UNIVERSITI SAINS MALAYSIA

Laporan Akhir Projek Penyelidikan Jangka Pendek

Development of an Automated Unit Testing Tool for Java Program

by
Dr. Kamal Zuhairi Zamli
Dr. Nor Ashidi Mat Isa

2009

UNIVERSITI SAINS MALAYSIA

LAPORAN AKHIR PROJEK PENYELIDIKAN JANGKA *PENDEK*

FINAL REPORT OF SHORT TERM RESEARCH PROJECT

Sila kemukakan laporan akhir ini melalui Jawatankuasa Penyelidikan di Pusat Pengajian dan Dekan/Pengarah/Ketua Jabatan kepada Pejabat Pelantar Penyelidikan

1. Nama Ketua Penyelidik: Dr Kamal Zuhairi Zamli
Name of Research Leader

[ ] Profesor Madya/ *Assoc. Prof.*   [✓ X] Dr./ *Dr.*   [ ] Encik/Puan/Cik *Mr/Mrs/Ms*

2. Pusat Tanggungjawab (PTJ):
School/Department

School of Electrical and Electronic Engineering

3. Nama Penyelidik Bersama:
Name of Co-Researcher

Dr Nor Ashidi Mat Isa

4. Tajuk Projek:
Title of Project

Development of An Automated Unit Testing Tool for Java Program

5. Ringkasan Penilaian/*Summary of Assessment*:

	Tidak Mencukupi *Inadequate* 1	2	Boleh Diterima *Acceptable* 3	Sangat Baik *Very Good* 4	5
i) **Pencapaian objektif projek:** *Achievement of project objectives*				X	
ii) **Kualiti output:** *Quality of outputs*			X		
iii) **Kualiti impak:** *Quality of impacts*			X		
iv) **Pemindahan teknologi/potensi pengkomersialan:** *Technology transfer/commercialization potential*			X		
v) **Kualiti dan usahasama :** *Quality and intensity of collaboration*			X		
vi) **Penilaian kepentingan secara keseluruhan:** *Overall assessment of benefits*			X		

6. **Abstrak Penyelidikan**
(Perlu disediakan di antara 100 - 200 perkataan di dalam **Bahasa Malaysia dan juga Bahasa Inggeris**. Abstrak ini akan dimuatkan dalam Laporan Tahunan Bahagian Penyelidikan & Inovasi sebagai satu cara untuk menyampaikan dapatan projek tuan/puan kepada pihak Universiti & masyarakat luar).

Abstract of Research
(An abstract of between 100 and 200 words must be prepared in Bahasa Malaysia and in English). This abstract will be included in the Annual Report of the Research and Innovation Section at a later date as a means of presenting the project findings of the researcher/s to the University and the community at large)

Please refer to the attachment report

7. **Sila sediakan laporan teknikal lengkap yang menerangkan keseluruhan projek ini. [Sila gunakan kertas berasingan]**
Applicant are required to prepare a Comprehensive Technical Report explaning the project. (This report must be appended separately)

Please refer to the attachment

Senaraikan kata kunci yang mencerminkan penyelidikan anda:
List the key words that reflects your research:

Bahasa Malaysia	*Bahasa Inggeris*
Alat Pengujian Unit Secara Automatik	Automated Unit Testing Tool
Pengujian Softwer	Software Testing

8. **Output dan Faedah Projek**
Output and Benefits of Project

(a) * **Penerbitan Jurnal**
Publication of Journals
(Sila nyatakan jenis, tajuk, pengarang/editor, tahun terbitan dan di mana telah diterbit/diserahkan)
(State type, title, author/editor, publication year and where it has been published/submitted)

Please refer to the attachment report

(b) **Faedah-faedah lain seperti perkembangan produk, pengkomersialan produk/pendaftaran paten atau impak kepada dasar dan masyarakat.**
State other benefits such as product development, product commercialisation/patent registration or impact on source and society.

Please refer to the attachment report

* Sila berikan salinan/*Kindly provide copies*

(c) **Latihan Sumber Manusia**
Training in Human Resources

i) Pelajar Sarjana:
Graduates Students
(Perincikan nama, ijazah dan status)
(Provide names, degrees and status)

Abdul Salim Daw Ahnissi, "SFIT Generator: Development fo Combinatorial Algorithms for Automatic Test Case Generation" (Msc Mixed Mode, Completed December 2007)

ii) Lain-lain:
Others

Two students are now completing their MSc.
- Mohd Firdaus Alias, Development of Change Impact Analysis Tool for Java Program

- Mohd Annuar Mat Isa, Development of an Automated Code Coverage Tool for Java Program

1 student is completing his PhD
- Mohammed Fadel Jamil Klaib, An Automated T-Way Strategy for Combinatorial Testing

[illegible]

28 January 2009

Tandatangan Penyelidik *Signature of Researcher*	**Tarikh** *Date*

Komen Jawatankuasa Penyelidikan Pusat Pengajian/Pusat
Comments by the Research Committees of Schools/Centres

The project has generated the following outputs:

1. International Journal Publications (3)
2. International Conference Publications (3)
3. Local Conference Publications (3)
4. Awards in Local Exhibition (1)
5. Awards in International Exhibition (1)
6. MSc Mixed Mode Dissertation (1)

Two eScience Fund projects have been secured based on the work done in this project (Kindly refer to the attachment report)

[illegible]

Dr [illegible]
Deputy Dean
Postgraduate And Research
[illegible] of Electrical and [illegible] Engineering
[illegible]

TANDATANGAN PENGERUSI
JAWATANKUASA PENYELIDIKAN
PUSAT PENGAJIAN/PUSAT
Signature of Chairman
[Research Committee of School/Centre]

28/1/09

Tarikh
Date

JABATAN BENDAHARI
UNIT KUMPULAN WANG AMANAH
UNIVERSITI SAINS MALAYSIA
KAMPUS KEJURUTERAAN
SERI AMPANGAN
PENYATA KUMPULAN WANG

TEMPOH BERAKHIR 31/12/2008

DR KAMAL ZUHAIRI ZAMLI

304.PELECT.6035195

JUMLAH GERAN :- 19,985.00

NO PROJEK :-

PANEL :- J/PENDEK

PENAJA :- JANGKA PENDEK

Tempoh Projek:15/07/2006 - 14/07/2008

DEVELOPMENT OF AN AUTOMATED UNIT TESTING TOOL FOR JAVA PROGRAM

Vot	Peruntukan (a)	Perbelanjaan sehingga 31/12/2007 (b)	Tanggungan semasa 2008 (c)	Perbelanjaan Semasa 2008 (d)	Jumlah Perbelanjaan 2008 (c + d)	Jumlah Perbelanjaan Terkumpul (b+c+d)	Baki Peruntukan Semasa 2008 (a-(b+c+d)
11000: GAJI KAKITANGAN AWAM	7,500.00	7,838.13	0.00	(713.35)	(713.35)	7,124.78	375.22
21000: PERBELANJAAN PERJALANAN DAN SARAHID	3,800.00	3,544.50	0.00	510.00	510.00	4,054.50	(254.50)
23000: PERHUBUNGAN DAN UTILITI	650.00	0.00	0.00	0.00	0.00	0.00	650.00
27000: BEKALAN DAN ALAT PAKAI HABIS	4,292.00	2,000.00	0.00	1,629.00	1,629.00	3,629.00	663.00
28000: PENYELENGGARAAN & PEMBAIKAN KECIL	750.00	0.00	0.00	0.00	0.00	0.00	750.00
29000: PERKHIDMATAN IKTISAS & HOSPITALITI	2,993.00	5,106.93	0.00	0.00	0.00	5,106.93	(2,113.93)
	19,985.00	18,489.56	0.00	1,425.65	1,425.65	19,915.21	69.79
Jumlah Besar	**19,985.00**	**18,489.56**	**0.00**	**1,425.65**	**1,425.65**	**19,915.21**	**69.79**

304.PELECT.6035195

UNIVERSITI SAINS MALAYSIA

USM Short Term Grants –
Development of An Automated Unit Testing Tool for Java Program

Final Report

Period: 15 July 2006 -14 July 2008

Document identifier :	**304.PELECT.6035195**
Date:	**15 January, 2009**
Version:	**1.0**
Document status:	**Final Report**
Author	**Dr Kamal Zuhairi Zamli**

Signature of Project Leader

[illegible]

(Dr Kamal Zuhairi Zamli)

PPKEE

Abstract

Software testing relates to the process of executing a program or system with the intent of finding errors. Covering as much as 25 to 35 percent of the development costs and resources, software testing is an integral part of the software development lifecycle. Despite its importance, current software testing practice lacks automation and is still primarily based on highly manual processes from the generation of test cases (i.e. from the specifications documents) up to the actual execution of the test. Although the emergence of helpful automated testing tools in the market is blooming, their adoptions are lacking as they do not adequately provide the right level of abstraction and automation required by test engineers.

JTst is a Java based automated unit testing tool that addresses some of the aforementioned issues. The main novel features of JTst are the fact that it provides a sound methodology and work context for the testing process as well as permits automated combinatorial test case generations and parallel execution for Java classes, enabling higher product quality at lower testing costs.

Keywords: Automated Unit Testing Tool, Software Testing

Abstrak

Pengujian softwer melibatkan proses melarikan sesuatu program atau sistem dengan tujuan untuk mengesan kesalahan. Dengan kos pembangunan dan sumber meliputi 25 hingga 30 peratus, pengujian adalah antara bahagian utama dalam kitar hayat pembangunan softwer. Walaupun ia penting, kaedah dan praktis untuk pengujian softwer semasa masih bergantung kepada proses manual meliputi aktiviti penghasilan kes ujian kepada pengujian kes tersebut. Walaupun terdapat semakin banyak perkembangan alatan untuk pengujian, penerimaan mereka adalah sedikit kerana gagal memberikan kemudahan dan automasi yang secukupnya kepada jurutera pengujian.

JTst adalah peralatan pengujain softwer yang dapat meringankan masalah yang telah diterangkan sebelumnya itu. Sumbangan utama JTst meliputi pembangunan metodologi dan konteks kerja serta membenarkan penerbitan kombinasi ujian kes dan larian serentak untuk menghasilkan produk sofwer yang berkualiti pada kos yang lebih rendah.

Kata Kunci: Alat Pengujian Unit Secara Automatik, Pengujian Sofwer

Introduction

Computing technology has gone a long way since the first Babbage computer in 1871. Today, many chores that were once manual have been taken over by computers. Factories use computers to control manufacturing equipments. Electronics manufacturing use computers to test everything from microelectronics to circuit card assemblies. The automation provided by computers avoids the errors that humans make when they get tired after multiple repetitions.

Software testing is one area which can also benefit from automation. According to Glen Myers [17], testing is the process of executing a program with the intent of finding errors. Testing covers several aspects:

- Unit testing is carried out during the programming activity. It makes sure that each elementary element (e.g. modules, methods) has a correct behavior, and aims at avoiding errors in these elementary elements during execution.
- Functional testing (i.e. integration testing) aims at ensuring correctness of operations and their conformance to the functional requirements.
- Performance testing (load testing or stress testing), aims at ensuring system performance when subjected to significant competition in the access to resources (e.g. processor, memory, disk, and network)

Although an important part of software development (i.e. covering as much as 25 to 35 percent of the development costs [8,19]), current software testing practice is still based on highly manual processes from the generation of test cases (i.e. from the specifications documents) up to the actual execution of the test. These manually generated tests are sometimes executed using *ad hoc* approach, typically requiring the construction of a test driver for the particular application under test [7, 17, 19]. The construction of a test driver puts extra burden to test engineers especially if the test cases are significantly large.

Additionally, test engineers are also under pressure to test increasing lines of code in order to meet market demands for more software functionalities. In order to attain the required level of quality, test engineers need to maintain high test coverage. Viewing from the aforementioned perspectives, testing is a tedious and error prone process. While there are significant proliferations of helpful automated testing tools in the market, much of which does not adequately provides the right level of abstraction and automation as required by test engineers [22].

In order to address some of this issue, this paper proposes a new software testing tool, called JTst, based on the use of Java technology [10]. The main aim of JTst is to provide high level of abstraction for testing as well as permit automated combinatorial test case generations and execution for Java classes, and hence allow higher product quality at lower testing costs.

Addressing the aforementioned aim, the objectives of this project are:

1. To build an automated unit testing tool, called JTst, for Java programs.
2. To identify the main characteristics and requirements for an automated unit testing tool.

3. To demonstrate the feasibility of developing JTst using mixed programming language with C++ and Java.
4. To utilize an object-oriented analysis and design techniques using the Unified Modeling Language (UML) for designing JTst.

Related Work

Concerning related work, the following paragraphs survey the current state-of-the-art on automated Java testing tools. This survey is based on our earlier work described in [2, 3, 4]).

- **Jaca**

  Jaca [16], developed at the State University of Campinas, is an automated testing tool that permits testing of Java classes by corrupting the method interfaces and attributes. Jaca does not require the application's source code, but it needs the some information about the application such as class name and method interfaces.

- **JUnit**

  JUnit [11,15] is a testing tool used to write and run automated and repeatable tests. In JUnit, test engineer need to write a unit test case, essentially a collection of tests designed to verify the behavior of a single unit within a user program. The unit test case can then be automatically executed by the JUnit environment.

- **FIONA**

  FIONA [20] is an automated software testing tool for distributed Java application. FIONA provides a Java Virtual Machine Tool Interface that enables the inspection and execution of faults of distributed application running in the Java Virtual Machine.

- **SFIT**

  SFIT [2, 3, 4, 21, 22, 23] is an automated software testing tool, developed in USM, for evaluating Java COTs in the absence of source code. Although the current version is lacking in terms of software components integration and GUI support, SFIT can permit up to 2500 test cases to be defined and automatically executed in a single click of a button.

- **Simple**

  Simple [1] is automated functional testing tool that can be used to assess reliability, robustness and performance of a system as a whole. The aim of simple is to facilitate testing of Java classes used in safety critical applications.

Requirements for an Automated Java Unit Testing Tool

Based on the aforementioned survey, the common characteristics of these tools can be summarized as follows:

i. Requires high level abstraction

Ideally, the testing tool should not assume that the user has significant knowledge of Java in order to be able to use the tool. In fact, a helpful tool should be sufficiently high level to facilitate the testing process in the sense that test engineers need not need to do any coding whatsoever in order to perform the actual testing.

ii. Supports testing in the absence of source code

In line with the current trends of using commercial-of-the-shelves (COTs) components to speed up software development, there is a need for an automated testing tool to be able to perform the functional and unit testing even in the absence of source code.

iii. Enables test automation

Test automation relieves the test engineer from routine task as well as allows multiple test cases to be executed in a single experiment. Additionally, test automation provided by the tools must also be sufficiently intuitive for the test engineers to master. Providing some level of intuition is important to help junior engineers to grasp the testing work context particularly in terms of how each testing activity fits together in the whole picture.

In general, test automation can come in a number of forms. In a nut shell, the test automation should relieve the test engineer from the routine tasks of creating Java test drivers for execution. In addition, test automation should also facilitate the generation and execution of the actual test cases. Here, parallel execution of test cases can help to speed up the testing process. In this manner, test engineers can put significant focus on the job at hand (i.e. coming up with good test cases) and be released from manually writing test drivers.

iv. Provides Graphical User Interface Support

GUI can help improve usability of the test tool. Often, GUI interface is better that command line interface as far as ease of use is concerned.

v. Permits extended test coverage

Ideally, the more test coverage the better quality the software is. To ensure high test coverage, test cases need to cover all the control flow paths. Nevertheless, in the absence of source code (e.g. COTs), covering all control flow paths is difficult. One way of alleviating from this difficulty is to generate as much as possible new test cases, for instance, through combinational test cases that are deduced from the given test cases. With combinational test cases, a set of new and unique combination of test cases can be produced in order to help extends the test coverage.

Using the aforementioned characteristics, Table 1 presents a digest of the Java based automated software testing tools discussed earlier.

Legends √ Implemented feature X Not implemented feature ☐ Not enough information	Automated Java Testing Tool				
Characteristics	JACA	JUNIT	FIONA	SFIT	SIMPLE
High level abstraction	√	X	X	√	X
Testing in the absence of source code	√			√	
Test automation	√	√	√	√	√
Graphical User Interface support	√	√	√	√	√
Extended test coverage	X	X	X	X	X

Table 1 – Summary of Automated Java Testing Tool

Referring to Table 1, it can be observed that no single tool support all the characteristics identified earlier. In fact, no single tool surveyed here supports extended test coverage using combinatorial approach discussed earlier. Thus, developing a new and automated Java testing tool with the abovementioned features would be an appealing task. It is the development of such a tool, called JTst, is the main focus of this research.

Introducing JTst

A key idea in JTst is the fact that tests are performed based on the values of the interface parameters (i.e. on the data types of the parameter lists) and not on the behavioral specifications. Thus, JTst is suitable for performing automated black box testing particularly involving commercial-of-the-shelves-components (COTs) where no source code and design are usually available apart from some user documentations.

Another key idea in JTst is that the test cases can be combinatorially generated based on some *base test cases*, the concept borrowed from Ammann and Offutt [5]. Ideally, these base test cases can either be collected from known combination of input variables that causes failures to the module under test from real systems in various application domains [12] or from the program specifications (see Figure 1).

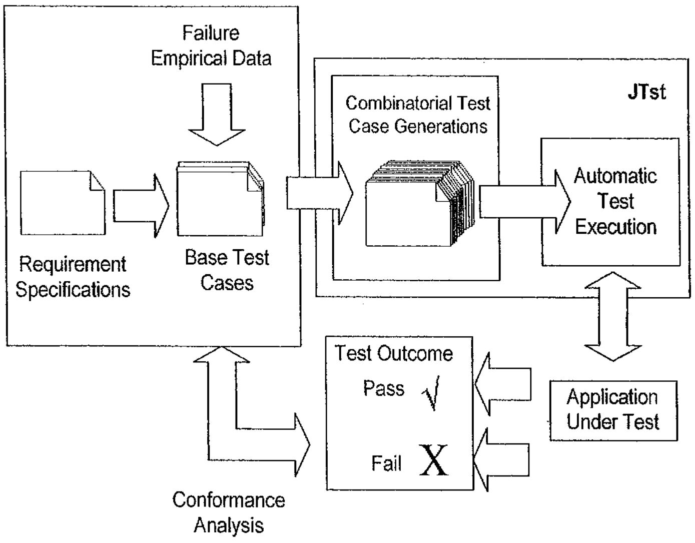


Figure 1 – JTst Overview

Kuhn and Okum [13] suggest that from empirical observation, the number of input variables involved in software failures is relatively small (i.e. in the order of 3 to 6), in some classes of software. If t or fewer variables are known to cause fault, test cases can be generated on all *t-way* combinations of discrete values (i.e. using equivalent class or boundary value analysis for continuous value variables). Empirical evidence suggests that in some software implementation, the execution of these test cases can typically uncover 50% to 75% of faults in a program [9, 14].

Combinatorial techniques are known to causes test data explosion [24, 25, 26]. Suppose that test input variables are 10, each had 3 values say 0, 1, and 2. Then, there are 3^{10} = 59,049 possible parameter combinations. Now, if the test input variables are increased to 13, then there are 3^{13} = 1,594,323 possible parameter combinations. This simple example illustrates that a small change in the input parameters can cause massive increase in the parameter combinations.

While exhaustive testing for all combinations would be impractical, partitioning the combinations might still be useful in order to detect faults. For this reason, JTst permits the concurrent execution of the test cases to alleviate bottle neck in testing combinatorial test cases (discussed later).

Concerning its implementation, JTst consists of a number of related components consisting of the Class Inspector; the Test Editor; the Test Combinator; the Automated Loader; and the Data Logger/Log (see Figure 2). The working context (i.e. process flow) for these components is summarised in Figure 3. The functionalities for each of these components will be discussed next.

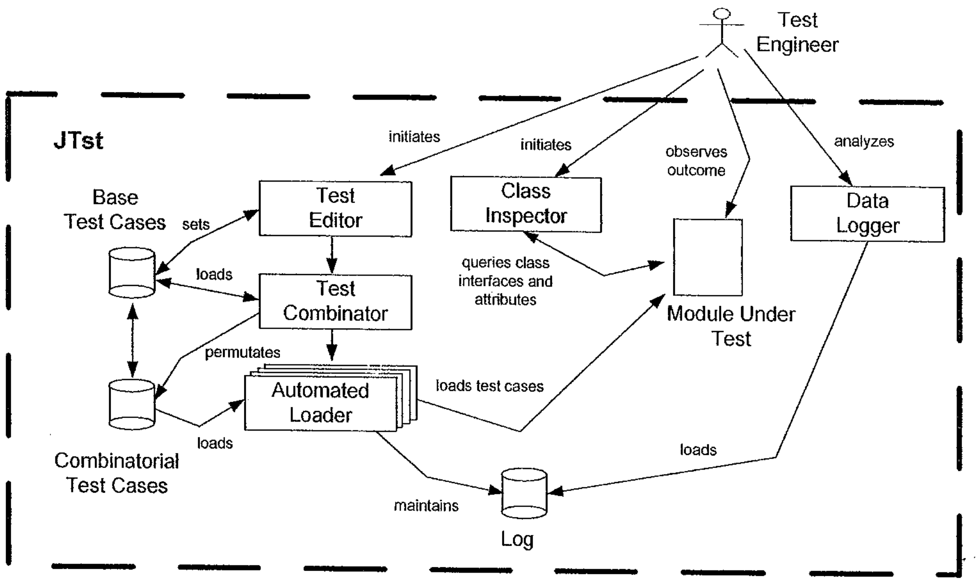


Figure 2 – JTst Main Components

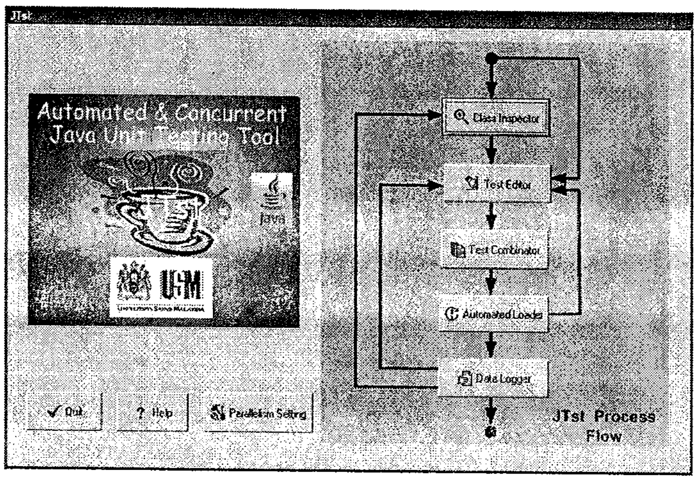
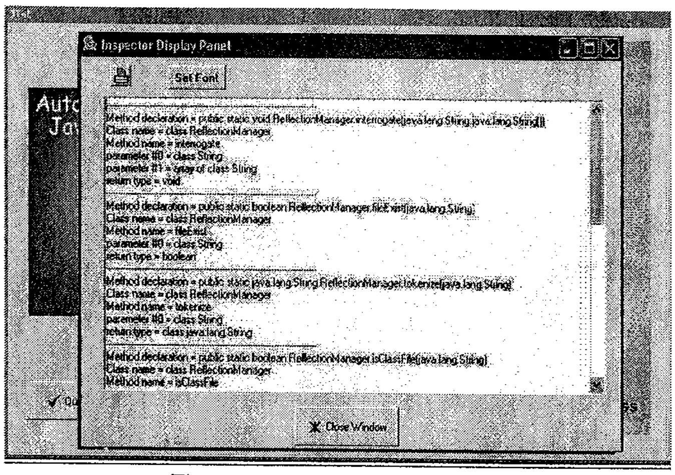


Figure 3 – JTst Work Context

Class Inspector

One useful feature of JTst is to allow unit testing in the absence of source codes. In this case, the class inspector can optionally be used to obtain details information of the Java class interface in order to permit black box testing. To do so, the class inspector exploits Java Reflection API in order interrogate Java classes for method interfaces including public, private, and protected ones (see Figure 4). This information can be used to set up the base test cases in the fault file (discussed next).



```
Method declaration = public static void ReflectionManager.interrogate(java.lang.String,java.lang.String[])
Class name = class ReflectionManager
Method name = interrogate
parameter #0 = class String
parameter #1 = array of class String
return type = void

Method declaration = public static boolean ReflectionManager.fileExist(java.lang.String)
Class name = class ReflectionManager
Method name = fileExist
parameter #0 = class String
return type = boolean

Method declaration = public static java.lang.String ReflectionManager.tokenize(java.lang.String)
Class name = class ReflectionManager
Method name = tokenize
parameter #0 = class String
return type = class java.lang.String

Method declaration = public static boolean ReflectionManager.isClassFile(java.lang.String)
Class name = class ReflectionManager
Method name = isClassFile
```

Figure 4 – Interrogating Java Class

Test Editor

As its name suggests, the test editor allows the user to edit and setup the test cases (i.e. including the base test cases) in a JTst *fault file*. Here, the test case definition follows certain predefined formatting rules in order to facilitate the parsing of data for automatic recombination (see Figure 5).

```
@FaultFile
//////////////////////////////////////
    Common Header Definition
//////////////////////////////////////
classname : adder
methodname : add_basictypes_integer
specifier: private
paramtypes : 2
returntype: int
parameter : partypes[0]=Integer.TYPE
parameter : partypes[1]=Integer.TYPE

//////////////////////////////////////
    Body - Test case 0
//////////////////////////////////////
arglist:arglist[0]=new Integer(Integer.MAX_VALUE)
arglist : arglist[1]=new Integer(Integer.MAX_VALUE)

//////////////////////////////////////
    Body - Test case 1
//////////////////////////////////////
arglist:arglist[0]=new Integer(Integer.MIN_VALUE)
arglist : arglist[1]=new Integer(Integer.MIN_VALUE)

...............
```

Figure 5 – Sample Fault File

Test Combinator

Test combinator manipulates the base test case in order to generate combinatorial test cases (see Figure 6).

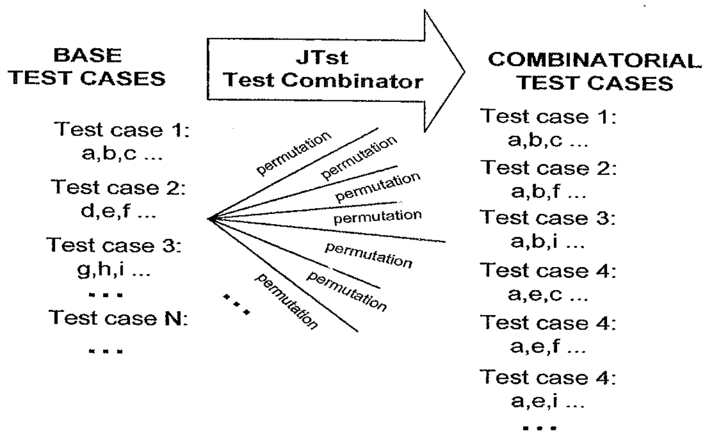


Figure 6 – JTst Test Combinator

To illustrate the JTst combinator algorithm, consider the following running example. Let a method M_1 has four inputs variables X = {A,B,C,D}. For simplicity sake, let us assume that the base test case for M_1 has been identified in Table 1.

Table 2 – Base Data Values for Method M_1

	Input Variables			
	A	B	C	D
Base Values	T1	99	C1	Large
	T2	2000	C3	Small

The test cases data can be viewed as a matrix with specified columns and rows. Here, one can traverse one column at a time (called *sensitivity* variable in JTst implementation), whilst keeping other column fixed to permutate and generate new test cases from existing ones. Table 3 depicts the possible combinatorial test cases with the sensitivity variable set to A.

Table 3 – Base and Combinatorial Data Values for Method M_1 With *Sensitivity* = A

	Input Variables			
	A	B	C	D
Base Values	T1	99	C1	Large
	T2	2000	C3	Small
Combinatorial Values	T1	2000	C3	Small
	T2	99	C1	Large

The complete pseudo code for single column sensitivity is given as follows (see Figure 7):

```
begin
  set the input variable as sensitivity variable, V
  for each defined test case till the end
     begin
      hold test case value for non sensitivity variable, D
      for each value of the sensitivity variable, P
        begin
          combine P,D
          get the next value of P
        end
      get the next value of D
     end
end
```

Figure 7 – Single Column Sensitivity Algorithm

Here, the algorithm starts by setting the input variable V, which corresponds to the specific input column (e.g. if v=0 a sensitive variable will be column 0). For each defined test case until the end, the algorithm holds the test case value for non-sensitivity variable D (i.e. the values for the other variables). The inner loop varies all the sensitivity variable values of P and combines to that of the non sensitive variable D.

Apart from permitting sensitivity column to be a single column, JTst combinator algorithm also allows the sensitivity column to be a combination of 2 or more columns or all columns (see Table 4). Here, repetitive test cases are automatically removed.

Table 4 – Base and Combinatorial Data Values for Method M_1 With *Sensitivity* = All Variables

	Input Variables			
	A	B	C	D
Base Values	T1	99	C1	Large
	T2	2000	C3	Small
Combinatoria l Values	T1	2000	C3	Small
	T2	99	C1	Large
	T1	2000	C1	Large
	T2	99	C3	Small
	T1	99	C3	Large
	T2	2000	C1	Small
	T1	99	C1	Small
	T2	2000	C3	Large

The pseudo code for multiple column sensitivity is given as follows (see Figure 8):

```
begin
 for each input variable
  begin
   set the input variable as sensitivity variable V
   for each defined test case till the end
    begin
      hold test case value for non sensitivity variable D
      for each value of the sensitivity variable P
         begin
           combine P,D
           if (P,D) combination already exist then
           reject P,D combination
           get the next the value of P
         end
      get the next value of D
    end
   assign the next sensitivity variable V
  end
end
```

Figure 8 – All Column Sensitivity Algorithm

Concerning the number of generated test cases, combinatorial recombination can generate new test cases following the following formulae.

In the case of a single column, provided that all the base data values are unique, recombination can regenerate new test cases as follows:

The no of new test cases = n^2
where n = number of completely define test cases

In the case of all columns, provided that all base data values are unique, recombination can regenerate new test cases based on:

The no of new test cases = $(p*n^2) - \phi$
where n = the number of completely define test cases
p = the number of input variables
ϕ = the number of repetitive combinatorial values
= $n*(p-1)$

Automated Loader

JTst automated loader have two main responsibilities. The first responsibility is to iteratively parse the test cases (defined in JTst *fault files*), and automatically generates and executes the appropriate Java code driver. The second responsibility is to manage concurrent execution of test cases. Here, the JTst automated loader is actually consists of two sub-components: Loader and Concurrent Manager (see Figure 9)

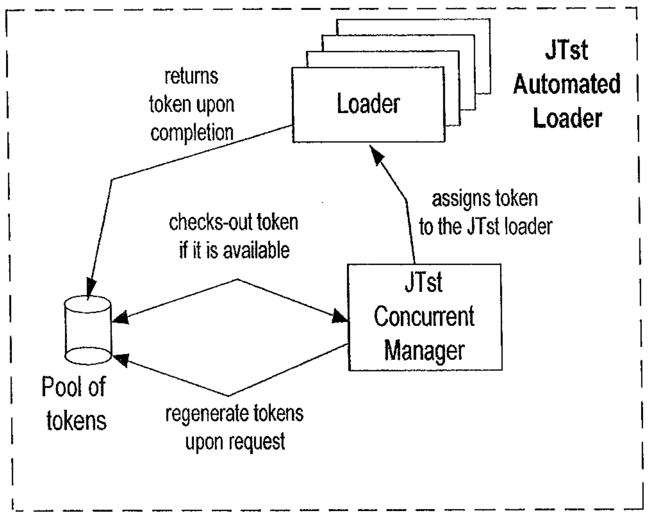
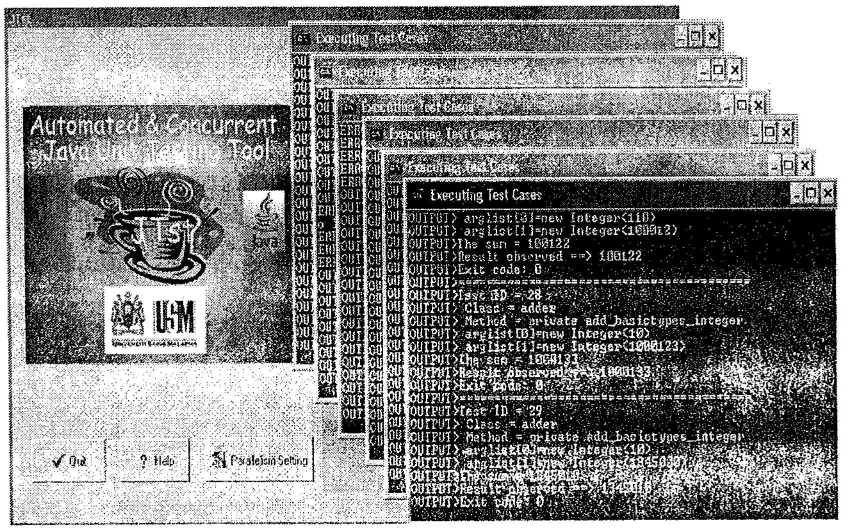

JTst Automated Loader
returns token upon completion
Loader
assigns token to the JTst loader
checks-out token if it is available
JTst Concurrent Manager
Pool of tokens
regenerate tokens upon request

Figure 9 – JTst Automated Loader & Token Passing Mechanism

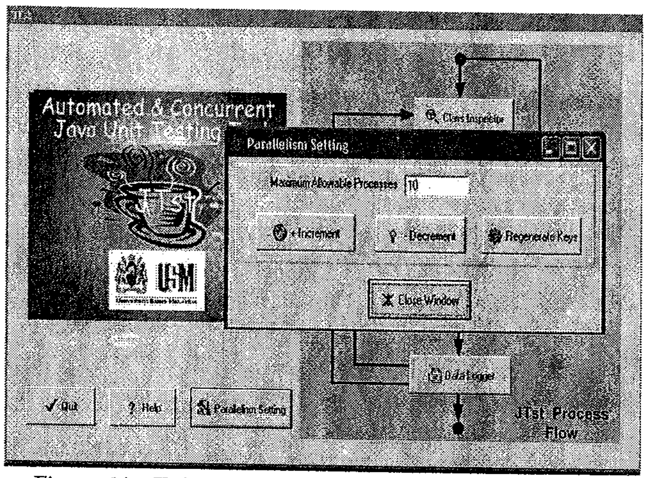
Concurrent execution is achieved in JTst through a well-known token passing algorithm. Sample concurrent execution of test cases is shown in Figure 10. In the current version, JTst has been tested to concurrently execute up to 15,000 test cases per execution.



```
OUTPUT> arglist[0]=new Integer(110)
OUTPUT> arglist[1]=new Integer(100012)
OUTPUT>The sum = 100122
OUTPUT>Result observed ==> 100122
OUTPUT>Exit code: 0
OUTPUT>==============================================
OUTPUT>Test ID = 28
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(10)
OUTPUT> arglist[1]=new Integer(100123)
OUTPUT>The sum = 100133
OUTPUT>Result observed ==> 100133
OUTPUT>Exit code: 0
OUTPUT>==============================================
OUTPUT>Test ID = 29
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(10)
OUTPUT> arglist[1]=new Integer([illegible])
OUTPUT>[illegible]
OUTPUT>Result observed ==> [illegible]
OUTPUT>Exit code: 0
```

Figure 10 – Concurrent Execution of Test Data

Here, a token is always associated for each concurrent execution. Once all the tokens have been used up, no further concurrent execution is allowed until one or more concurrent executions have terminated (i.e. release its token). Here, the number of defined tokens in the pool of tokens can be dynamically configured through the user interface provided should the need arise. Obviously, the more tokens are allowed, the slower the test case executions will be. This token setting can be illustrated in Figure 11.



Maximum Allowable Processes 10

Figure 11 – Token Generation for Concurrent Execution

Data Logger/Log

Data logger is a text browser with customised search capability to perform offline analysis of the output captured by the automated loader (see Figure 2) in the form of logs. Here, logs are special database storing the input output behavior of the module under test (MUT). If the specification of the MUT method exists, conformance analysis can be made using this database.

Nevertheless, in the absence of source codes and formal specification, the trivial outcome of "doesn't hang and doesn't crash" suffices to determine whether MUT passes the minimum testing requirement. In this case, the operating system can be queried if the test program terminates abnormally and a process monitor can be employed to detect hangs. A key issue here is the fact that the faults can always be reproducible with the same sets of inputs.

JTst Implementation Summary

In this research project, JTst has been implemented using Borland C++ Builder 6.0 and Java Development Kit 1.5. Figure 12 depicts the overall JTst class diagram expressed using the Unified Modeling Language (UML) notation.

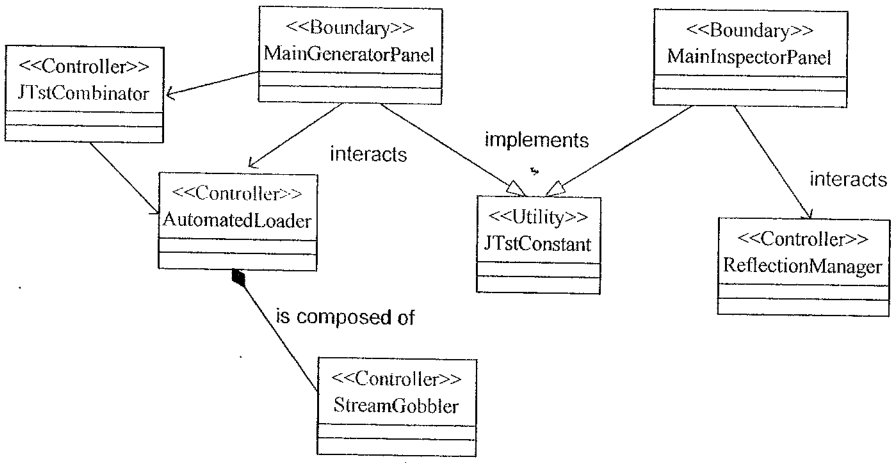

<<Boundary>> MainGeneratorPanel
<<Boundary>> MainInspectorPanel
<<Controller>> JTstCombinator
interacts
implements
interacts
<<Controller>> AutomatedLoader
<<Utility>> JTstConstant
<<Controller>> ReflectionManager
is composed of
<<Controller>> StreamGobbler

Figure 12 – JTst Class Diagram Implementation

The graphical user interfaces are implemented in the MainGeneratorPanel and the MainInspectorPanel class. The JTst combinatory implements both the single column and all column sensitivity algorithms. The class inspector functionalities are implemented by the ReflectionManager. The ReflectionManager class heavily relies on the Reflection API for performing most of its functionalities. The loader and generator, concurrent execution functionalities are handled by the AutomatedLoader. At a glance, due to the advantage of separation of concern, one may question the fact that both the loader and generator and the concurrent execution functionalities are combined into a single class (i.e. in the AutomatedLoader class). Nevertheless, a counter argument suggests that combining both the loader generator and concurrent execution functionalities in a single class facilitates implementation as well as improves cohesion because both components are tightly coupled with each other. If there is a need to change the loader generator, it is highly likely that the same change need to be propagated through the concurrent execution functionalities as the components need to rely on the Reflective API to collaborate with each other. StreamGobbler implements the database file access. As seen in Figure 12, the StreamGobbler class is declared to be a composition of the AutomatedLoader class. The rationale for using the composition relationship is that the injector needs to "use" the StreamGobbler every time it needs to perform the database access. Thus, the AutomatedLoader and StreamGobbler class share the same lifetime (i.e. if the AutomatedLoader object exists, so does the StreamGobbler object). Finally, as the name suggests, the JTstConstant class defines all the JTst constant.

Experimentations with JTst

In this section, JTst will be evaluated in terms of its correctness as well as its applicability as a helpful unit testing tool. To demonstrate its correctness, it is necessary to prove the the algorithms behave as expected, that is, in terms of the conformance from expected results and the actual results. Now, to demonstrate its applicability, a simple step-by-step case study evaluation will be demonstrated.

Demonstration of Correctness

Two experiments will be discussed here involving the algorithm for single column sensitivity, and algorithm for all column sensitivity. In each experiment, the following input data will be used (see Table 4.1). The rationale for using these data inputs stemmed from the fact that historically the same data inputs have been used by other researchers in the area. By adopting the same data inputs, objective comparison may be made amongst different algorithm implementation.

Table 5 – Base Data Input

Parameter 1	Parameter 2	Parameter 3	Parameter 4
Netscape	Window	LAN	Local
IE	Macintosh	PPP	Networked
Other	Linux	ISDN	Screen

Experiment 1: Demonstration of Correctness for Single Column Sensitivity Algorithm

In this experiment, parameter 2 is arbitrarily selected to be the sensitivity variables. Based on the input data in Table 5, the fault file for this input is as follows.

```
@FaultFile
//////////////////////////////////////
  Class information
//////////////////////////////////////
classname : not specified
methodname : not specified
specifier: 4
paramtypes : not specified
returntype: not specified
parameter : partypes[0]=String.class
parameter : partypes[1]=String.class
parameter : partypes[2]=String.class
parameter : partypes[3]=String.class

//////////////////////////////////////
  Test case 0
//////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

//////////////////////////////////////
  Test case 1
//////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

//////////////////////////////////////
  Test case 2
//////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen
```

Figure 13 – Fault File for Experiment 1

Given the above fault files, it is expected that the combinatorial test cases would yeild n^2 or $3^2 =$ 9 test cases. The 9 expected results are depicted in the Table 6 below.

Table 6 – Expected Results for Experiment 1

	Parameter 1	Parameter 2	Parameter 3	Parameter 4
Base Values	Netscape	Windows	LAN	Local
	IE	Macintosh	PPP	Networked
	Other	Linux	ISDN	Screen
Combinational Values	Netscape	Windows	LAN	Local
	Netscape	Windows	PPP	Local
	Netscape	Windows	ISDN	Local
	IE	Macintosh	LAN	Networked
	IE	Macintosh	PPP	Networked
	IE	Macintosh	ISDN	Networked
	Other	Linux	LAN	Screen
	Other	Linux	PPP	Screen
	Other	Linux	ISDN	Screen

Using JTst, the following output fault file is generated.

```
@FaultFile
//////////////////////////////////////
  Class information
//////////////////////////////////////
classname : not specified
methodname : not specified
specifier: not specified
paramtypes : 4
returntype: not specified
parameter : partypes[0]=String.class
parameter : partypes[1]=String.class
parameter : partypes[2]=String.class
parameter : partypes[3]=String.class

//////////////////////////////////////
  Test case 0
//////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

//////////////////////////////////////
  Test case 1
//////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= PPP
arglist : arglist[3]= Local

//////////////////////////////////////
  Test case 2
//////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= ISDN
arglist : arglist[3]= Local

//////////////////////////////////////
  Test case 3
//////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= LAN
arglist : arglist[3]= Networked

//////////////////////////////////////
  Test case 4
//////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

//////////////////////////////////////
  Test case 5
//////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= ISDN
arglist : arglist[3]= Networked

//////////////////////////////////////
  Test case 6
//////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= LAN
arglist : arglist[3]= Screen

//////////////////////////////////////
  Test case 7
//////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= PPP
arglist : arglist[3]= Screen

//////////////////////////////////////
  Test case 8
//////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen
//////////////////////////////////////
```

Figure 14 – Output Fault File for Experiment 1

Referring to Table 6 and the output fault file in Figure 14, it can be seen that the expected output as well as the actual output matches. Therefore, the JTst for single column sensitivity algorithm demonstrated to be correct. Although not shown here, the sensitivity column can also be column 0 or any other desired columns.

Experiment 2: Demonstration of Correctness for All Columns Sensitivity Algorithm

Here, all parameters have been selected to be the sensitivity variables. Based on the input data given in Table 5, the fault file for this input is as follows.

```
@FaultFile
////////////////////////////////////////
  Class information
////////////////////////////////////////
classname : not specified
methodname : not specified
specifier: not specified
paramtypes : 4
returntype: not specified
parameter : partypes[0]=String.class
parameter : partypes[1]=String.class
parameter : partypes[2]=String.class
parameter : partypes[3]=String.class

////////////////////////////////////////
  Test case 0
////////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

////////////////////////////////////////
  Test case 1
////////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

////////////////////////////////////////
  Test case 2
////////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen
```

Figure 15 – Fault File for Experiment 2

Given the above fault file, it is expected that the combinatorial test cases would be $(p * n^2) - n*(p-1)$ or $(4*3^2) - 3(4-1) = 27$ test cases. The 27 expected results are depicted in the Table 7 below.

Table 7 – Expected Results

	Parameter 1	Parameter 2	Parameter 3	Parameter 4
Base Values	Netscape	Windows	LAN	Local
	IE	Macintosh	PPP	Networked
	Other	Linux	ISDN	Screen
Column 0	Netscape	Windows	LAN	Local
	IE	Windows	LAN	Local
	Other	Windows	LAN	Local
	Netscape	Macintosh	PPP	Networked
	IE	Macintosh	PPP	Networked
	Other	Macintosh	PPP	Networked
	Netscape	Linux	ISDN	Screen
	IE	Linux	ISDN	Screen
	Other	Linux	ISDN	Screen
Column 1	Netscape	Macintosh	LAN	Local
	Netscape	Linux	LAN	Local
	IE	Windows	PPP	Networked
	IE	Linux	PPP	Networked
	Other	Windows	ISDN	Screen
	Other	Macintosh	ISDN	Screen
Column 2	Netscape	Windows	PPP	Local
	Netscape	Windows	ISDN	Local
	IE	Macintosh	LAN	Networked
	IE	Macintosh	ISDN	Networked
	Other	Linux	LAN	Screen
	Other	Linux	PPP	Screen
Column 3	Netscape	Windows	LAN	Networked
	Netscape	Windows	LAN	Screen
	IE	Macintosh	PPP	Local
	IE	Macintosh	PPP	Screen
	Other	Linux	ISDN	Local
	Other	Linux	ISDN	Networked

Employing JTst, the following output file is generated.

```
@FaultFile
/////////////////////////////////////
  Class information
/////////////////////////////////////
classname : not specified
methodname : not specified
specifier: not specified
paramtypes : 4
returntype: not specified
parameter : partypes[0]=String.class
parameter : partypes[1]=String.class
parameter : partypes[2]=String.class
parameter : partypes[3]=String.class
```

```
////////////////////////////////////
  Test case 0
////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

////////////////////////////////////
  Test case 1
////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

////////////////////////////////////
  Test case 2
////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

////////////////////////////////////
  Test case 3
////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

////////////////////////////////////
  Test case 4
////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

////////////////////////////////////
  Test case 5
////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

////////////////////////////////////
  Test case 6
////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen
```

```
/////////////////////////////////////
  Test case 7
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 8
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 9
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Macintosh
arglist : arglist[2]= LAN
arglist : arglist[3]= Local

/////////////////////////////////////
  Test case 10
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Linux
arglist : arglist[2]= LAN
arglist : arglist[3]= Local
/////////////////////////////////////
  Test case 11
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Windows
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

/////////////////////////////////////
  Test case 12
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Linux
arglist : arglist[2]= PPP
arglist : arglist[3]= Networked

/////////////////////////////////////
  Test case 13
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Windows
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 14
```

```
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Macintosh
arglist : arglist[2]= ISDN
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 15
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= PPP
arglist : arglist[3]= Local

/////////////////////////////////////
  Test case 16
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= ISDN
arglist : arglist[3]= Local
/////////////////////////////////////
  Test case 17
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= LAN
arglist : arglist[3]= Networked

/////////////////////////////////////
  Test case 18
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= ISDN
arglist : arglist[3]= Networked

/////////////////////////////////////
  Test case 19
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= LAN
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 20
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= PPP
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 21
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Networked

/////////////////////////////////////
  Test case 22
/////////////////////////////////////
arglist : arglist[0]= Netscape
arglist : arglist[1]= Windows
arglist : arglist[2]= LAN
arglist : arglist[3]= Screen
/////////////////////////////////////
  Test case 23
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Local

/////////////////////////////////////
  Test case 24
/////////////////////////////////////
arglist : arglist[0]= IE
arglist : arglist[1]= Macintosh
arglist : arglist[2]= PPP
arglist : arglist[3]= Screen

/////////////////////////////////////
  Test case 25
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Local

/////////////////////////////////////
  Test case 26
/////////////////////////////////////
arglist : arglist[0]= Other
arglist : arglist[1]= Linux
arglist : arglist[2]= ISDN
arglist : arglist[3]= Networked
```

Figure 16 – Output Fault File for Experiment 2

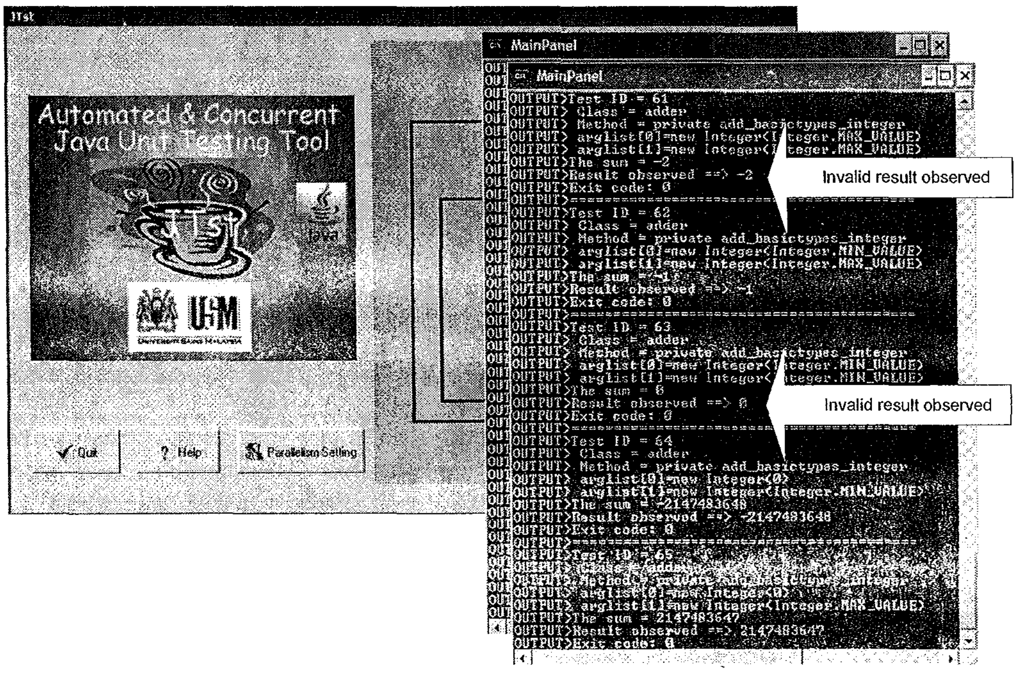
Observing the base input in Table 7 and the output fault file in Figure 16, both the expected result and actual result matches implying that the algorithm for all column sensitivity is correct.

Applicability of JTst

One experiment will be demonstrated here in order to highlight the features of JTst. In particular, the experiment will highlight the JTst automation and (concurrent) execution capabilities. The experiment involves an adder class module. The main purpose of the adder module is to perform addition of two integer numbers as well as two complex floating point numbers. The complete source code implementation for the adder module is straightforward and will not be shown here. Here, we aim to demonstrate that JTst have sufficiently rich features to perform unit test on a typical Java module.

There are three methods declared in the adder module. The method interface *add_basictypes_integer*, is declared as private, returns an integer, and takes two integers as input parameters. Unlike the method interface *add_basictypes_integer*, the method interface *add_java_Integer*, is declared as protected , returns void, and takes two Integer class as input parameters. Finally, the method interface *add_user_defined_Complex_Double*, is declared as public, returns void, and takes two user defined class called Complex. Here, the user defined Complex class takes a constructor of the real part and imaginary part of type String (and later converted to floating points) in order to initialize the Complex object.

For each of the method interfaces, 65 test cases were (combinatorially) defined. Concerning the method interface *add_basictypes_integer*, unit testing is still possible although the method is declared as private (and returns an integer). The snapshot of the output can be seen in Figure 17.



```
OUTPUT>Test ID = 61
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(Integer.MAX_VALUE)
OUTPUT> arglist[1]=new Integer(Integer.MAX_VALUE)
OUTPUT>The sum = -2
OUTPUT>Result observed ==> -2
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 62
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(Integer.MIN_VALUE)
OUTPUT> arglist[1]=new Integer(Integer.MAX_VALUE)
OUTPUT>The sum = -1
OUTPUT>Result observed ==> -1
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 63
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(Integer.MIN_VALUE)
OUTPUT> arglist[1]=new Integer(Integer.MIN_VALUE)
OUTPUT>The sum = 0
OUTPUT>Result observed ==> 0
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 64
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(0)
OUTPUT> arglist[1]=new Integer(Integer.MIN_VALUE)
OUTPUT>The sum = -2147483648
OUTPUT>Result observed ==> -2147483648
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 65
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(0)
OUTPUT> arglist[1]=new Integer(Integer.MAX_VALUE)
OUTPUT>The sum = 2147483647
OUTPUT>Result observed ==> 2147483647
OUTPUT>Exit code: 0
```

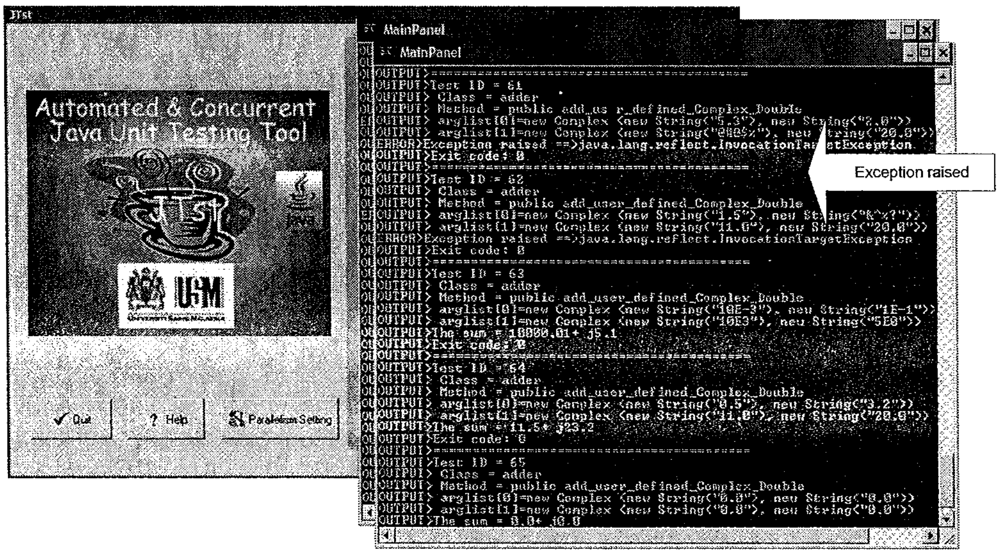
Figure 17 – JTst Unit Testing Snapshot for *add_basictypes_integer*

In testing the method interface *add_basictypes_integer*, an alarming result is observed when two maximum boundary values based on the Java pre-defined constant Integer.MAX_VALUE are added together. It is expected that the operation would throw a floating point overflow exception (i.e. integer out-of-bound). Rather, the operation gives a wrong value of -2 (see Figure 17). Similarly, adding two minimum boundary values based on Java pre-defined constant Integer.MIN_VALUE also produces erroneous answer. These results show that the adder module used in this experiment could not tolerate even the simplest robust inputs.

As far as the method interface *add_java_Integer* is concerned, it is also possible to test all the 65 test cases even though the method is declared as void protected. Similar to the method interface *add_basictypes_integer*, the method interface would fail to give correct results when the boundary values were used.

Finally, concerning the method interface *add_user_defined_Complex_Double*, 65 test cases were also (combinatorially) defined. Unlike the two method interfaces defined earlier, *add_user_defined_Complex_Double* is declared as void public. While public method is universally accessible to all, it is also necessary to demonstrate the fact that JTst can also test public methods in order to qualify as a general software testing tool.

In order to initialize the Complex object as the passing parameters of the method interface *add_user_defined_Complex_Double*, two input String classes were used for the real and imaginary part of the complex numbers of interest as part of the constructor. The reason for using string inputs is to demonstrate that SFIT can also trigger Java exception handling mechanism in the case of invalid String class inputs (e.g. "@#@$%" and "&^%?") As discussed earlier, the two String inputs will undergo floating point conversion in the Complex class before that addition takes place. This issue is further demonstrated in the snapshot given in Figure 18.



```
OUTPUT>==========================================
OUTPUT>Test ID = 61
OUTPUT> Class = adder
OUTPUT> Method = public add_us r_defined_Complex_Double
OUTPUT> arglist[0]=new Complex (new String("5.3"), new St ing("2.0"))
OUTPUT> arglist[1]=new Complex (new String("@#@$%"), new  tring("20.0"))
ERROR>Exception raised ==>java.lang.reflect.InvocationTa etException
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 62
OUTPUT> Class = adder
OUTPUT> Method = public add_user_defined_Complex_Double
OUTPUT> arglist[0]=new Complex (new String("1.5"), new St ing("&^%?"))
OUTPUT> arglist[1]=new Complex (new String("11.0"), new String("20.0"))
ERROR>Exception raised ==>java.lang.reflect.InvocationTargetException
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 63
OUTPUT> Class = adder
OUTPUT> Method = public add_user_defined_Complex_Double
OUTPUT> arglist[0]=new Complex (new String("10E-3"), new String("1E-1"))
OUTPUT> arglist[1]=new Complex (new String("10E3"), new String("5E0"))
OUTPUT>The sum = 10000.01+ j5.1
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 64
OUTPUT> Class = adder
OUTPUT> Method = public add_user_defined_Complex_Double
OUTPUT> arglist[0]=new Complex (new String("0.5"), new String("3.2"))
OUTPUT> arglist[1]=new Complex (new String("11.0"), new String("20.0"))
OUTPUT>The sum = 11.5+ j23.2
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 65
OUTPUT> Class = adder
OUTPUT> Method = public add_user_defined_Complex_Double
OUTPUT> arglist[0]=new Complex (new String("0.0"), new String("0.0"))
OUTPUT> arglist[1]=new Complex (new String("0.0"), new String("0.0"))
OUTPUT>The sum = 0.0+ j0.0
```

Figure 18 – JTst Unit Testing Snapshot for *add_user_defined_Complex_Double*

Overall, all the JTst features have been demonstrated and summarised in the experimentations given in this section. In fact, JTst has successfully facilitated the process of finding errors through unit testing. In the next section, details assessment of JTst will be made.

JTst Assessment

The main issues which are under consideration in this section relate to the applicability of JTst as a helpful unit testing tool. Such consideration may help to improve JTst by providing necessary feedback based on the author's practical experiences.

The first issue for JTst assessment relates to the fact that all JTst algorithms are correctly implemented. Experimentation with all JTst algorithm implementation in earlier section demonstrates that the expected results match with the actual results, indicating conformance.

Addressing the correctness issue is important to ensure that the final unit results are meaningful and that errors are not mistakenly coming from JTst itself rather from the module under test.

Secondly, there is also issue relating to the fact that some software implementation do not come source codes. In the absence of source codes; unit testing is still possible using JTst. Recall that, JTst provides an inspector tool for interrogating module under question for class name and method interfaces. This information would then be used in order to do the actual unit testing through the method interface.

Although the running experiment with JTst discussed earlier demonstrated testing of module with known source code, the same experiment could have been done in the absence of source code. The reason for having tested modules with source code is to enable comparison with the expected results with the actual value. In this way, the true behavior of JTst can be ascertained.

In all the experiments discussed earlier, JTst has been successful to test even on the private method interfaces. Recall that in object-oriented languages, private methods are inaccessible to any module other than itself. The fact that JTst can access the private method reflects the suitability of JTst as a robust unit testing tool. As discussed earlier, the capability of JTst to access private method is actually made possible by the use of computational reflection. For this reason, computational reflection is seen a useful technique for implementing an automated unit testing tool.

In addition to allowing the testing of private methods (and public ones), JTst also permits testing of protected methods. Conceptually, protected methods are only accessible to the child of the module under test derived from inheritance relationship. As seen earlier, JTst can still successfully test the component under test without the need to rely on inheritance relationship. With such a feature, JTst appears to be a useful testing tool for object oriented program.

Apart from injecting private and protected methods, the experiment discussed earlier also demonstrated the fact that JTst can also test user-defined classes (see Figure 18). The significant of this capability demonstrates the scalability of JTst for testing commercial software module which sometimes does not use standard Java classes.

Perhaps, the most important feature of JTst is its automation support. As a comparison, Figure 19 depicts traditional testing versus automated testing provided by JTst. As the figure illustrates, a number of saving in terms of test driver generations and feedback loops can be observed. In this respect, JTst can be compared to JUnit [11, 15]. In order to use JUnit, programmers would still be required to write test drivers. In JTst, no programming would be required as the test drivers are automatically generated by the JTst automated loader. Such a feature is helpful to relieve the test engineers from the mundane tasks inherent in the testing process.

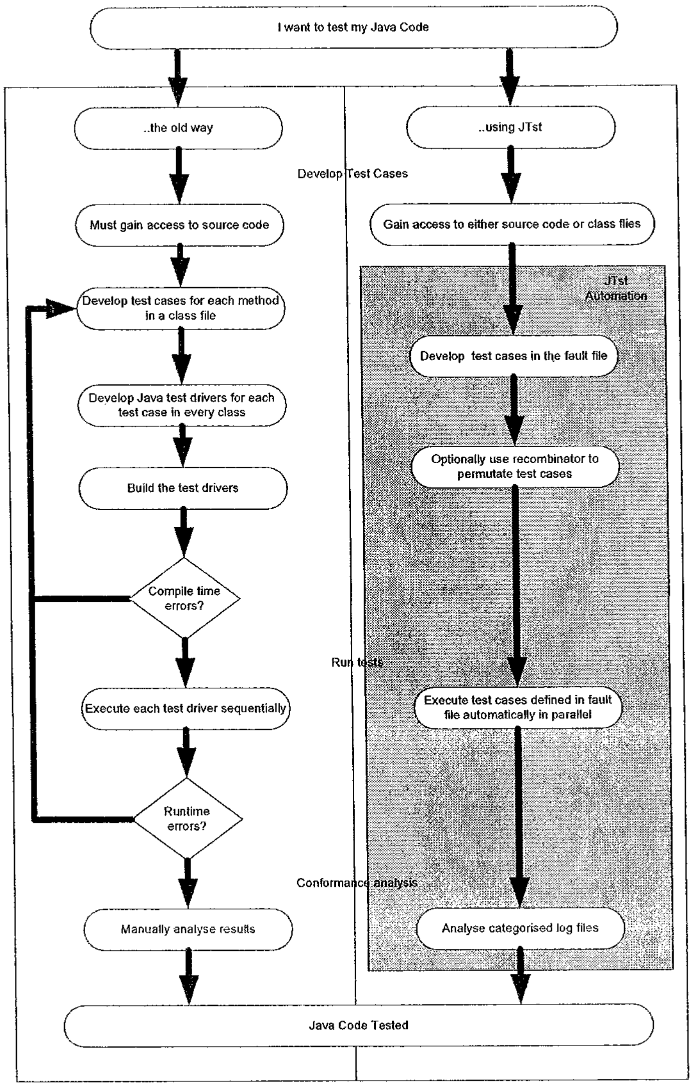

I want to test my Java Code
..the old way
..using JTst
Develop Test Cases
Must gain access to source code
Gain access to either source code or class files
JTst Automation
Develop test cases for each method in a class file
Develop test cases in the fault file
Develop Java test drivers for each test case in every class
Optionally use recombinator to permutate test cases
Build the test drivers
Compile time errors?
Run tests
Execute each test driver sequentially
Execute test cases defined in fault file automatically in parallel
Runtime errors?
Conformance analysis
Manually analyse results
Analyse categorised log files
Java Code Tested

Figure 19 – JTst versus Traditional Testing

Apart from automation support, JTst also permits (concurrent) test execution. This feature is useful in the case where there are many test cases to be executed (see Figure 17 and Figure 18 respectively). Indeed, there could be significant time saving given proper utilization of such feature.

JTst approach is similar to JACA [16] in the sense that JACA also uses computational reflection in order to inject faults in a Java program. At a glance, JACA appears to have all the features of JTst. Nevertheless, a closer look reveals that, unlike JTst, JACA requires that the test engineer who performs the testing have substantial knowledge of Java in order to undertake the injection process, that is, in order to manually write the test driver program. As seen earlier, the driver code for JTst are automatically generated and executed in a single-click of a button. Furthermore, the injection process in JTst is highly automated allowing 15,000 test cases to be executed at a particular instant. As such, JTst can be seen as offering a high level of abstraction for Java based unit testing.

Future Work

One useful avenue to improve JTst would be to integrate the sensitivity measure matrix such as the CRASH scale [6] [18]. The CRASH scale refers to the acronyms which stands for Catastrophic, Restart, Abort, Silent, and Hindering. Catastrophic failures refer to failures that can cause the whole system to stop functioning. Restart failures means that the system hangs and require user intervention to kill the appropriate tasks. Abort failures refer to abnormal termination of the tester process. Silent failures are false successes, that is, when error should have occurred. Finally, hindering failures mean return incorrect error codes. With the CRASH scale, it is possible to measure how sensitive is the components under test in response to the extreme inputs and stressful environment. Furthermore, with such integration, it may be possible to simplify and automate the process of objectively rating the software resilience against faults.

Apart from the integration with sensitivity measure matrix, it would also be useful to investigate the support for parallel execution of test cases within a large scale distributed environment such that of the GRID. With such features, more time saving can be obtain as far as testing is concerned.

Finally, another aspect which could be investigated is the application of JTst as a testing tool for real mission critical applications. Only through comprehensive and thorough evaluations can the true value of JTst be known.

Concluding Remarks

The work discussed in this thesis is significant in order to alleviate the difficulties in software testing, that is, by automating (and parallelizing) the test execution process as well as permitting extended test coverage through combinatorial generation of test cases. As human are more and more dependent toward software, this work presents a small leap toward the betterment of the techniques and processes for the ensuring quality software.

References

[1] N.J.P. Acantilado and C.P. Acantilado. "Simple: A Prototype Software Fault Injection Tool", MSc Thesis, Naval Postgraduate School, Monterey, California, December 2002.

[2] M.I. Ahmad. "A Software Fault Injection Tool", MSc ESDE Dissertation, School of Electrical and Electronics, University Science Malaysia, May 2005.

[3] M.I. Ahmad, A.R. Mohd Saad, M.N. Mat Isa, Kamal Z. Zamli, "Design of Software Fault Injection Tool Using Computational Reflection", in proceedings of the International Conference on Science and Technology: Application in Industry & Education (ICSTIE 2006) Dec 8-9, UiTM Penang.

[4] M.D. Alang Hassan. "Enhancing and Evaluating A Software Fault Injection Tool", MSc ESDE Dissertation, School of Electrical and Electronics, University Science Malaysia, December 2005.

[5] P.E. Ammann and A.J. Offutt. "Using Formal Methods to Derive Test Frames in Category-Partition Testing". In *Proc. of the 9th Annual Conference on Computer Assurance (COMPASS'94)*, IEEE CS Press, June 1994, pp. 69-80.

[6] The Ballista Project. URL http://www.ices.cmu.edu/ballista

[7] K. Beck and E. Gamma. "Test Infected: Programmers Love Writing Tests", *Java Reports* Vol 3, No 7, 1998, pp 37-50.

[8] B. Beizer. Software Testing Technique. Thomson Computer Press, 2nd Edition, 1990.

[9] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing In Practice". In *Proc. of the 21st Intl. Conf. on Software Engineering*, ACM Press, May 1999, pp. 285–294.

[10] Java Programming Language. URL http://iava.sun.com

[11] JUnit Website - URL http://www.junit.org

[12] D.R. Kuhn and M.J. Reilly. "An Investigation of the Applicability of Design of Experiments to Software Testing". In *Proc. of the 27th NASA/IEEE Software Engineering Workshop*, IEEE CS Press, December 2002, pp. 69-80.

[13] D.R. Kuhn and V. Okun. "Pseudo-Exhaustive Testing for Software". In *Proc. of the 30th NASA/IEEE Software Engineering Workshop*, IEEE CS Press, April 2006.

[14] D.R. Kuhn and D.R. Wallace. "Software Fault Interactions and Implications for Software Testing". *IEEE Transactions on Software Engineering 30(6)*, June 2004, pp. 418-421.

[15] A. Matt. "Testing Java Interface with JUnit". *Dr Dobb's Journal*, February 2003.

[16] R.L.O. Moraes, and E. Martins. "Jaca – A Software Fault Injection Tool". In *Proc. of the 2003 Intl. IEEE Conf. on Dependable Systems and Networks*, IEEE CS Press, p.667.

[17] G.J. Myers. The Art of Software Testing, John Wiley and Sons, 1976.

[18] J. Pan, P. Koopman, and D. Seiwiorek, "A Dimensionality Model Approach to Testing and Improving Software Robustness", in *Proceedings of the IEEE Systems Readiness Technology Conference*, IEEE CS Press, 1999, pp. 493-502.

[19] R. Pressman. Software Engineering: A Practitioner's Approach, 5th Edition McGraw Hill, 2000.

[20] G.J. Silva, R.J. Drebes, J. Gerchman, T.S. Weber. "FIONA: A Fault Injector for Dependability Evaluation of Java-based Network Applications". In *Proc. of the 3rd Intl. Symposium on Network Computing and Applications (NC'04)*.

[21] K.Z. Zamli, N.A. Mat Isa, O. Sidek, and M.I. Ahmad. "Ensuring Dependability of COTs: A Work in Progress". In *the CD-ROM Proc. of the Intl. Conference on Robotics, Vision, and Information Processing (ROVISP05)*, Penang, Malaysia, July 2005.

[22] K.Z. Zamli, M.D. Alang Hassan, N.A. Mat Isa, and S.N. Azizan. "SFIT – An Automated Software Fault Injection Tool". In *Proc. of the 2nd Malaysian Software Engineering Conference (MySec2006)*, Kuala Lumpur, December 2006, pp. 99-105.

[23] K. Z. Zamli, M. D. Alang Hassan, N.A. Mat Isa, S.N. Azizan. "An Automated Software Fault Injection Tool For Robustness Assessment of Java COTs". In *Proc. Of the IEEE Intl. Conference on Computing and Informatics (ICOCI06)*, IEEE CS Press, June 2006.

[24] K.Z. Zamli, N.A. Mat Isa, M.F.J. Klaib and S.N. Azizan. "Designing a Combinatorial Java Unit Testing Tool". In *Proc. of the IASTED Intl. Conference on Advances in Computer Science and Technology (ACST 2007)*, Phuket, Thailand, April 2007.

[25] Kamal Z. Zamli, Nor Ashidi Mat Isa, Mohammad Fadel Jamil Klaib, Zainal Hisham Che Soh, and Che Zalina Zulkifli, "On Combinatorial Explosion Problem for Software Configuration Testing". In *Proc. of the Intl. Conference on Robotics, Vision, Information and Signal Processing 2007* (ROVISP2007), Nov 28-30, 2007, Penang, pp. 26-30.

[26] Kamal Z. Zamli and Nor Ashidi Mat Isa, "JTst: An Automated Unit Testing Tool for Java Program", in the American Journal of Applied Science, Vol 5, No 2, pp. 77-82.

Project Members

The project members for this project are:

1. Dr Kamal Zuhairi bin Zamli (Project Leader)
2. Dr Nor Ashidi Mat Isa
3. Siti Norbaya Azizan (RO)
4. Mohamad Fadel Jamil Klaib (RO) – writing up PhD Thesis
5. Mohd Firdaus Alias (RO) – writing up MSc Thesis
6. Mohd Annuar Mat Isa (RO) – writing up MSc Thesis

Current Project Outputs – Publications & Awards

Local Conference Publications

1. M.I. Ahmad, A.R. Mohd Saad, M.N. Mat Isa, Kamal Z. Zamli, "Design of Software Fault Injection Tool Using Computational Reflection", in proceedings of the International Conference on Science and Technology: Application in Industry & Education (ICSTIE 2006) Dec 8-9, UiTM Penang.

2. Kamal Z. Zamli, Mohd Daud Alang Hassan, Nor Ashidi Mat Isa, Siti Norbaya Azizan, "Implementing an Automated Java Unit Testing Tool: Some Early Experiences", in proceedings of the 2nd Malaysian Software Engineering Conference 2006 (MySec06), pp. 99-105.

3. Kamal Z. Zamli, Mohamed Fadel Jamil Klaib, Nor Ashidi Mat Isa, "Combinatorial Explosion Problem in Software Testing – Issues and Practical Remedies", in proceedings of the 3rd Malaysian Software Engineering Conference 2007 (MySec07), pp. 24-28

International Conference Publications

4. Kamal Z. Zamli, Mohd Daud Alang Hassan, Nor Ashidi Mat Isa, Siti Norbaya Azizan, "An Automated Software Fault Injection Tool For Robustness Assessment of Java COTs", in proceedings of the IEEE International Conference on Computing and Informatics (ICOCI06), Kuala Lumpur, June 6-8, 2006.

5. Kamal Z. Zamli, Nor Ashidi Mat Isa, Mohammed Fadel Jamil Klaib, Siti Norbaya Azizan, "Designing a Combinatorial Java Unit Testing Tool", in proceedings the IASTED International Conference on Advances in Computer Science and Technology (ACST 2007), Phuket, Thailand, Apr 2-4, 2007.

6. Kamal Z. Zamli, Nor Ashidi Mat Isa, Mohammad Fadel Jamil Klaib, Zainal Hisham Che Soh, and Che Zalina Zulkifli, "On Combinatorial Explosion Problem for Software Configuration Testing", in proceedings of the International Conference on Robotics, Vision, Information and Signal Processing 2007 (ROVISP2007), Nov 28-30, 2007, Penang, pp. 26-30.

International Journal Publications

7. Kamal Z. Zamli, Nor Ashidi Mat Isa, Mohd Daud Alang Hassan, "Software Testing and Automation", in International Journal of Factory Automation, Robotics and Soft Computing, Issue 2 (April 2007), pp. 38-43.

8. Kamal Z. Zamli, Nor Ashidi Mat Isa, Mohamed Fadel Jamil Klaib, Siti Norbaya Azizan, "A Tool for Automated Test Data Generation (and Execution) Based on Combinatorial Approach", in International Journal of Software Engineering and Its Applications, July 2007, pp. 19-34.

9. Kamal Z. Zamli and Nor Ashidi Mat Isa, "JTst: An Automated Unit Testing Tool for Java Program", in the American Journal of Applied Science, Vol 5, No 2, pp. 77-82.