Unit Testing Tool competition

Sebastian Bauersfeld*, Tanja E.J. Vos*, Kiran Lakhotia†, Simon Poulding‡ and Nelly Condori*
*Universidad Politecnica de Valencia
Email: sbauersfeld, tvos, nelly @ pros.upv.es
†University College London
Email: k.lakhotia@cs.ucl.ac.uk
‡University of York
Email: simon.poulding@york.ac.uk

***Abstract*—The abstract goes here. DO NOT USE SPECIAL CHARACTERS, SYMBOLS, OR MATH IN YOUR TITLE OR ABSTRACT.**

***Keywords*-component; formatting; style; styling;**

I. Introduction

This paper describes the benchmark set-up of the Java Unit Testing Tools competition that run for SBST 2013 edition. According to [1] a benchmark should have three parts: (1) a clearly defined objective of what is and what can be evaluated (Section II-A and II-B); (2) a task sample (Section II-C and II-E); (3) performance measures (Section II-D and II-F).

II. Designing the benchmark

A. Objective - What to achieve?

The objective of the benchmark is to evaluate tools that generate JUnit test cases for Java classes. Evaluation will be done with respect to a benchmark score that takes into account test effectiveness (fault finding capability and coverage) and test efficiency (time to prepare, generate and execute). Two baselines for comparison will be used. The baselines are each on an extreme end of the level of intelligence that is used for generating the test cases. On the one hand a random testing approach implemented by the tool Randoop [2]. On the other hand, the manually created tests that come with the classes that are part of the benchmark data.

B. Uniform description of the tools being studied

In order to be able to unify and combine the results of the benchmark and aggregate the results in secondary studies [3], [4], we need to use a taxonomy (as [5] calls it) or a hierarchy (as [3] calls it) or a characterisation schema (as [6] calls it) of the tools under investigation. We decided to use the taxonomy from [5], that we have adapted to software testing and augmented with the results from [3], [7], [6].

To illustrate the resulting schema, Table I contains the description of baseline Randoop [8].

Prerequisites	
Static or dynamic	Dynamic testing at the Java class level.
Software Type	Java classes.
Lifecycle phase	Unit testing for Java programs.
Environment	In principle all development environments, special versions/plugins exist for Eclipse and the Microsoft .NET platform.
Knowledge required	JUnit unit testing for Java.
Experience required	Basic unit testing knowledge.
Input and Output of the tool	
Input	Java Classes (compiled)
Output	JUnit test cases (source)
Operation	
Interaction	Through the command line.
User guidance	Through the command line.
Source of information	Manuals and papers online [2]
Maturity	
Technology behind the tool	Feedback-Directed Random testing
Obtaining the tool and information	
License	MIT License.
Cost	Open source.
Support	Could try to contact the developers directly.
Empirical evidence about the tool	
Only studies about effectiveness have been found [9], [10], [11]	

Table I
Description of Randoop

C. Objects - the Java Classes Under Test (CUTs)

1) Selecting the CUTs: The motivation for selecting the CUTs that constitute the benchmark was to have applications that are reasonably small, but not trivial, so we can run the competition and finish it in restricted time. Therefore, we have selected classes from well-known test subjects used in the SBST community that come with a number of manually written JUnit test classes which we need for the previously stated baseline [12], [13], [14], [15]. Classes for which at least one manually written unit test existed in the project are considered interesting, because developers had made the effort to write a unit test for them. We decided not to use the

SF100 benchmark [16] because it is too large for the tools to complete in reasonable time for the competition and it contains many unconventional classes. Our final benchmark only contains some classes that are unconventional in that they contain difficult to test methods like for example a constructor that takes a file as an argument. These come from the sqlsheet [17] project.

Our competition relies on the coverage tool Cobertura [18] to collect coverage information and the mutation testing tool Javalanche [19] to compute the mutation score. Therefore, we further restricted the classes based on the limitation of each of those tools (and combined use) .

2) Characteristics of the CUTs: Table II shows the characteristics of the classes that constitute the benchmark. *AMC* denotes the Average Method Complexity per Class, i.e. the sum of cyclomatic complexities of the methods of the given class divided by the amount of methods; *LOC* denotes the Lines of Code of the CUT; *TLOC* denotes the Lines of Code of the Test Class that tests the CUT; *TAss* denotes the number of invocations of JUnit `assert<X>()` and `fail()` methods that occur in the code of the corresponding test class. These measures are given to aid benchmark participants in analyzing the strengths and weaknesses of their tools when comparing their results to the manual test cases.

3) Seeding mutants: In order to be able to evaluate the fault finding effectiveness of the generated test cases, we decided to use the mutation testing tool Javalanche [19]. The purpose of mutation testing is to insert artificial faults into a program (*i.e.*, faults a programmer might make), and assess how good test cases are at detecting those faults. In Javalanche a fault is considered detected (*i.e.*, killed) if the result of running a unit test on the original program and the mutated version differs. This is typically indicated by a passing test failing on a mutant program and is akin to strong mutation testing [24]. Thus, the ability to kill mutants generated by Javalanche depends upon how thorough a test checks the output of a particular test execution. One example in which this can be done is by using JUnit `assert` functions to check properties of the class under test, or simply check the return value of a method under test (if applicable).

In theory one can generate a very large number of mutants, because a program typically has many statements or expressions that can be changed by a mutation tool. In order to make mutation testing tractable, Javalanche only implements a subset of all possible mutation operators. These are: replacing numerical constants, negating jump conditions, replacing arithmetic operators, and omitting method calls [19].

To further optimise the mutation testing process Javalanche uses mutant schemata. Instead of generating many copies of a class, each containing a single mutation, Javalanche adds all supported mutations for a particular class in one instrumented class file. It uses guard statements to enable selective mutations and compute which mutants are detected by a test case. Further, instead of executing *all* tests for each mutant, Javalanche uses coverage information of tests to determine a subset of tests to run for a particular mutant. In order for a test to kill a mutant it must satisfy three requirements: 1) reaching the mutated statement, 2) infecting the program state after executing the mutant and

Number	Class	Project	AMC	LOC	TLOC	TAss	Reference
1	ArrayUtils	Apache Commons Lang	3.13	2046	2268	1025	[20]
2	Barcode	Barbecue	1.53	234	177	25	[21]
3	BaseDateTimeField	Joda Time	1.72	311	495	112	[22]
4	BlankModule	Barbecue	1.00	17	7	1	[21]
5	BooleanUtils	Apache Commons Lang	3.92	365	793	249	[20]
6	BuddhistChronology	Joda Time	1.63	99	301	164	[22]
7	CalendarConverter	Joda Time	2.20	54	121	28	[22]
8	CharRange	Apache Commons Lang	2.42	150	300	177	[20]
9	Chronology	Joda Time	1.00	116	230	72	[22]
10	CodabarBarcode	Barbecue	2.20	98	151	48	[21]
11	Code128Barcode	Barbecue	2.56	229	722	255	[21]
12	Code39Barcode	Barbecue	1.80	86	175	25	[21]
13	CompositeModule	Barbecue	1.42	49	59	5	[21]
14	ConverterManager	Joda Time	1.92	268	826	146	[22]
15	ConverterSet	Joda Time	5.33	178	144	19	[22]
16	DateConverter	Joda Time	1.00	17	74	17	[22]
17	DateTimeComparator	Joda Time	2.69	101	639	172	[22]
18	DateTimeFieldType	Joda Time	2.37	285	267	170	[22]
19	DateTimeFormat	Joda Time	4.02	437	824	176	[22]
20	DateTimeFormatter	Joda Time	1.88	284	545	178	[22]
21	DateTimeFormatterBuilder	Joda Time	2.89	1708	153	31	[22]
22	DateTimeUtils	Joda Time	1.86	156	322	57	[22]
23	DateTimeZone	Joda Time	2.41	518	723	207	[22]
24	Days	Joda Time	1.80	156	298	104	[22]
25	DefaultEnvironment	Barbecue	1.00	11	16	2	[21]
26	DurationField	Joda Time	1.09	34	31	5	[22]
27	DurationFieldType	Joda Time	2.29	150	140	67	[22]
28	EnvironmentFactory	Barbecue	1.50	45	48	6	[21]
29	FieldUtils	Joda Time	2.87	139	137	27	[22]
30	FixedBitSet	Apache Lucene	2.50	283	230	32	[23]
31	Fraction	Apache Commons Lang	3.73	443	1015	333	[20]
32	GJChronology	Joda Time	2.02	672	401	166	[22]
33	GraphicsOutput	Barbecue	1.83	51	73	13	[21]
34	GregorianChronology	Joda Time	1.82	121	224	135	[22]
35	HeadlessEnvironment	Barbecue	1.00	11	12	2	[21]
36	Hours	Joda Time	1.84	159	295	106	[22]
37	ISOChronology	Joda Time	1.53	110	359	174	[22]
38	ISODateTimeFormat	Joda Time	2.69	923	414	134	[22]
39	ISOPeriodFormat	Joda Time	1.83	116	132	41	[22]
40	IllegalFieldValueException	Joda Time	1.42	167	291	160	[22]
41	Int2of5Barcode	Barbecue	1.50	39	75	9	[21]
42	LenientChronology	Joda Time	1.88	84	111	19	[22]
43	LinearBarcode	Barbecue	2.33	41	161	16	[21]
44	LongConverter	Joda Time	1.00	18	73	18	[22]
45	MillisDurationField	Joda Time	1.09	81	156	48	[22]
46	Minutes	Joda Time	1.65	144	280	96	[22]
47	Module	Barbecue	2.00	64	67	15	[21]
48	ModuleFactory	Barbecue	1.75	383	33	9	[21]
49	Modulo10	Barbecue	1.75	29	30	7	[21]
50	Months	Joda Time	2.29	151	250	99	[22]
51	MutableDateTime	Joda Time	1.22	454	178	26	[22]
52	NullConverter	Joda Time	1.00	27	133	27	[22]
53	NumberUtils	Apache Commons Lang	5.00	636	1049	507	[20]
54	OffsetDateTimeField	Joda Time	1.19	90	431	119	[22]
55	PeriodFormat	Joda Time	1.50	38	82	6	[22]
56	PeriodFormatter	Joda Time	1.47	117	171	34	[22]
57	PeriodFormatterBuilder	Joda Time	3.46	1166	679	308	[22]
58	PeriodType	Joda Time	2.30	472	757	450	[22]
59	PreciseDateTimeField	Joda Time	1.42	47	541	119	[22]
60	PreciseDurationDateTimeField	Joda Time	1.50	62	541	123	[22]
61	PreciseDurationField	Joda Time	1.27	52	208	66	[22]
62	ReadableDurationConverter	Joda Time	1.25	27	77	20	[22]
63	ReadableInstantConverter	Joda Time	1.80	40	97	28	[22]
64	ReadableIntervalConverter	Joda Time	1.50	41	137	41	[22]
65	ReadablePartialConverter	Joda Time	1.39	35	101	17	[22]
66	ReadablePeriodConverter	Joda Time	1.00	22	72	20	[22]
67	ScaledDurationField	Joda Time	1.29	80	224	67	[22]
68	Seconds	Joda Time	1.65	144	272	93	[22]
69	SeparatorModule	Barbecue	1.00	19	10	2	[21]
70	Std2of5Barcode	Barbecue	1.62	58	77	9	[21]
71	StringConverter	Joda Time	5.71	133	460	168	[22]
72	UCCEAN128Barcode	Barbecue	5.11	180	54	8	[21]
73	UnsupportedDateTimeField	Joda Time	1.11	195	374	104	[22]
74	WeakIdentityMap	Apache Lucene	1.55	130	200	53	[23]
75	XlsSheetIterator	sqlsheet	8.50	235	60	20	[17]
76	XlsxSheetIterator	sqlsheet	6.20	256	55	18	[17]
77	Years	Joda Time	1.85	124	232	81	[22]

Table II
THE CUTS THAT CONSTITUTE THE BENCHMARK

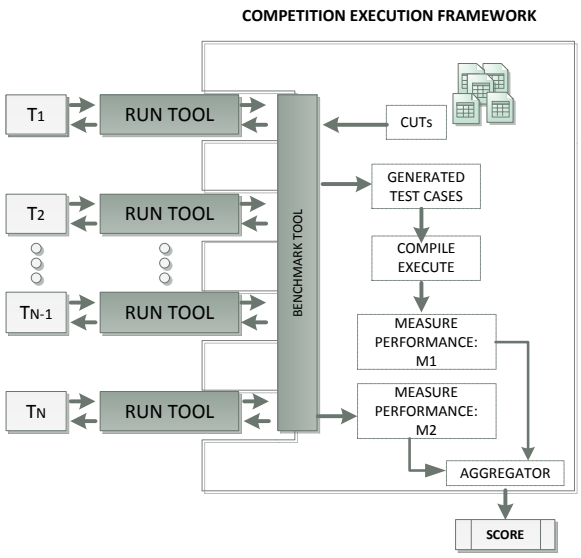

Figure 1. Competition Execution Framework

3) propagating the infected state to the output checked by the test oracle. Javalanche will only run those tests that can satisfy condition 1) for a particular mutation, since a test that does not exercise the piece of code containing the mutant cannot kill that mutant.

D. Variables - Which data to collect?

Independent variables are the testing tools $T_1 \dots T_N$ used. Other factors that can affect the results are the selected CUTs from Table II and the mutants that have been seeded by Javalanche. Dependent variables are effectiveness (coverage) and efficiency (time). The following measures are used to calculate the benchmark-score for each T_i (for $1 \leq i \leq N$):

t_{prep} preparation time that T_i needs before it starts generating test cases (i.e. instrumentation, etc.)

And for each class listed in Table II:

t_{gen} time it takes to generate the test cases
t_{exec} time it takes to execute these testcases
cov_l line coverage (measured by Cobertura [18])
cov_b branch coverage (measured by Cobertura [18])
cov_m (measured by Javalanche [19]) mutation coverage

E. Protocol - How has the benchmark been carried out?

Figure 1 shows the architecture of the framework used to carry out the competition. T_1 to T_N are the testing tools that participated in the competition. Each participant had to implement a *run tool*, which is a wrapper around the corresponding testing tool T_i and enables communication with the benchmark framework. It implements a simple protocol over the standard input and output streams, as depicted in Figure 2. The benchmark framework starts the protocol by sending the string "BENCHMARK" to the standard output stream. It proceeds by sending the location of the SUT's source code, the compiled class files and its dependencies (the Java classpath). Once the run tool received this information, it may inform the framework about its own dependencies which might be necessary to compile the generated unit test cases. It therefore can send a classpath string to the framework to be used during the compilation stage. Once it has done this, it will emit "READY" to inform the framework that it awaits the testing challenges. The Framework then starts to send the fully qualified name of the first CUT to stdout. The run tool reads this name, analyzes the class, generates a unit test and creates one or more JUnit test case files in the "temp/testcases" directory. Then, it emits "READY", after which the framework looks into "temp/testcases", compiles the file(s), executes the test cases and measures the appropriate variables. These steps are repeated until the run tool generated responses for all CUT challenges in the benchmark.

Prior to the final benchmark, we offered a set of 5 test benchmarks compiled from popular open source projects. The participants were able to use these in order to test the correct implementation of the protocol and to tune their tool's parameters. However, none of the classes of these test benchmarks were part of the final benchmark.

We carried out the benchmarks on an Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz with 8GB of main memory running Ubuntu 12.04. 1 LTS. Since most of the tools work non-deterministic and make use of random number generation, the results can slightly vary between distinct runs. Thus, it was necessary to run the benchmark multiple times, in order to obtain an average value for the achieved score. We carried out 6 benchmark runs for each tool before we averaged the achieved score over all runs. Due to time and resource restrictions we were unable to carry out more runs. However, we are confident that the obtained results are accurate enough, since for each tool the sample standard deviation and resulting confidence intervals of the scores were small. All timing information was measured in wall clock time using Java's `System.currentTimeMillis()` method. If a run tool crashed during a run or got stuck for more than one hour, we continued the run with the remaining CUTs and deducted all points for the CUT that caused the run tool to crash.

After we obtained and averaged the data, we made the results available to all participants on our benchmark website.

F. Data Analysis - How to interpret the findings?

Having measured all variables during the benchmark runs, we had to define a ranking scheme in order to determine which tool performed best. We defined the two most important indicators of a tool's quality to be the time needed to pass the benchmark and the ability of the generated tests to kill mutants. In addition, we rewarded a tool's ability to generate tests with good code coverage. To express the

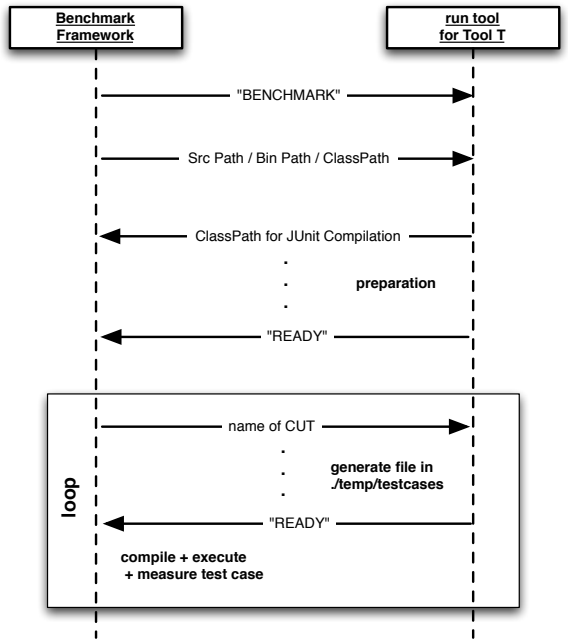

Figure 2. Benchmark Automation Protocol

quality of a tool T as a single number, we defined a benchmark function which assigns each run of a test tool T a score as a weighted sum over the measured variables:

$$score_T := \sum_{class} \Big[\omega_l \cdot cov_l(class) + \omega_b \cdot cov_b(class) + \omega_m \cdot cov_m(class) \Big] - \omega_t \cdot \left(t_{prep} + \sum_{class} \left[t_{gen}(class) + t_{exec}(class) \right] \right)$$

where, consistent with Section II-D, cov_l, cov_b, cov_m refer to achieved line, branch and mutation coverage and t_{prep}, t_{gen}, t_{exec} referring to the tool's preparation time, test case generation time and test case execution time, measured in hours. ω_l, ω_b, ω_m and ω_t are the weights, for which we chose the values $\omega_l = 1$, $\omega_b = 2$, $\omega_m = 4$ and $\omega_t = 5$. As mentioned before, we chose time and the ability to kill mutants to be the most important quality indicators, thus ω_t and ω_m have been assigned the highest values. Since it is generally more difficult to obtain a good branch coverage than a good line coverage, we chose ω_b to be two times the value of ω_l. The reason why we included line coverage, is to compensate for Cobertura's non-standard definition of branch coverage, where methods without conditional statements are considered branch-free. Therefore, in the worst case, it is possible to obtain 100% branch coverage, but at the same time achieving only extremely low line coverage.

In order to obtain a benchmark score for the manual test cases, it would be necessary to obtain the value of t_{gen} for each class. Since we do not know how much time the developers of the manual tests spent writing the corresponding JUnit classes, we cannot directly calculate a score for the manual case.

The benchmark function and the chosen weight values have been announced several days before the start of the benchmark, so that the participants were able to tune their tools' parameters.

G. Threats to Validity of the Studies Performed

Conclusion validity The next threats could affect the ability to draw the correct conclusion about relations between our treatments (testing tools) and their respective outcomes:

Reliability of treatment implementation: It means that there is a risk that application of treatments to subjects is not similar. In order to reduce this threat, a clear protocol was designed, by giving the same instructions to all participants (developers of testing tools that will be evaluated) of the unit testing tool competition.

Reliability of measures: Unreliable measures can invalidate our competition. We tried to be as objective as possible for measuring the test efficiency and effectiveness. For example, all timing information was measured in wall clock time using Javas System.currentTimeMillis () method. Effectiveness measures were automatically measured, by Cobertura [18] and Javalanche [19]. Each of these tools are widely used in the testing community but could still contain faults that might threaten the validity of the results. Furthermore, Cobertura has a non-standard definition of branch coverage, where methods without conditional statements are considered branch-free. To deal with this we also included line coverage in the benchmark score. In this way, it is possible to obtain 100% branch coverage, but at the same time achieving only extremely low line coverage. Finally, as the participants tools are based on non-deterministic algorithms, it was necessary to run the benchmark multiple times in order to obtain an average value for the measured variables. However, due to time and resource restrictions we could only run each tool a maximum of six times. This could have affected in the accuracy of the results obtained, but our benchmark was focused on obtaining an average score for each tool that participated in the competition and not on the underlying algorithms.

Internal validity The following threats could affect the interpretability of the findings.

Artefacts used in the study: This is the effect caused by the artifacts used for (experiment) competition execution, such as selected CUTS and seeded mutants to be used in a testing experiment. Our study could be affected negatively if both artefacts (CUTS and seeded mutants) were not selected appropriately. The CUTS have been taken from other experiments that have been done in the SBST community and the mutants are those seeded by Javalanch. Another artifacts used for this study are the benchmark-tool and the run tools

that were specifically developed for this competition. To ensure a good performance of the benchmark-tool, it was previously tested by the benchmark developer. With respect to the run-tools, implemented by each participant, they were also tested to tune their tools parameters, by using 5 CUTS that were not part of the final benchmark competition.

Construct validity relate to the admissibility of statements about the underlying constructs on the basis of the operationalization. The following threat is identified

Inadequate preoperational explication of constructs: it is related to the admissibility of statements about the underlying constructs on the basis of the operationalization. In our study, the final score is calculated based on a benchmark function that defines weights were assigned in accordance with the those quality indicators that are considered most important. The weights were chosen based on the experience gained through empirical studies evaluating other testing tools in industry. Most of these studies conclude that for companies the most important characteristics for a testing tool is to find errors (mutation coverage) quickly (time).

III. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank Arthur Baars. FITTEST project.

Appendix

A. Benchmark results for baselines

Tables III and IV show the benchmark results for Randoop and manual testing, averaged over 6 runs. Since we only have 6 samples, we calculated the confidence interval for the Randoop score using Student's t-distribution and Bessel's Correction to estimate the standard deviation from the sample data. For manual testing we cannot assign a concrete benchmark score, since we do not know the values for t_{gen}. Instead, we provide the score as a function of t_{gen}.

For convenience during interpretation, we listed t_{gen}, t_{exec} and t_{prep} in seconds. However, for calculation of the benchmark score, these are measured in hours!

Class	Variable (values are averaged over 6 runs, times are in seconds)				
	$t_{prep} = 0.1525$ seconds				
	t_{gen} (s)	t_{exec} (s)	cov_l	cov_b	cov_m
1	101.34	0.08	0.8538	0.6954	0.0312
2	100.35	0.00	0.0000	0.0000	0.0000
3	100.38	0.00	0.0000	0.0000	0.0000
4	102.13	0.16	0.4286	1.0000	0.4333
5	101.50	0.10	0.7211	0.5547	0.0344
6	101.20	0.00	0.0000	0.0000	0.0000
7	100.31	0.00	0.0000	0.0000	0.0000
8	100.32	0.00	0.0000	0.0000	0.0000
9	100.38	0.00	0.0000	1.0000	0.0000
10	100.58	0.08	0.5179	0.2083	0.0207
11	102.07	0.54	0.7714	0.4457	0.0077
12	101.87	0.46	1.0000	1.0000	0.0000
13	101.45	0.04	0.6800	0.6667	0.3394
14	102.52	4.12	0.5349	0.3548	0.0000
15	100.33	0.00	0.0000	0.0000	0.0000
16	100.33	0.00	0.0000	1.0000	0.0000
17	100.98	0.73	0.6786	0.5435	0.0030
18	100.30	0.00	0.0000	1.0000	0.0000
19	100.47	0.03	0.3300	0.1469	0.0449
20	100.39	0.00	0.0000	0.0000	0.0000
21	100.95	0.04	0.7981	0.6003	0.0226
22	100.35	0.01	0.2432	0.0789	0.0000
23	100.36	0.00	0.0000	0.0000	0.0000
24	100.98	0.10	0.6494	0.4103	0.0053
25	101.97	0.03	1.0000	1.0000	1.0000
26	100.31	0.00	0.0000	0.0000	0.0000
27	100.31	0.00	0.0000	1.0000	0.0000
28	100.30	0.00	0.7037	0.5000	0.0000
29	114.73	0.00	0.0000	0.0000	0.0000
30	101.23	1.53	0.9688	0.5325	0.0000
31	100.91	0.00	0.0000	0.0000	0.0000
32	101.13	0.11	0.8480	0.7273	0.0174
33	100.36	0.00	0.0000	0.0000	0.0000
34	101.14	0.07	0.7627	0.5714	0.0093
35	105.40	0.07	1.0000	1.0000	1.0000
36	101.12	0.12	0.6456	0.4000	0.0056
37	101.35	0.07	0.9111	0.7500	0.0181
38	100.40	0.07	0.7340	0.3511	0.0020
39	100.36	0.02	0.9783	0.5000	0.0000
40	104.38	0.67	0.4561	0.5000	0.4883
41	100.47	0.02	0.2667	0.5000	0.1765
42	101.19	0.00	0.0000	0.0000	0.0000
43	100.39	0.00	0.0000	0.0000	0.0000
44	100.36	0.00	0.0000	1.0000	0.0000
45	100.39	0.00	0.0000	0.0000	0.0000
46	101.33	0.13	0.6667	0.4571	0.0047
47	101.14	0.05	0.6571	0.6111	0.4340
48	100.42	0.01	0.9887	0.7000	0.0007
49	100.36	0.00	0.8000	0.6250	0.3696
50	100.98	0.08	0.6582	0.4773	0.0048
51	101.28	1.09	0.7081	0.1622	0.0000
52	100.39	0.00	0.0000	1.0000	0.0000
53	104.08	0.15	0.6756	0.4594	0.0900
54	102.31	0.00	0.0000	0.0000	0.0000
55	100.35	0.01	0.9333	0.5000	0.0000
56	100.46	0.00	0.0000	0.0000	0.0000
57	101.11	0.05	0.9180	0.6835	0.0478
58	100.95	0.03	0.8430	0.6731	0.0587
59	100.36	0.00	0.0000	0.0000	0.0000
60	100.33	0.00	0.0000	0.0000	0.0000
61	100.35	0.00	0.0000	0.0000	0.0000
62	100.38	0.00	0.0000	0.0000	0.0000
63	100.41	0.00	0.0000	0.0000	0.0000
64	100.40	0.00	0.0000	0.0000	0.0000
65	100.39	0.00	0.0000	0.0000	0.0000
66	100.40	0.00	0.0000	1.0000	0.0000
67	100.73	0.00	0.0000	0.0000	0.0000
68	100.98	0.06	0.6667	0.4571	0.0047
69	101.68	0.04	0.6250	1.0000	0.4638
70	102.00	0.34	1.0000	0.9000	0.0000
71	100.38	0.00	0.0000	0.0000	0.0000
72	101.86	0.54	0.6992	0.5789	0.0000
73	100.39	0.00	0.0000	0.0000	0.0000
74	101.28	0.06	0.9677	0.8333	0.2162
75	100.42	0.00	0.0000	0.0000	0.0000
76	100.40	0.00	0.0000	0.0000	0.0000
77	101.21	0.06	0.6393	0.4857	0.0048
Score	**101.8129** ($CI = \boxed{100.10, 103.52}$ with $\alpha = 0.05$)				

Table III
Results for Randoop

References

[1] S. Sim, S. Easterbrook, and R. Holt, "Using benchmarking to advance research: a challenge to software engineering," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, may 2003, pp. 74 – 83.

[2] "Randoop v1.3.3," http://code.google.com/p/randoop/, accessed: 22/02/2013.

[3] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 456–473, 1999.

Class	Variable (values are averaged over 6 runs, times are in seconds)				
	$t_{prep} = 0.00$ seconds				
	t_{gen} (s)	t_{exec} (s)	cov_l	cov_b	cov_m
1	unknown	0.08	0.6567	0.7830	0.0170
2	unknown	0.49	0.6991	0.6176	0.1218
3	unknown	0.08	0.7515	0.6111	0.0753
4	unknown	0.00	0.4286	1.0000	0.0500
5	unknown	0.01	1.0000	0.9609	0.0337
6	unknown	0.01	0.9259	0.7857	0.0121
7	unknown	0.06	1.0000	1.0000	0.6129
8	unknown	0.03	1.0000	0.9200	0.4346
9	unknown	0.14	1.0000	1.0000	0.0038
10	unknown	0.01	0.9643	0.9167	0.0925
11	unknown	0.12	0.9857	0.8913	0.0052
12	unknown	0.01	1.0000	1.0000	0.0238
13	unknown	0.01	0.7200	0.6667	0.1368
14	unknown	0.06	1.0000	1.0000	0.1226
15	unknown	0.01	0.7222	0.6938	0.4240
16	unknown	0.01	1.0000	1.0000	0.7059
17	unknown	0.07	0.9821	0.8696	0.0041
18	unknown	0.05	0.9867	1.0000	0.0075
19	unknown	0.10	0.7550	0.6573	0.0171
20	unknown	0.03	0.9294	0.7647	0.0205
21	unknown	0.06	0.6243	0.5825	0.0209
22	unknown	0.09	0.9459	1.0000	0.0040
23	unknown	0.14	0.8966	0.8188	0.0088
24	unknown	0.07	1.0000	0.9744	0.0045
25	unknown	0.00	1.0000	1.0000	0.1429
26	unknown	0.04	1.0000	1.0000	0.0028
27	unknown	0.04	0.9667	1.0000	0.0025
28	unknown	0.00	0.7407	0.7500	0.6364
29	unknown	0.00	0.1899	0.2069	0.2500
30	unknown	0.84	0.7604	0.3766	0.0584
31	unknown	0.11	0.9774	0.9184	0.1608
32	unknown	0.09	0.9006	0.8182	0.0205
33	unknown	0.01	0.5667	0.4000	0.1860
34	unknown	0.03	0.8475	0.6071	0.0183
35	unknown	0.00	1.0000	1.0000	0.3448
36	unknown	0.06	1.0000	0.9750	0.0040
37	unknown	0.07	0.9333	0.8125	0.0222
38	unknown	0.04	0.5455	0.3664	0.0213
39	unknown	0.01	1.0000	1.0000	0.0341
40	unknown	0.07	0.9474	0.7500	0.0035
41	unknown	0.01	0.8667	0.8333	0.0811
42	unknown	0.04	0.6863	0.3125	0.0165
43	unknown	0.02	1.0000	0.7500	0.2903
44	unknown	0.01	1.0000	1.0000	0.7222
45	unknown	0.03	1.0000	1.0000	0.0394
46	unknown	0.05	1.0000	0.9714	0.0036
47	unknown	0.01	0.7714	0.8333	0.2044
48	unknown	0.01	0.9887	0.7000	0.0006
49	unknown	0.00	0.8667	1.0000	0.8261
50	unknown	0.05	1.0000	0.9773	0.0049
51	unknown	0.06	0.2228	0.2162	0.0059
52	unknown	0.02	1.0000	1.0000	0.7727
53	unknown	0.01	0.9786	0.8712	0.1309
54	unknown	0.04	0.9487	0.7500	0.0681
55	unknown	0.02	1.0000	1.0000	0.0352
56	unknown	0.02	1.0000	0.8636	0.0335
57	unknown	0.02	0.8962	0.7848	0.0472
58	unknown	0.06	0.9522	0.8750	0.0074
59	unknown	0.04	0.9524	0.8333	0.0744
60	unknown	0.04	0.9630	0.9000	0.0771
61	unknown	0.04	1.0000	1.0000	0.0672
62	unknown	0.00	0.4167	0.0000	0.3750
63	unknown	0.01	1.0000	1.0000	0.6786
64	unknown	0.02	1.0000	1.0000	0.6970
65	unknown	0.02	1.0000	1.0000	0.6429
66	unknown	0.01	1.0000	1.0000	0.5000
67	unknown	0.04	1.0000	0.9286	0.0717
68	unknown	0.06	1.0000	0.9714	0.0042
69	unknown	0.00	1.0000	1.0000	0.1549
70	unknown	0.01	0.8889	0.8000	0.0902
71	unknown	0.04	0.9551	0.9242	0.1065
72	unknown	0.02	0.2602	0.1842	0.2231
73	unknown	0.03	0.9710	0.9167	0.0627
74	unknown	1.98	1.0000	0.8333	0.8304
75	unknown	12.33	0.5203	0.6479	0.2715
76	unknown	18.14	0.8043	0.7037	0.3214
77	unknown	0.05	1.0000	0.9714	0.0054
Score	only score function (due to unknown generation time): **Score Function:** $246.5554 - 5 \cdot \overline{t_{gen}}$ (t_{gen} in h!))				

Table IV
RESULTS FOR THE MANUAL TEST CASES

[4] B. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proc of ICSE*. IEEE, 2004, pp. 273–281.

[5] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, "Empirical studies in reverse engineering: state of the art and future trends," *Empirical Softw. Engg.*, vol. 12, no. 5, pp. 551–571, 2007.

[6] S. Vegas and V. Basili, "A characterisation schema for software testing techniques," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 437–466, 2005.

[7] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 721–734, 2002.

[8] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[9] ——, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[10] B. Daniel and M. Boshernitsan, "Predicting effectiveness of automatic testing tools," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, sept. 2008, pp. 363 –366.

[11] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .net with feedback-directed random testing," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390643

[12] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, Apr. 2010, pp. 45–54.

[13] J. Andrews, T. Menzies, and F. Li, "Genetic algorithms for randomized unit testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 1, pp. 80 –94, jan.-feb. 2011.

[14] A. Nistor, Q. Luo, M. Pradel, T. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *Software Engineering (ICSE), 2012 34th International Conference on*, june 2012, pp. 727 –737.

[15] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278 –292, march-april 2012.

[16] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 178–188. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337245

[17] "sqlsheet v6.4," https://code.google.com/p/sqlsheet, accessed: 22/02/2013.

[18] "Cobertura v1.9.4.1," cobertura.sourceforge.net, accessed: 22/02/2013.

[19] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Aug. 2009, pp. 297–298.

[20] "Apache Commons Lang v3.1," http://commons.apache.org/lang, accessed: 22/02/2013.

[21] "Barbecue v1.5 beta," http://barbecue.sourceforge.net, accessed: 22/02/2013.

[22] "Joda Time v2.0," http://joda-time.sourceforge.net, accessed: 22/02/2013.

[23] "Apache Lucene v4.1.0," http://lucene.apache.org, accessed: 22/02/2013.

[24] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *SIGSOFT FSE*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 212–222. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025144