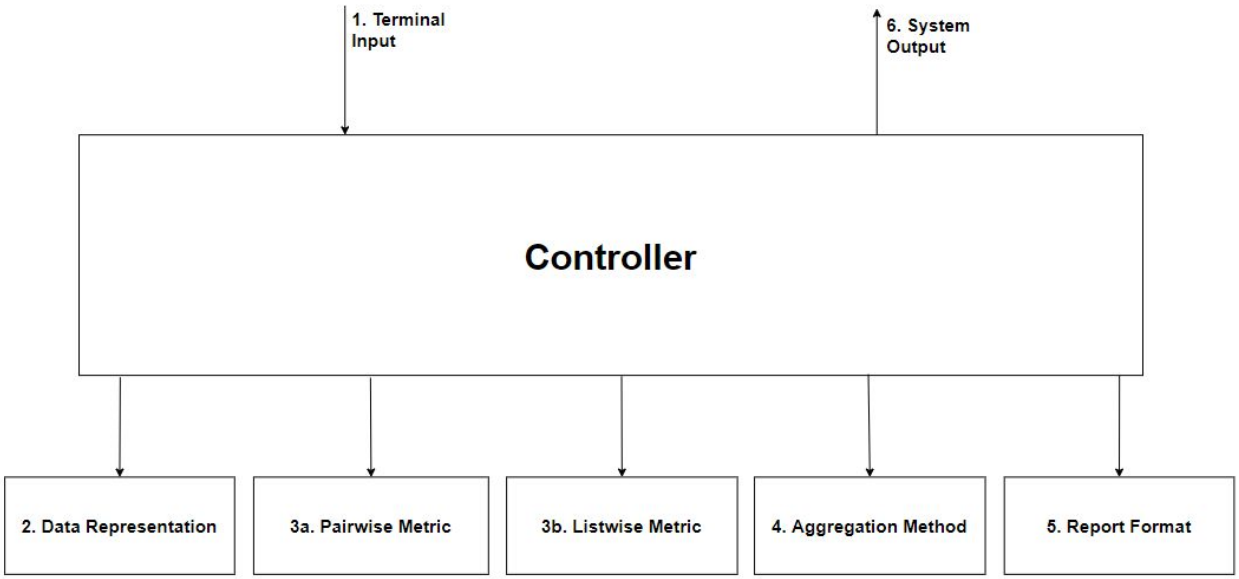
User Manual

Project Description

The system allows a user to measure the diversity within or between large test suites using a variety of diversity metrics. Results of diversity measurements can be reported in several human-readable and machine-readable formats. The system is designed as a framework to allow users to extend the system with their own diversity metrics, test case formats, and report methods.

The system functionality can be described by the following diagram:



When input is provided to the system to perform a diversity calculation, a number of activities occur. First the test case input (specified by filename) is checked against an expected data representation. This is to ensure that the test cases all follow a format that can be read by the system, and that all test cases follow the same format so they can be compared. These test cases are then compared using a pairwise/listwise metric, which produces a number of values that are combined by an aggregation method into some value that is representative of the overall test suite diversity. Finally, this result is formatted into whatever format is required with a report format. All these five options are configurable for each diversity calculation, and are all extensible for users.

Obtaining a Copy of the System

A copy of the program can be obtained from a public GitHub repository at https://github.com/LukeANewton/Tool-Framework-to-Measure-Test-Case-Diversity. The repository includes a copy of the system JAR, JCompare.jar, along with documentation, source code and project files.

System Requirements

Java 8+ is required for using the system. To run the system "as-is," JRE 8+ is required; to recompile the system from the provided source files, JDK 8+ is required.

Assumed Knowledge

It is assumed that users of the system have knowledge of how to use a terminal/command prompt to navigate their computer and run programs. For users who wish to extend the system, knowledge of Java programming is required.

Running the System

The program should be executed through the command line, with your instruction specified as command line arguments. As it is a JAR file, the system can be run using the command "java -jar JCompare.jar <system-instruction>". The system provides instructions for performing comparisons, configuring the system, and providing help to the user; these instructions are discussed in detail in subsequent sections.

The Configuration File

The system uses a JSON-formatted configuration file called "config.json". The first time the system is run in a folder which does not contain this file, the JAR will create the configuration file in the same directory the JAR resides in. This file contains several defaults options that are used by the system. These default values can be edited directly in the JSON file, or can be configured through the system itself. While system commands have flags that can be used to configure them, the configuration file parameters are system options that should be able to be changed, but do not vary as often as the values configured in flags, so they should not need to be specified in every command. The values contained in the configuration file are described in the table below.

Name of parameter in Configuration file	Description	Default value in new system-generated files
listwiseMethod		ShannonIndex
listwiseMethodLocation		metrics.comparison.listwise
pairwiseMethod	The default diversity metric to use in the system if no metric is specified.	CommonElements
pairwiseMethodLocation	The path to the folder containing comparison methods.	metrics.comparison.pairwise
dataRepresentationLocation	The path to the folder containing data representations.	data_representation
delimiter	The regular expression that matches the string separating test cases in the test suite.	\r\n
aggregationMethod	The default aggregation method to use in the system if no method is specified.	AverageValue
aggregationMethodLocation	The path to the folder containing aggregation methods.	metrics.aggregation
numThreads	The number of threads to use in the thread pool used by the system for concurrent execution.	15
resultRoundingScale	For rounding results to a specified precision. Describes the number of decimal places to round the result of a comparison.	2
resultRoundingMode	Specifies the policy for rounding. See https://docs.oracle.com/javase/7/docs/api/java/math/RoundingMode.html for valid options here.	HALF_UP

outputFileName	The default name of a file to write comparison results to when the user does not specify a filename.	comparison_result
outputFileLocation	The path to prepend to the output filename to change output location	(this is empty by default, specifying the file to be saved wherever is specified by outputFileName)
reportFormat	The default report method to use when formatting output.	RawResults
reportFormatLocation	The path to the folder containing report formats.	metrics.report_format

Comparison Instructions

Comparison commands are used to measure the diversity of test suites, the primary function of the system. The results of such a command are displayed to the terminal the system is running in, and can be optionally saved to a file. Below is the general format for comparison commands in the system. This command should be entered as command line arguments when running the program.

compare <test suite> [<test suite>] <data representation> [<flags>]

The only required parts of a comparison command are the keyword “compare,” the name of the test suite, and the data representation that matches the format of the test cases.

The test suite name specified can match a file or a folder name. For a folder, the folder’s contents are recursively searched to obtain all test cases in files and folders within; all contents of the folder are parsed as test cases, so the folder should only contain test cases.

The user has the option to specify the path of another test suite in the comparison. Specification of a single test suite calculates the diversity within the test suite, while specifying two test suites will calculate the diversity between the two test suites.

Comparison commands are configured by any number of flags following the test suite name(s) and data representation. These flags are listed below:

-m <diversity metric>: Specify the diversity metric that will be used in the calculation.

-a <aggregation methods>: Specify one or more aggregation methods with this flag followed by a space separated list of aggregation methods available to the system.

-r <report format>:Specify one or more ways to format results when outputting to the terminal and file. Each report format should be separated by a space.

-d <delimiter>: Specify the character(s) that separate test cases in the test suite file(s). The delimiter is treated as a regular expression.

-t [<number of threads>]: The use of this flag specifies that the system should use a thread pool in comparisons to improve performance. A number of threads to use can be optionally specified, or a default value from the configuration file can be used.

-s [<output filename>]: The use of this flag specifies that the results of the command should be saved to a file. A specific filename/path can be specified, and a default filename in the configuration file will be used if the filename is left unspecified.

Configuration Instructions

Configuration commands are used to edit values in the configuration file. As previously stated, the configuration file can be directly edited through other means, but this command allows the user to edit values in a safer way. The format for the configuration commands follow the format below:

config <parameter name> <parameter value>

This command verifies that the specified parameter name is a valid choice for the configuration file, and that the value provided is of a valid type for the associated parameter (eg. For parameters that should be set to numbers, this command will not allow the value to be set to non-numbers). The configuration command however does not verify that the given value itself is a valid choice (eg. When specifying a default diversity metric, the system does not check that the new string given does correspond to an actual diversity metric that has been implemented.

Help Instructions

Help commands are used to provide information about the system. The format for using this command is:

help [<help-type>]

Simply typing the word “help” provides a list of the available commands in the system, along with the syntax for those commands in the same form specified in this document, and each optional flag that can be specified on those commands. The help command can also be used to get information about the various diversity metrics, aggregation methods, data representations, and report formats supported by the system. These are the optional help types in the command format and are listed below:

-m: list the available diversity metrics in the system

-a: list the available aggregation methods in the system

-f list the available data representations in the system

-r: list the available report formats in the system

Instruction Examples

Instruction	**Description**
help	Displays the commands available to the system
help -m	Displays the available pairwise and listwise diversity metrics in the system
config comparisonMethod Levenshtein	Set the default comparison metric for the system to use to the ‘Levenshtein’ metric
config numThreads 10	Set the default number of threads for the system to use to ten
compare test CSV	Perform a diversity calculation on a file/folder of files called ‘test’, where each test case follows the format outlined by the ‘CSV’ data representation. Default diversity metrics, aggregation methods, and report formats will be used. Each calculation will be performed sequentially (ie. no concurrency through threading)
compare test EventSequence -m Hamming -a AverageValue -t 5 -s	Perform a diversity calculation on a file/folder of files called ‘test’, where each test case follows the format outlined by the ‘EventSequence’ data representation. The diversity calculation will be performed through a hamming distance, and all results will be averaged. The system will use five threads in execution and will save the calculation results to a file with the default name specified in the configuration file. The default report format will be used
compare suite CSV -s out -t -r XML -m Levenshtein -a MaxValue	Perform a diversity calculation on a file/folder of files called ‘suite’, where each test case follows the format outlined by the ‘CSV’ data representation. The results will be calculated with the ‘Levenshtein’ metric, will be aggregated as the maximum result from the results obtained, and be saved to a file called ‘out’ in XML format. The calculations will be multithreaded using the default number of threads.
Compare suite1 suite2 CSV -r JSON -a AverageValue MedianValue -s out	Perform a diversity calculation between 2 test suites named ‘suite1’ and ‘suite2’. All test cases in both suites are formatted according to the CSV data representation. The results will be saved to a file called ‘out’ in JSON format. The results reported will be the average and median of all the pairwise comparisons made.

Available Diversity Metrics in Base Version

Diversity metrics are separated into two different types: pairwise and listwise. Pairwise metrics are diversity metrics that individually compare every test case to each other test case, while listwise metrics compare the whole set of test cases at once. The key similarity between all diversity metrics is that they compare the basic elements that make up a test case. What those elements actually are is defined in the data representation.

Pairwise Metrics

CommonElements is a simple diversity metric that counts the number of elements in a pair of test cases that are in the same position and are equal. Consider two sequences of values: 1,2,3,4,5,6 and 6,2,3,4. For this pair of test cases, the CommonElements value is 3, corresponding to the elements 2, 3, and 4. For this metric, a larger number indicates higher similarity (and therefore lower diversity).

CompressionDistance is a pairwise metric based on the idea of file compression. The idea behind this metric is that files are compressed more when they contain more redundancies, so this can be leveraged to see how similar test cases are. The formula for calculating the CompressionDistance between test cases *x* and *y* is:

$$CD(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$ [2]

where *C(x)* is the length of the string *x* after compression, and *xy* denotes the concatenation of strings *x* and *y*. This metric yields a value between zero and one, where values closer to one are more diverse. It should be noted that this metric can be slow, since it involves writing and compressing files.

Dice is a metric that compares the sets of elements from each test case. The equation for dice measure is:

$$\text{Dice(A, B)} = \frac{|A \cap B|}{|A \cap B| - \frac{|A \cup B| - |A \cap B|}{2}}$$ [3]

where A and B are both sets. This metric produces a value between zero and one, where values closer to zero are more diverse.

Hamming distance is, in a way, the inverse of CommonElements. This metric counts the number of elements in each pair of test cases that are not equal [4]. For bit strings, this is equivalent to performing an XOR operation on the elements. The larger the value produced by this metric, the more diverse the two test cases

JaccardIndex is another metric that evaluates diversity based on the set of elements in a pair of test cases. The Jaccard index (also known as the Jaccard similarity coefficient) is the size of the intersection of the two sets, divided by the size of the union of the two sets [1]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This metric calculates a value between zero and one, where values closer to zero indicates the pair is more diverse.

Levenshtein distance is also known as the edit distance. This is a metric used to compare two test cases, where the value calculated is the minimum number of element insertions, deletions, or edits to transform one test case into the other [1]. The larger the value calculated is, the more diverse the test case pair.

LongestCommonSubstring reports a similarity based on the length of the largest continuous sequence of elements that appears in both test cases [1].

Listwise Metrics

Nei implements a diversity metric from life sciences called Nei's Measure. Nei's measure looks at each index of the set of test cases and calculates the simpson diversity across that position in all test cases. The final result is the average of this calculation across all indices. This is given by:

$$H(P) = \frac{1}{T} \sum_{j=1}^{T} (1 - \sum_{i=1}^{A_j} q_{ij}^2)$$

where, P is the set of test cases, T is the number of elements in a test cases, A_j is the number of possible different elements at index j, and q_{ij} is the frequency of i^{th} element at location j. [3]

ShannonIndex is a measure of the relative frequencies of elements appearing in the test suite. The larger the number, the more evenly distributed the frequencies of elements in the test cases are. It is an entropy equation, given by:

$$Sh(P) = -\sum_{r=1}^{S} q_r ln(q_r)$$

where P is the set of test cases, q_r is the frequency of element r, and S is the total number of different elements in all test cases. [3]

SimpsonDiversity is a measure of the diversity of a test suite on a scale of 0 to 1. the higher the number, the more evenly distributed the frequencies of elements in the test cases are. This is given by:

$$Si(P) = 1 - \sum_{r=1}^{S} q_r^2$$

where P is the set of test cases, q_r is the frequency of element r, and S is the total number of different elements in all test cases.[3]

This number can be expressed as a probability. If we consider all the different elements in all the test cases and select two at random, this is the probability that the elements are not equal.

StoddardIndex measures the relative frequencies of elements in the test cases and reports a value >= 1. A higher value indicates more diverse test cases. The equation is given by:

$$St(P) = (\sum_{r=1}^{S} q_r^2)^{-1}$$

where P is the set of test cases, q_r is the frequency of element r, and S is the total number of different elements in all test cases. [3]

Available Aggregation Methods in Base Version

Aggregation methods are used to pool together several diversity metric results into a single representative value. One or more aggregation methods can be specified for a single command.

AverageDissimilarity is another diversity metric from life sciences, much like the available listwise metrics are. This method takes a set of test cases as input, and the value obtained is equal to the sum of all pairwise similarity measures, divided by the size of the set squared [3]. This is given by:

$$AD = \frac{1}{n^2} \sum_{i,j=1}^{n} \delta(x_i, x_j)$$

where, P is the set of sequences, n is the size of P, and $\delta(x_i, x_j)$ is the pairwise similarity between test cases x_i and x_j.

AverageValue is an aggregation method for providing the mean of the diversity metric results. This is equal to the sum of all diversity metric results, divided by the total number of results.

Euclidean yields a value calculated by using the test case diversity results in a euclidean length calculation, equal to:

$$\sqrt{\sum x_i^2}$$

Manhattan gives a summation of the absolute values of each test case diversity result.

MaximumValue yields only the largest value from a set of test case diversity results.

MedianValue yields the median from a set of test case diversity results. This is the middle value when all diversity results are ordered from smallest to largest. If the set of diversity results has an even number of values, there will be two median values reported.

MinimumValue yields only the smallest value from a set of test case diversity results.

ModeValue yields the mode from a set of test case diversity results. This is the most frequently occuring value(s) in the set of results.

SquaredSummation squares each test case diversity result and sums all the squared values together.

Summation simply sums together every test case diversity result.

Available Data Representations in Base Version

Data representations specify the format that the test cases in a test suite file follow. A data representation describes how test cases are expected to look, and what parts of those test cases are considered the basic elements that are used in diversity metric calculations.

CSV is a data representation for comma separated values. Test cases that use this format can contain any non-newline characters where elements of the test case are separated by commas. The elements from this data representation used in diversity calculations are each element between the commas.

EventSequence, StateSequence, and **EventStatePairs** are three data representations that all follow the same format. These data representations should be used for state machine test cases that follow a format of <state>-<event>-<state>-<event>-<state>... That is, test cases of this format are a sequence of states and events, separated by dashes. The sequence must always begin with “Start” and end in another state. Test cases of this format can optionally include an id surrounded by square brackets preceding the test case. For **EventSequence**, only the events are considered when performing a diversity calculation. For **StateSequence**, only the states are considered when performing a diversity calculation. For **EventStatePairs**, the elements considered in diversity calculations are an event followed by the state the event results in; in a test case of this format, this would be an event in the list and the proceeding state.

Available Report Methods in Base Version

Report methods are used to format the results of the diversity calculation and aggregation into a format desired by the user. One or more report formats can be specified for a single command. If the user specifies that the result should be saved to a file, but multiple report formats are specified, a file will be created for each separate format, named as the user specified filename with the report format name appended to it.

RawResults is the simplest report method. This displays just the results of the diversity calculation aggregations.

Pretty provides human readable output text with a formatted header including timestamp, and provides all the parameters used in the diversity calculation, including those specified in the command line arguments and the defaults used from the configuration file. The results of the diversity aggregations are displayed along with the corresponding name of the aggregation method used.

JSON and **XML** both provide the same information as the previous report format, but in machine readable formats.

Extending the System

While the system provides several diversity metrics, aggregation methods, data representations, and report formats, it is possible to add any number of these for additional functionality. Adding to the system requires writing a Java class that implements a certain interface, but the system is designed in such a way that no changes need to be made to the existing system code to add new functionality. This section goes into more detail about exactly how new diversity metric, aggregation methods, data representations, and reporting methods can be added to the system.

Making New Files Accessible to the System

The system comes as a JAR file, so there are a number of ways to add new functionalities.

An easy way to do this is to use the JAR tool that comes with JDK. The command to do this is “jar uvf JCompare.jar <new functionality path>” where the path specified should mirror the folder structure in the JAR file. This is expanded on in each following section.

Since the program is a JAR file, which is built on the ZIP format, it is also possible alter the contents of the JAR through the use of a 3rd party file archiver utility (eg. WinRAR) to add files in the appropriate locations. The JAR file itself can also be unzipped, altered, and re-compressed to add files.

Another option is to rebuild the jar itself with the new files added. The source code and class files are made freely available and could be easily repackaged into a new JAR file through the JAR tool or through an IDE.

Adding a new Data Representation

A new data representation is likely the most frequently required addition to be made to the system, since they are specific to how the test case is written in its file.

A data representation does 3 things for the system:

1. Describes how to read the test case into the system.
2. Defines a private data structure for the test case to be stored in.
3. Provides a means of extracting elements out of the data structure for use in diversity calculations.

Giving the data representation these responsibilities gives the user a lot of freedom in how the code is written, the only requirement is that the new data representation must implement an interface called DataRepresentation that specifies five functions: *parse(), getDescription(), next(), hasNext(),* and *toString()*. Their function headers are listed below.

void parse(String s) throws InvalidFormatException
String getDescription()
Object next()
boolean hasNext()
String toString()

parse() is the means by which test cases are read into the system. This function defines a parser for whatever format test cases are structured in, and operates on a String that is the literal contents of the test case in the specified test case file. In the event the provided test case *s* does not match the format expected by the defined parser, an new InvalidFormatException should be thrown, though this is not strictly required (the system will just fail in a less graceful way if the test case cannot be parsed). This function should store that test case *s* in a private member to the DataRepresentation which can be anything, but an iterator-like *next()* and *hasNext()* must be written for the internal representation.

getDescription() is a simple method that returns a String containing a short description of the new DataRepresentation. This method is optional, omission of this function will result only in “no description available” being displayed in system “help -f” commands.

next() and *hasNext()* follow the convention of the iterator functions by the same name. The *parse()* function should read the test case into a private field, and *next()* should be used to get elements out of that private field one at a time. *hasNext()* should return a boolean which denotes true if there is another element to get from the private member, or false if all the

elements have been extracted. Like *parse()*, these two functions are critical and must be implemented as described here.

Finally a *toString()* function should be implemented. This function is a typical toString override of an object, and is useful in displaying error messages if something should go wrong. It would be best if the function could create a string that would uniquely identify the test case held within the DataRepresentation, but this is not required.

In the system, DataRepresentations are all in a package called “data_representation”. The newly created DataRepresentation must be added to this package by any method described in the section “Making New Files Accessible to the System”. The command to use the JAR tool for this is “jar uvf <JAR name> data_representation/<data representation name>”.

Adding a new Pairwise Metric

Pairwise comparison metrics must be written to implement an interface called PairwiseComparisonStrategy. This interface has two methods to implement: *compare()* and *getDescription()*. Their function headers are listed below.

double compare(DataRepresentation testCase1, DataRepresentation testCase2)
String getDescription()

The *compare()* function is the method that performs the pairwise comparison between two test cases. These test cases are the DataRepresentation objects *testCase1* and *testCase2*. To extract elements from a DataRepresentation to perform the comparison, the object should be treated as an iterator with *next()* and *hasNext()* methods. The *compare()* function should return the result of the test case comparison as a double.

getDescription() is a simple method that returns a String containing a short description of the new metric. This method is optional, omission of this function will result only in “no description available” being displayed in system “help -m” commands.

In the system, pairwise metrics are all in a package called “metrics/comparison/pairwise”. The newly created PairwiseComparisonStrategy must be added to this package by any method described in the section “Making New Files Accessible to the System”. The command to use the JAR tool for this is “jar uvf <JAR name> metrics/comparison/pairwise/<metric name>”.

Adding a new Listwise Metric

New listwise comparison metrics must be written to implement an interface called ListwiseComparisonStrategy. This interface has two methods to implement: *compare()* and *getDescription()*. Their function headers are listed below.

double compare(List<DataRepresentation> testsuite)

String getDescription()

The *compare()* function is the method that performs the pairwise comparison between two test cases. These test cases are the DataRepresentation objects in the list *testsuite*. To extract elements from a DataRepresentation to perform the comparison, the objects should be treated as an iterator with *next()* and *hasNext()* methods. The *compare()* function should return the result of the test case comparison as a double.

getDescription() is a simple method that returns a String containing a short description of the new metric. This method is optional, omission of this function will result only in “no description available” being displayed in system “help -m” commands.

In the system, listwise metrics are all in a package called “metrics/comparison/listwise”. The newly created ListwiseComparisonStrategy must be added to this package by any method described in the section “Making New Files Accessible to the System”. The command to use the JAR tool for this is “jar uvf <JAR name> metrics/comparison/listwise/<metric name>”.

Adding a new Aggregation Method

Aggregation methods must be written to implement an interface called AggregationStrategy. This interface has two methods to implement: *aggregate()* and *getDescription()*. Their function headers are listed below.

String aggregate(List<Double> similarities)
String getDescription()

The *aggregate()* function is the method that performs the aggregation of test case diversity results. These results are listed in the list of doubles called *similarities*. These results can be combined in any way in this function, and the resulting combination should be returned as a String.

getDescription() is a simple method that returns a String containing a short description of the new method. This method is optional, omission of this function will result only in “no description available” being displayed in system “help -a” commands.

In the system, aggregation methods are all in a package called “metrics/aggregation”. The newly created AggregationStrategy must be added to this package by any method described in the section “Making New Files Accessible to the System”. The command to use the JAR tool for this is “jar uvf <JAR name> metrics/aggregation/<method name>”.

Adding a new Report Format

Report formats must be written to implement an interface called ReportFormat. This interface has two methods to implement: *format()* and *getDescription()*. Their function headers are listed below.

String format(CompareDTO dto, List<Double> similarities, List<String> aggregations)
String getDescription()

The *format()* function is the method that performs the formatting of test case diversity results. The aggregation results are listed in the list of strings called *aggregations* and the pre-aggregated diversity results are listed in the list of doubles called *similarities*. The CompareDTO object, *dto*, contains all the parameters used in the diversity calculation, both from the specified command line arguments and defaults used from the configuration file. There is no requirement for what the result report string should contain, any number of these parameters can be used to create the desired format.

The ReportFormat interface also provides a couple of methods that can aid in the creation of a new report format. *getRunParameters()* provides a map of parameter names to values, as an alternative to using *dto* to get the parameter values used. *getAggregations()* provides a map of the aggregation methods used to their resulting values.

getDescription() is a simple method that returns a String containing a short description of the new method. This method is optional, omission of this function will result only in “no description available” being displayed in system “help -r” commands.

In the system, report formats are all in a package called “metrics/report_format”. The newly created ReportFormatmust be added to this package by any method described in the section “Making New Files Accessible to the System”. The command to use the JAR tool for this is “jar uvf <JAR name> metrics/report_format/<method name>”.