, Aristotle: a System for Development of Program Analysis Based Tools, *Proceedings of the ACM 33rd Annual Southeast Converence*, March 1995 (pages 110–119).
8. M. J. Harrold, G. Rothermel, A Safe, Efficient Regression Test set Selection Technique, *ACM Transactions on Software Engineering and Methodology*, April 1997 (pages 173–210).
9. J-M. Kim, A. Porter, G. Rothermel, An Empirical Study of Regression Test Application Frequency, *Proceedings of the 22nd International Conference on Software Engineering*, June 200 (pages 126–135).
10. D. Libes, *Exploring Expect*, O'Reilly & Associates, Inc., 1995.
10. Y. Liu, Regression Testing Experiments and Infrastructure, *Computer Science Department, Oregon State University*, August 1998.

11. Walter F. Tichy, Should Computer Scientists Experiment More? *Computer*, May 1998 (pages 32–40).
12. F. I. Vokolos, P.G. Frankl, Empirical Evaluation of the Textual Differencing Regression Testing Technique, *Proceedings of the International Conference on Software Maintenance*, November 1998 (pages 44–53).