Tool Framework to Measure Test Case Diversity: Final Report

Project Group 19

Project Supervisor:

Yvan Labiche

Project Group Members:

Eric Bedard	101009607
Luke Newton	100999309
Cameron Rushton	101002958

Table of Contents

Introduction

All software needs to be tested, and when the test complexity is high and many test cases are written, a few test cases are likely to be redundant. To reduce test case redundancy and a test suite's complexity, test cases can be compared to find similar cases using a variety of comparison algorithms. This will eliminate redundant test cases and result in a more concise test suite.

Likewise, when a system is large, it may be infeasible to execute all test cases every time changes/additions are made. In this scenario, it would only be feasible to execute a subset of test cases. Ideally, this subset of test cases should be as diverse as possible to increase the likelihood of detecting faults, so comparison algorithms would be necessary to quantify this diversity.

For a state machine, a test case may be a sequence of states, transitions, or pairs of states and transitions. A suite of maximally diverse tests will exercise every sequence once, and not exercise previously tested sequences of states or transitions. For example, consider a software with functionalities {A, B, C}. One test case may test functionality A, while another case tests using functionalities B and C - the two cases are dissimilar.

Different comparison algorithms can be selected to compare two sequences and these algorithms may depend on the type of data being compared. Consider the simple state machine below (figure 1):

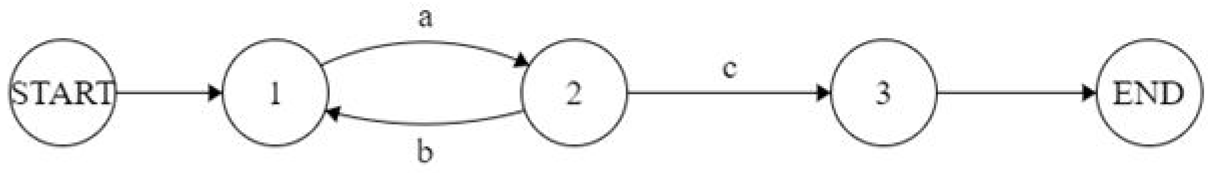


Figure 1: A simple state machine.

If test cases are defined as sequences of states, two test cases can be defined as the strings '123' and '12123'. A comparison metric which compares the sets of states in each test case would indicate these tests to be identical, whereas a Levenshtein distance [1] algorithm would notice the intermediate steps taken to exercise a loop condition and mark the test inputs more dissimilar.

The input data can also be modified to produce different results. Inputs can be grouped together to form new inputs, or each input can be a complex object of attributes. If the previous example is reused, but defined in terms of sequences of transitions instead, this yields the strings 'ac' and 'abac.' Before, the set of states between the two test cases were identical, but now the set of transitions for each test case are more dissimilar.

This shows that both the comparison algorithm and data representation can impact the perceived diversity. When comparing test cases there must be a way to measure diversity using many different metrics and data representations, in order to see a test suite from multiple views.

Ideally, there would be one clear diversity metric and data representation that would best represent that actual diversity of a test suite, but at this time this is not the case. It is not clear which diversity metrics and data representations offer the best perception of diversity, so it would be beneficial to have a tool to automate several of these diversity calculations as a way to help determine how well different metrics are at finding this diversity.

Problem Statement

The aim of this project is to create a framework to automate the diversity evaluation of any given test suite. It allows the user to choose the way in which it will measure the diversity. Since new methods of calculation and different forms of test cases will need to be evaluated in the future, the product of this project will be modular and facilitate the addition of new diversity metrics. The project will execute a full software lifecycle over the course of the academic year.

Relevance to Team Members' Degrees

The project relates strongly to Software Engineering, the degree of all three team members, in many ways. In addition to the project being the development of a software system, the knowledge learned in several courses directly applies to the project. A full project lifecycle was executed over the course of the project, meaning that in addition to the implementation, thorough requirements, design, and verification phases are performed.

In Project Requirements Engineering (SYSC3120), group members learned how to find the requirements for a project and verify that they are all valid. This knowledge was used to create a requirements document for the project to ensure the project was well understood and to keep group members on track during the remainder of the project. The Software Design Project (SYSC3110) and Software Architecture and Design (SYSC4120) courses gave an in depth teaching of design patterns, which were heavily used in the system design to ensure a good design and the non-functional requirements were met. For the implementation of the design itself, principles learned in Object Oriented Software Development (SYSC2004) and Algorithms and Data Structures (SYSC2100) were used to write well-written and efficient software. Software Validation (SYSC4101) was very relevant for the verification phase. The software testing techniques from the course were used to generate and evaluate the test cases written, and provided a means to structure testing in an efficient way. Software Project Management (SYSC4106) taught many useful techniques for managing a long term software project, and allowed the group to make an informed decision on the software lifecycle to undertake. Finally, Communication Skills for Engineering (CCDP 2100) has helped to create informative documents over the course of the project

Project Management

- Incremental life cycle
- Two increments, separated into MVP and finished project

Software Tools Used

- Java
- JUnit
- Maven
- IntelliJ
- Gson
- Github
- For each, explain what it is used for and why it was chosen

Project Scheduling

- Updated gantt charts
- Talk about whether we were on schedule or not

Project Requirements

Early on in the project, requirements for the system were determined. This was done to ensure that each team member had the same idea of what the final system would be, and to ensure that the team members' understanding of the project matched the project supervisors expectations for the project. These requirements also kept team members on track during design and implementation, ensuring that any code being written could be traced back to a requirement. If a team member wanted to implement something not defined in the requirements, they would need to discuss with the other team members before proceeding. Each requirement defined is listed below, along with any further specification or changes that were made to the requirement when a better understanding of the system had been developed.

Functional Requirements

Six main functional requirements were defined for the system:

1. The system shall be able to compare two or more test cases to each other using a pairwise diversity metric. This is the main purpose of the system. Later into the project, This was expanded to included not just pairwise metrics, but also non-pairwise (listwise) metrics as well.

2. The system shall be able to combine the diversity measurements of a set of test cases according to some aggregation method. This allows the system to not only automate large numbers of test case comparisons, but also to get some immediate meaning out of all those comparison results. Aggregation methods can, among other things, be used to collect statistics about the diversity of a test set.

3. The system shall be able to swap diversity metrics for comparing test cases. This is important because the system should not only automate comparisons for a single diversity metric, but should be able to be reused for any diversity metric that is defined.

4. The system shall read test cases from files specified by the user. Test cases may each be in separate files, or all test cases for a test suite be in a single file. Later on this requirement was relaxed so that test cases can be in separate files, but each file can contain any number of test cases.

5. The system shall give the user the choice to save the diversity measurements in a file, or simply display results in the user interface.

6. The system shall report the test case diversity measurements according to one of possibly many defined report formats. Combining this requirement with the previous allows for the system to be used as one stage in a larger process. The results of a diversity measurement can be stored in a file, in some machine readable report format, that could then be fed into some other tool.

Look and Feel Requirements

These requirements describe how the user interface for the system should look. Three requirements were defined here:

1. The system shall be run using a terminal/command prompt.
2. The user interface shall be text based within the terminal/command prompt the system was run in.
3. The user interface shall allow users to specify which test case input files to compare, which diversity metrics to use, and how to record the results in a single command. In the end, these are all specified as command line arguments when the system is run through a terminal/command prompt.

Usability Requirements

These requirements describe the user's ability to use the system and make sense of the output the system displays. Eight requirements were made for this:

1. The system shall be able to be operated by anyone with a basic knowledge of the command line. This means that if a user is familiar with using a terminal, use of the system should be straightforward. No niche terminal or operating system knowledge should be required to execute the system and perform system operations.
2. The user interface shall use plain english language and flags for commands. This means that keywords in commands should be an intuitive description of what that command does, like "compare" or "help," and command options are specified as linux-style flags with a dash followed by some string of characters.
3. The system shall display the progress of large operations as a percentage or loading bar. This is to assure the user that the operation is moving forward as normal, and to give some notion of an expected remaining time in the calculation. This is implemented in the final system as a progress bar.
4. The system will use a default diversity metric when no specific metric is chosen. This default metric can be configured, so that if the user wants to perform several operations with the same metric, they do not need to explicitly type the metric name out every time.
5. The system shall by default only display results in the user interface if the command contains no specification on how to record results. This is to avoid any unexpected files appearing on the user's computer.

6. The system shall provide a description of how a supported diversity metric works on request.

7. The system shall provide a clear user manual that is accessible in the program on command. This requirement, along with the previous, are implemented as a help menu in the system. There is also a separate user manual that goes into more detail on how the system can be used.

8. The system shall confirm with the user the diversity metric and output format for commands comparing more than two test cases. This requirement was eventually scrapped. The idea here was to avoid the system performing a large calculation if the user made an error writing their command, but a confirmation would impact workflow too much if the user wants to perform multiple operations, and programs running in a terminal can be cancel very simply through standard means (ie. CTRL+C) if the command was incorrectly written.

Operational and Environment Requirements

These requirements specify how the system should run, and under what conditions. There are four requirements here:

1. The system will be able to run on Mac, Windows or Linux.

2. The system will be designed to run on desktops/laptops.

3. The system shall be run as an executable file.

4. The system executable and source code should be easily installed with a copy of the program downloaded through a file sharing service or transferred through a shared physical storage device. This means that there should be no installer or lengthy manual install process to get the system operational on a supported platform.

Performance Requirements

These requirements are concerned with how the system should operate, with requirements related to data integrity, precision, fault tolerance, and execution capacity. In total, ten requirements are specified here:

1. The system shall not delete files on the platform when saving results to a file.
2. The system shall ask for user confirmation before overwriting a file when saving results to a file. This also helps to maintain the previous requirement.
3. The system shall not alter the test case input files used in comparison. This, along with the first requirement, allows the same test suite to be used in multiple operations, and disallows the system from interfering with the user's files
4. The system shall return a digit to user specified length, and if left unspecified will default to three decimal places. This requirement was altered in the final system to allow the default precision to be configured. The default configuration is still three decimal places.
5. The system shall not compare two test case representations if they are in different formats. This is to avoid inaccurate comparisons. Two test cases that are in different formats may be made up of different types of elements that should not be compared.
6. The system shall report parsing errors and diversity calculation errors found when using the input data, and create a log file to capture the error information.
7. The system shall be able to compare any type of object given as input. The only requirement is that a test case be supplied as a file; the user can add a parser for any type of test case representation.
8. The system shall not have a built in mechanism to cancel a command gracefully once the command has been issued. This would require extra work on the system's part to listen for such a command, but the system is run through terminal so it can be cancelled anytime with a CTRL/COMMAND + C.
9. The design should allow for the possibility of concurrent execution of test case comparisons when a command would perform more than a single pairwise comparison. In the final system, the amount of concurrency used by the system (ie. the number of threads) is configurable.
10. The results of any currently executing command must be reported before the next command can be issued to the system. This does not however, prevent multiple instances of the system running at once.

Maintainability Requirements

These requirements describe how easy it should be to add/modify the systems functionalities in the future. This includes the ability to alter the system’s diversity metrics, aggregation methods, test case representations, and reporting methods. There are eight requirements here:

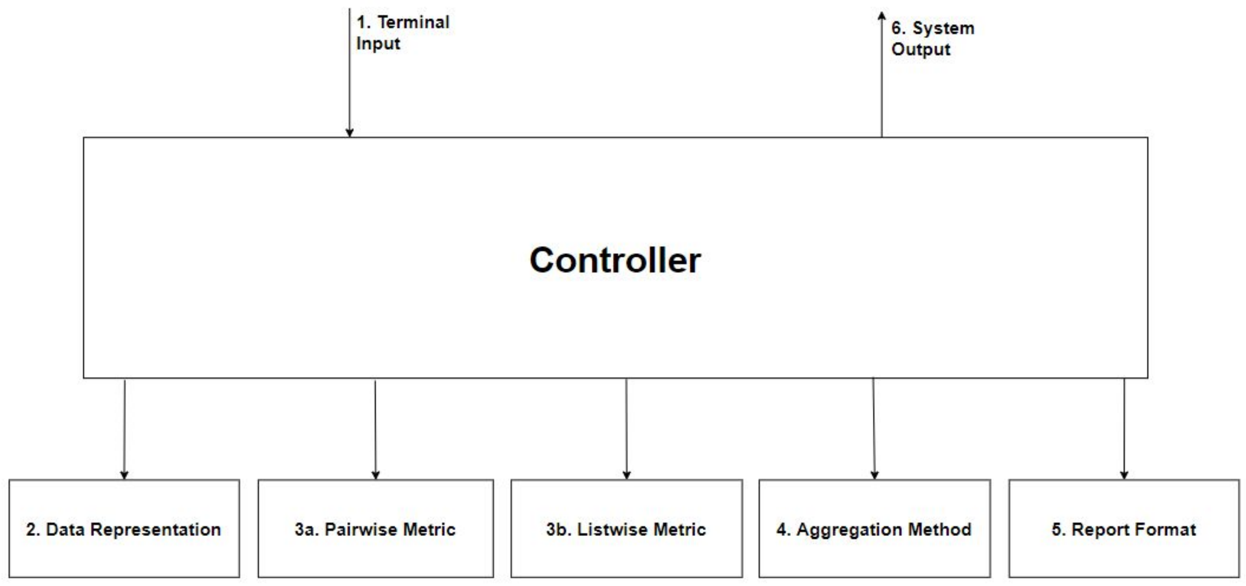
1. The source code should only need to be altered in a single file when modifying the behaviour of a diversity metric.
2. The source code should only require the addition of a single file when creating an additional diversity metric.
3. The source code should only need to be altered in a single file when modifying the behaviour of an aggregation method.
4. The source code should only require the addition of a single file when creating an additional aggregation method.
5. The source code should only need to be altered in a single file when modifying the behaviour of a reporting method.
6. The source code should only require the addition of a single file when creating an additional reporting method.
7. The source code should only need to be altered in a single file when modifying the format of a test case representation.
8. The source code should only require the addition of a single file when creating an additional test case representation.

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
comparisonMethod	The default diversity metric to use in the system if no metric is specified.	CommonElements
comparisonMethodLocation	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

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

This section provides several examples of instructions available to use in the system as an aid in how to use the system.

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:

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

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:

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

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 ith 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:

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

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:

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

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.

<TO DO WHEN REPORT FORMATS ARE DONE>

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 <JAR name> <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 the command 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 the command 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 the command 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 AggregationStrategymust 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 the command is "jar uvf <JAR name> metrics/aggregation/<method name>".

Adding a new Report Format

System Design

Overall Architecture

The architecture for the system follows a Model-View-Controller (MVC) approach. User input is designed to enter the system as command line arguments, so there is no input view. The output view is the ConsoleOutputService, which simply prints information to the terminal running the system. The controller centers around the Controller object, which coordinates several other controller objects that each perform specialized tasks. The model consists of DataTransferObjects, which hold the relevant information contained in the user's command; a Config object, which holds configuration information read in from a JSON file; and DataRepresentations, which are the representation of a test case in the system. Below is an image of the system structure, seperated into the model, view, and controller.

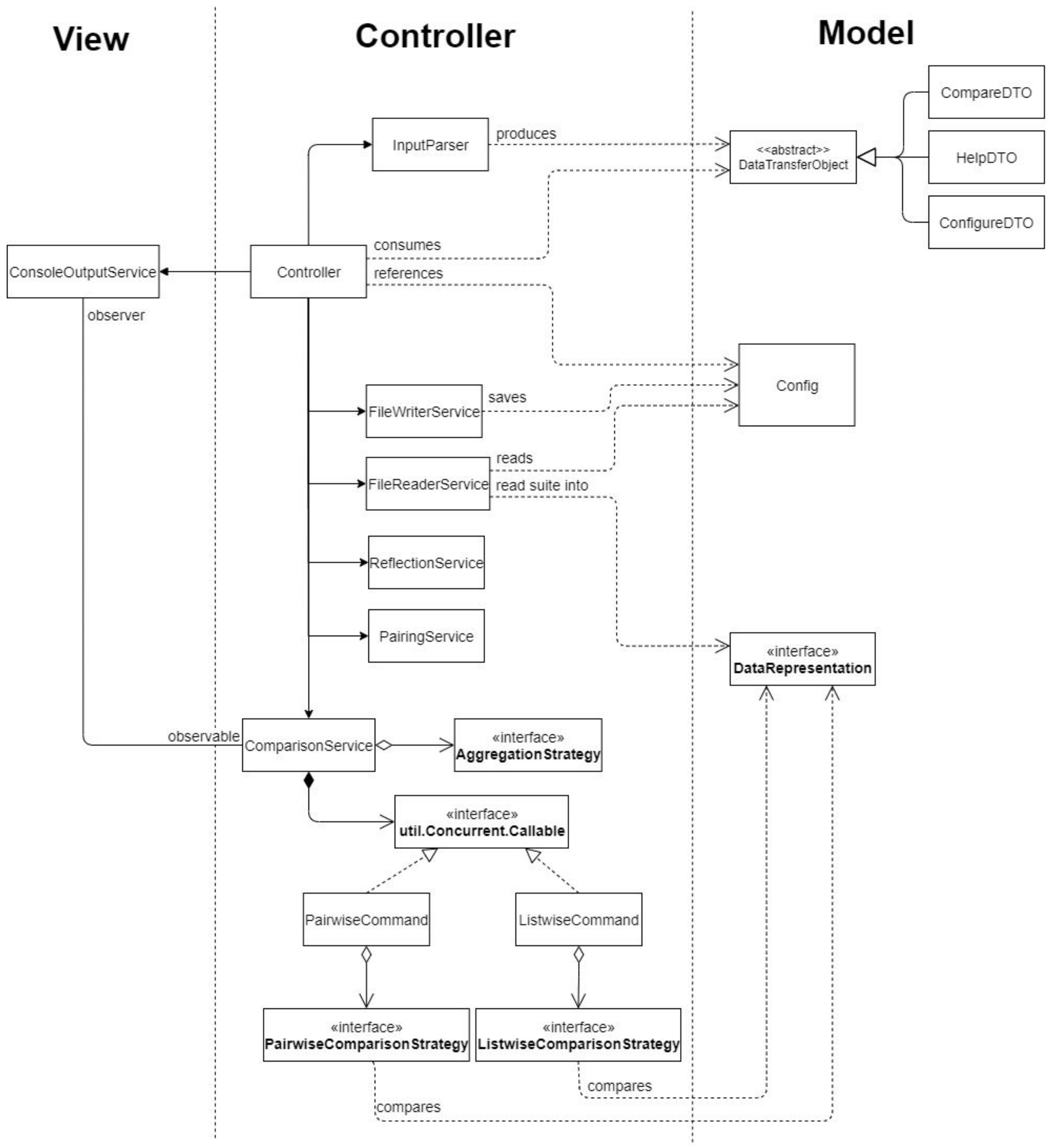


Figure _:The overall system design, separated into its MVC components.

Object Responsibilities

This section gives a short description of the use of each object in the diagram above.

View

ConsoleOutputService: Prints content to the terminal in which the program is running.

Controller

AggregationStrategy: The interface to be implemented by all aggregation methods.

ComparisonService: Performs the actual comparison of test cases.

Controller: Coordinates the use of other objects and performs error handling.

FileReaderService: Handles the reading of test case files and configuration files.

FileWriterService: Handles the writing of comparison results and configuration files.

InputParser: Converts text input from the terminal into a form usable by the system.

ListwiseCommand: A container for a ListwiseComparisonStrategy.

ListwiseComparisonStrategy: The interface to be implemented by all non-pairwise diversity metrics.

PairingService: Generates pairs of test cases from a test suite for pairwise comparisons.

PairwiseCommand: A container for a PairwiseComparisonStrategy.

PairwiseComparisonStrategy: The interface to be implemented by all pairwise diversity metrics.

ReflectionService: Provides the function to search system files for the required diversity metrics, aggregation strategies, and data representations.

Model

DataTransferObject: The abstract class extended by system command representations.

DataRepresentation: The interface to be implemented by all test case representations.

CompareDTO: The internal representation of a comparison command in the system.

Config: The system's internal representation of the configuration file.

ConfigureDTO: The internal representation of a configure command in the system.

HelpDTO: the internal representation of a help command in the system.

Coordination of Controller Objects

As previously mentioned, the Controller coordinates a number of adjacent objects to perform system commands. Each of these adjacent objects perform specialized tasks, and act as a way to group together all the similar controller functionalities together; for example, all the file reading is performed by the FileReaderService, and all the functionality for performing the actual comparisons in the system are encapsulated in the ComparisonService. This allows the system to be more modular so components can be replaced if needed, and if something breaks, it is easier to find where the issue is, based on what type of issue occurs. The Controller in the center orders calls to each of these adjacent objects in different orders to achieve different commands, and essentially just passes the output of one object as input to the next adjacent object. This design would also allow for much of the existing code to be reused in the event that completely new types of commands were to be added to the system. How exactly the Controller orders calls to adjacent objects for the existing commands is elaborated in the following sections.

Pairwise Comparisons

Comparisons made in the system are of one of two types: pairwise or listwise comparisons. The former of the two makes use of all the components in the system, as shown in the collaboration diagram below.

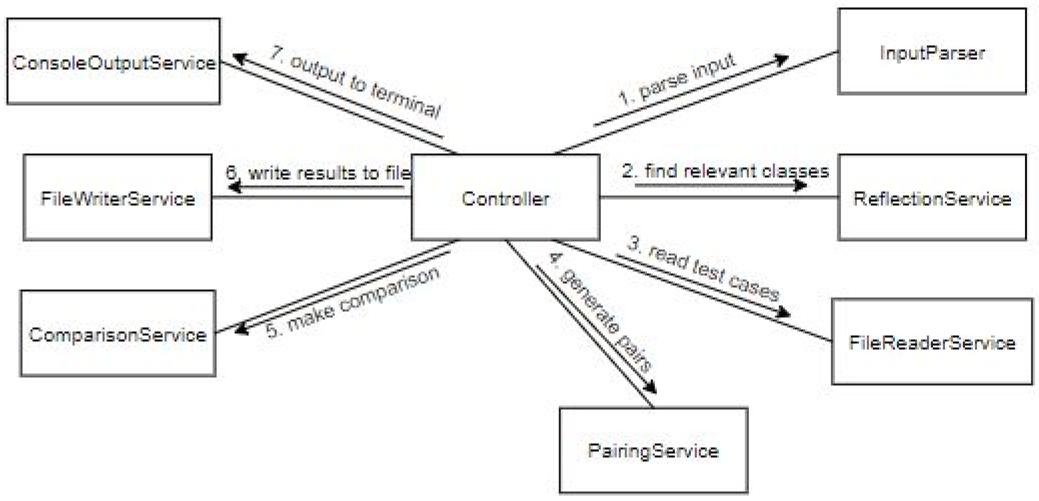


Figure _: Collaboration diagram of pairwise comparisons in the system.

As it can be seen above, when the system starts the first task is to parse the input, which is received through the command line. The InputParser does this, and returns an object containing all the relevant information from the command. This includes information such as what file(s) the test cases are stored in, what data representation those test cases are stored in, what metric should be used to perform the comparison, and what aggregation method should be used to format the output. Next, the ReflectionService is used to find and instantiate the relevant classes for the data representation, diversity metric and aggregation method. The FileReaderService is invoked next to read in the test cases according to the specified data representation object, and this list of test cases is passed to the PairingService to generate pairs of test cases to compare. The pairs of test cases, along with the diversity metric and aggregation method are passed to the ComparisonService which performs the calculation. Finally, the result is saved to a file (if the user specified this in their command) by the FileWriterService and is also displayed to the terminal by the ConsoleOutputService.

Listwise Comparisons

Listwise comparisons behave very similarly to pairwise comparisons, as seen below.

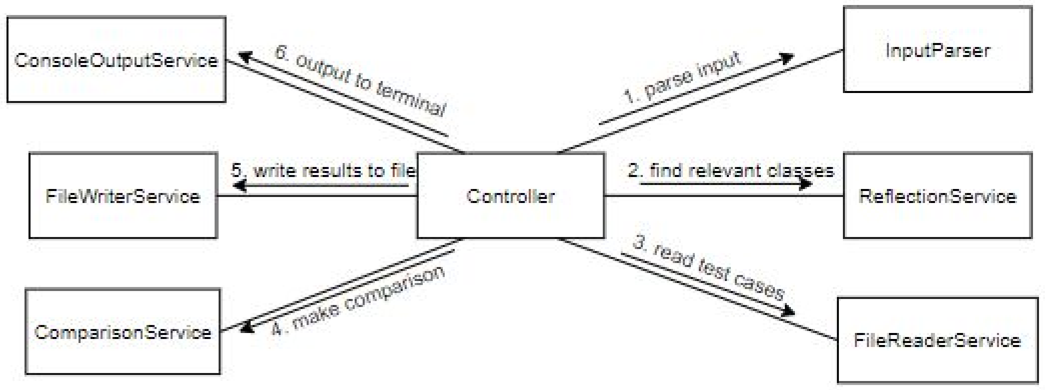


Figure _: Collaboration diagram of listwise comparisons in the system.

The key difference in listwise comparisons is that test cases do not need to be paired with each other, since the whole set of test cases is compared at once. As such, the order of calling and use of each object is the same as a pairwise comparison, with the omission of the call to the PairingService.

Configuration Commands

As previously mentioned, the system allows for safe configuration of system defaults through a “configure” command; exactly how this is done is detailed below.

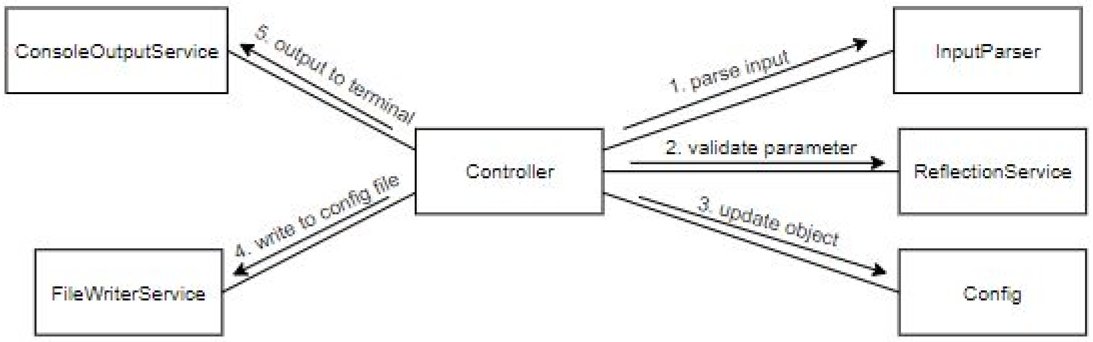


Figure _: Collaboration diagram of configure commands in the system.

Configuration commands start with the InputParser, which extracts the parameter and value to set from the user input. The ReflectionService validates this by using reflection to check the Config object (the internal representation of the configuration file) for a setter method that matches the parameter. The Config object is updated with the new value, which is then written to the JSON file by the FileWriterService. Finally, a message is displayed to the terminal indicating the operation's success by the ConsoleOutputService.

Help Commands

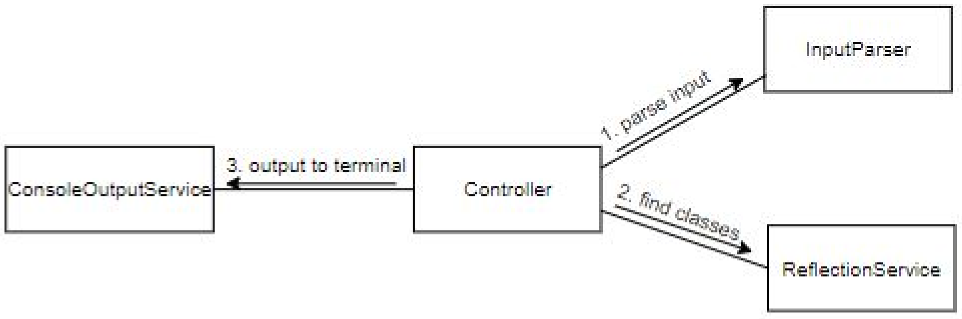


Figure _: Collaboration diagram of help commands in the system.

First the InputParser is used to extract the type of help requested from the user command. The ReflectionService is then invoked to find all the classes that map to the type of help required in order to obtain the descriptions of each; for example, a command to list the available diversity metrics in the system would scan the package which contains those metrics, and call the getDescription() method on each metric found. The compiled list of descriptions is displayed to the user through the ConsoleOutputService.

Design Patterns

Usage of design patterns has heavily influenced the overall system design. These patterns are listed discussed in the following sections, including the roles each system component plays, the rationale for using the design pattern, and how the usage compares to the general design of the pattern being used.

Data Transfer Objects for Input Parsing

The first step of any operation in the system is parsing the input obtained from the command line. This is performed by the InputParser object which takes the command line string as input and returns a data transfer object (DTO) containing the relevant information needed to perform the command.

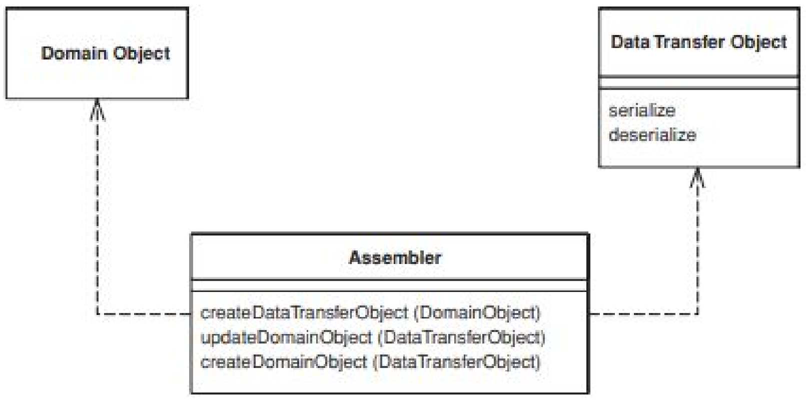
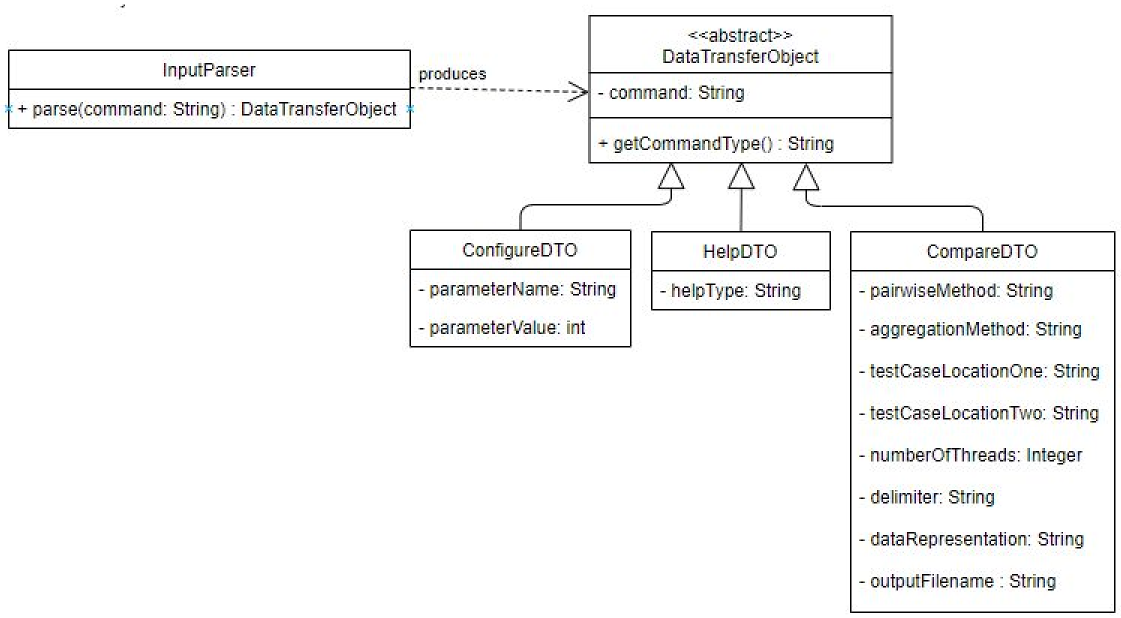

Domain Object
Data Transfer Object
serialize
deserialize
Assembler
createDataTransferObject (DomainObject)
updateDomainObject (DataTransferObject)
createDomainObject (DataTransferObject)

Figure _: The general design of a data transfer object pattern. [5]



InputParser
+ parse(command: String) : DataTransferObject
produces
<<abstract>>
DataTransferObject
- command: String
+ getCommandType() : String
ConfigureDTO
- parameterName: String
- parameterValue: int
HelpDTO
- helpType: String
CompareDTO
- pairwiseMethod: String
- aggregationMethod: String
- testCaseLocationOne: String
- testCaseLocationTwo: String
- numberOfThreads: Integer
- delimiter: String
- dataRepresentation: String
- outputFilename : String

Figure _: Use of data transfer objects in the system.

In the figures above, it can be seen that the InputParser plays the role of the Assembler in the general design. In the system, the domain object is the user input, which is a string, and the InputParser pulls the relevant information out of that string to populate the fields in a DTO.

There are a few differences between the general design and the system implementation of this design pattern. In the general design diagram, DTOs have a serialize and deserialize operation. This is because in an enterprise application, the DTOs are typically sent as part of a remote request or reply. [5] All the components of the system are hosted locally, so these methods are not required. The Assembler may also have methods for updating DTOs and converting a DTO back to a domain object, but since the InputParser only produces DTOs which are directly consumed by the Controller, these methods are not needed.

Early on it made sense to extract the relevant information from the command string, rather than require each component to do its own parsing of the string to find the information it needs. This meant delegation of parsing to a specialized component, and in order to return all that information from a single method, the information needed to be packaged into an object. Each type of command in the system has a seperate type of DTO, which avoids having one large DTO with many fields that are unused most of the time, because different types of commands have different possible flags and values to specify. The use of DTOs throughout the system further benefits the design because only the InputParser deals with the string commands from the user's command line. This means that the way a user supplies commands could be changed (eg. to a graphical interface or a remote call) and as long as that user interface can create DTOs, the rest of the system would not need to be changed.

Instantiating Classes with a Factory Method

The AggregationStrategy, PairwiseComparisonStrategy, ListwiseComparisonStrategy, and DataRepresentation are the interfaces meant to be extended by users of the framework to perform any comparison method on and data representation, and so the system needs a way to instantiate these in an extensible way. Rather than keep a list of what implementations of these interfaces exist in the system, they are found and instantiated through reflection in a factory method.

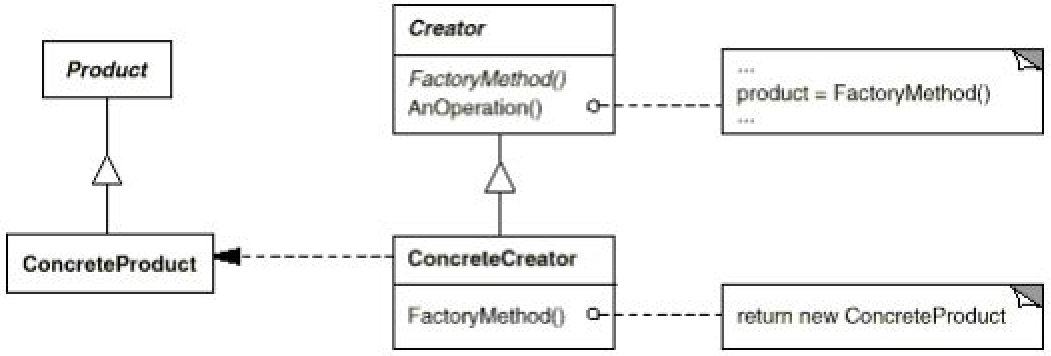


Figure _: The general Factory Method design. [6]

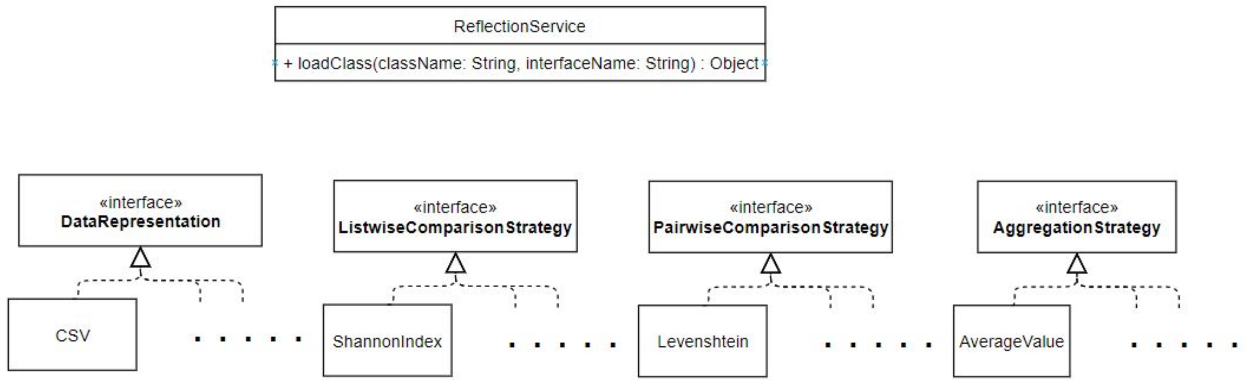


Figure _: The use of Factory Method in the system.

Each of the four interfaces are the Products, and their implementations are the ConcreteProducts. In the system, there is only the ConcreteCreator (the ReflectionService) because there is only one type of factory method, so a subclassing structure is not needed. The loadClass() of the ReflectionService is the system's factory method, but because objects are instantiated through reflection, there is not any strong association between the factory method and any of the ConcreteProducts it instantiates. The factory method takes two strings, the name of the class to instantiate and the name of an interface. The method looks for a class matching the specified name and instantiates the class if the class implements the provided interface name. This allows the use of the single factory method to be used for all four Products, instead of requiring a separate factory method each for DataRepresentation implementations and implementations of the aggregation/comparison strategies. This also means the factory method could be used to instantiate other types of Products if the project were continued in the future.

Strategy Pattern to Swap Out Diversity Metrics

In order to easily switch which aggregation methods, diversity metrics, and report formats are used in the system, a strategy pattern was implemented for each. The use of such a pattern (shown below) simply requires that each of the three groups of items implement a common interface amongst themselves and any of them can be used in the same context [7]. In the following diagrams, the concrete strategies used are all instantiated through the Factory Method discussed above.

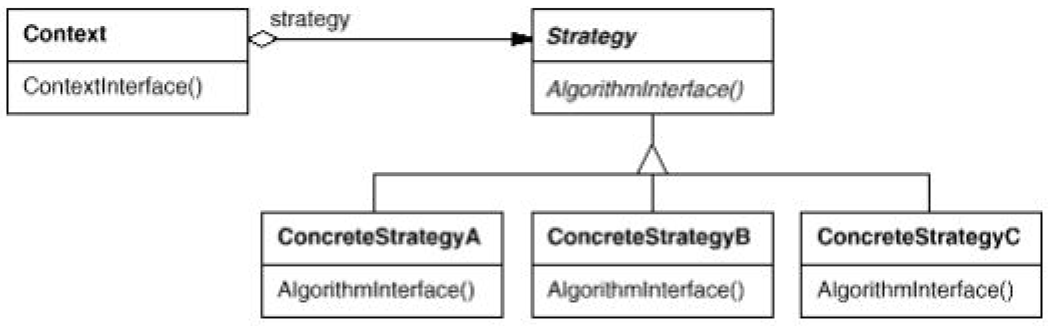


Figure _: The general Strategy Pattern design. [7]

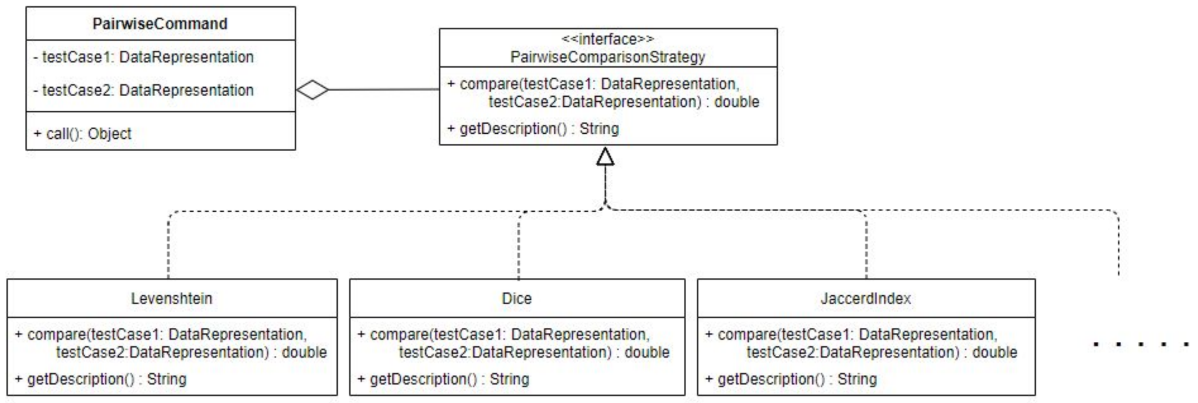

PairwiseCommand
- testCase1: DataRepresentation
- testCase2: DataRepresentation
+ call(): Object
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
+ getDescription() : String
Levenshtein
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
+ getDescription() : String
Dice
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
+ getDescription() : String
JaccerdIndex
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
+ getDescription() : String
. . . . .

Figure _: Strategy usage for pairwise comparisons in the system.

The usage of strategy pattern in pairwise comparisons (above) and listwise comparisons (below) strongly follow the general design for the pattern. For pairwise diversity metrics, context is the PairwiseCommand created by the Comparison Service, the Strategy is the PairwiseComparisonStrategy, the AlgorithmInterface is compare() , and the ConcreteStrategies are the various implementations of specified pairwise metrics, including Levenshtein, Dice, and JaccardIndex. For listwise metrics, the context is the ListwiseCommand, the Strategy is the ListwiseComparisonStrategy, the AlgorithmInterface is compare(), and the ConcreteStratgies are the many implemented listwise metrics, Including Nei, ShannonIndex, and SimpsonDiversity.

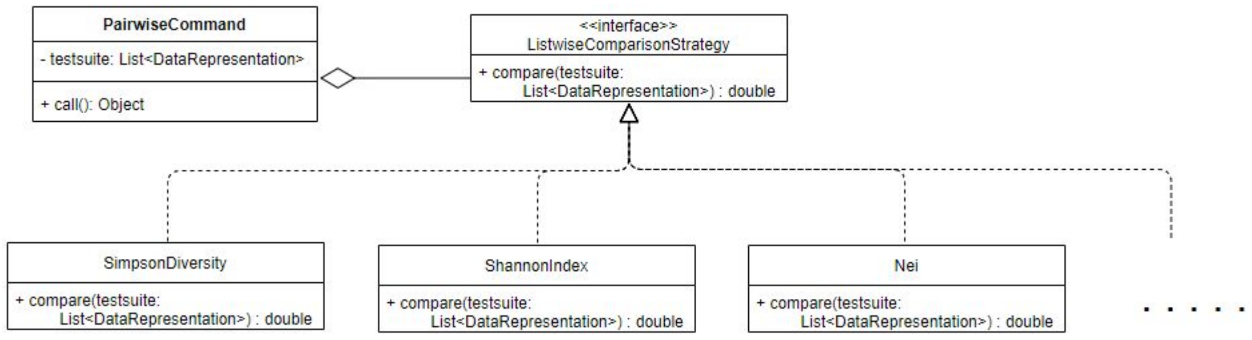

PairwiseCommand
- testsuite: List<DataRepresentation>
+ call(): Object
<<interface>>
ListwiseComparisonStrategy
+ compare(testsuite: List<DataRepresentation>) : double
SimpsonDiversity
+ compare(testsuite: List<DataRepresentation>) : double
ShannonIndex
+ compare(testsuite: List<DataRepresentation>) : double
Nei
+ compare(testsuite: List<DataRepresentation>) : double
. . . . .

Figure _: Strategy usage for listwise comparisons in the system.

The aggregation methods are implemented in a similar way. Once the ComparisonService completes its calculation using the diversity metric, the results are returned to the Controller and fed into the AggregationStrategy specified by the user. Like the diversity metrics, the user specifies the aggregation method to use, which is instantiated by the ReflectionService's Factory Method, and the common interface used by all aggregation methods allows them to be easily switched. In the diagram below, it can be seen that the Controller takes the role of the Context, the AggregationStrategy is the Strategy, the method aggregate() is the AlgorithmInterface, and the ConcreteStrategies are all the implemented aggregation methods discussed in the User Manual section, including AverageValue, Summation, and MaximumValue.

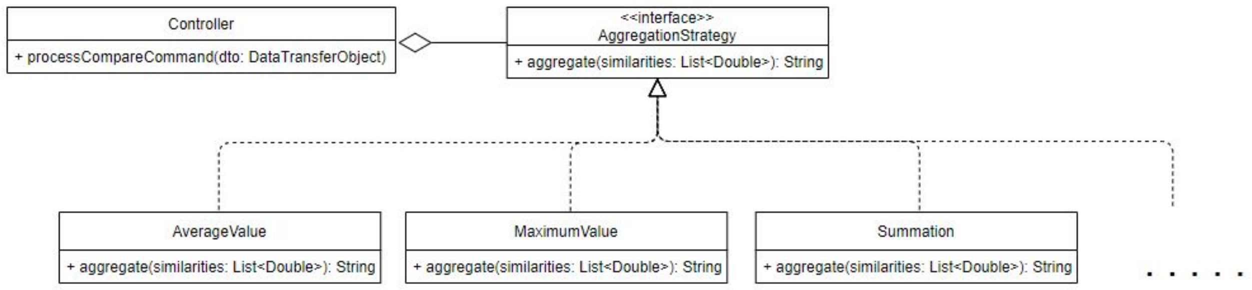

Controller
+ processCompareCommand(dto: DataTransferObject)
<<interface>>
AggregationStrategy
+ aggregate(similarities: List<Double>): String
AverageValue
+ aggregate(similarities: List<Double>): String
MaximumValue
+ aggregate(similarities: List<Double>): String
Summation
+ aggregate(similarities: List<Double>): String
. . . . .

Figure _: Strategy usage for aggregation methods in the system.

INCLUDE THE REPORTING METHOD HERE WHEN IT IS READY

Bridge Relationship Between Data Representations and Metrics

One of the defining parts of the system is that any diversity metric should be able to be used on any type of test case data representation. Implementing a version of a diversity metric that could work on each supported data representation would quickly become infeasible - the two groups of classes (diversity metrics and data representations) needed to be able to vary independently. For this, a Bridge Pattern seemed like a strong choice. The use of this pattern, as seen below, allows helps to decouple, and allows greater extensibility of, the diversity metrics and data representations [8].

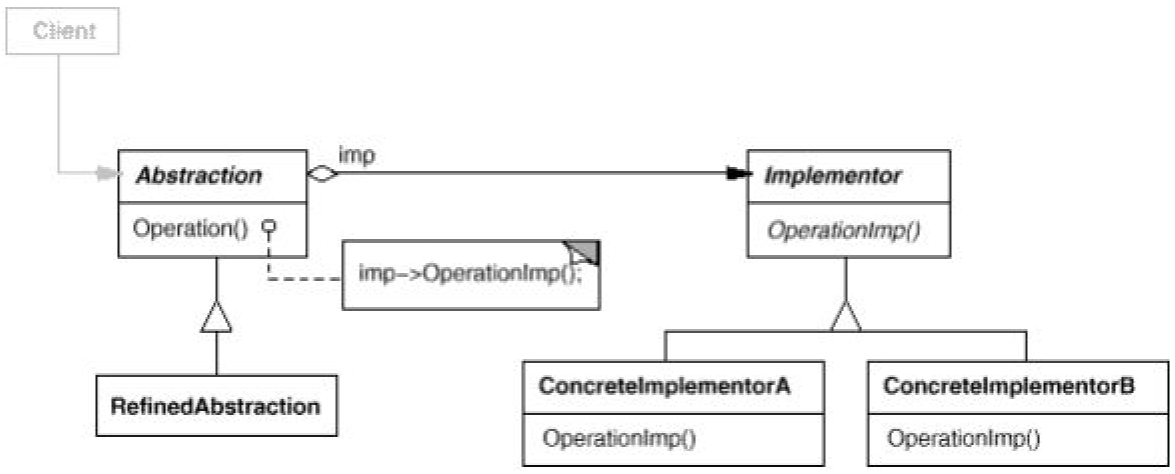


Figure _: The general structure of a Bridge Pattern. [8]

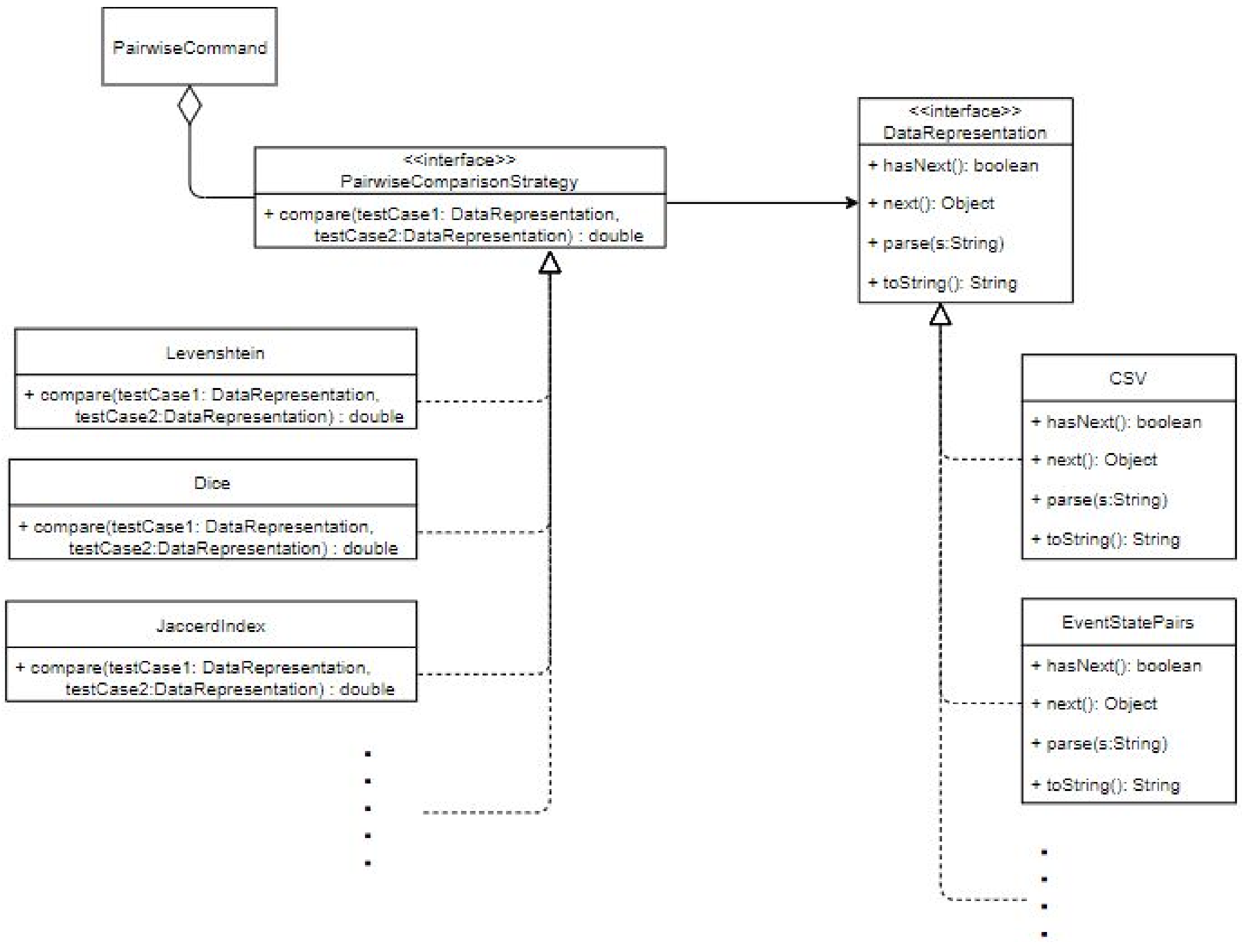

PairwiseCommand
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
<<interface>>
DataRepresentation
+ hasNext(): boolean
+ next(): Object
+ parse(s:String)
+ toString(): String
Levenshtein
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
Dice
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
JaccerdIndex
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
CSV
+ hasNext(): boolean
+ next(): Object
+ parse(s:String)
+ toString(): String
EventStatePairs
+ hasNext(): boolean
+ next(): Object
+ parse(s:String)
+ toString(): String

Figure _: Use of Bridge Pattern between pairwise metrics and data representations.

For the bridge between pairwise metrics and data representations (seen above), the PairwiseCommand is the client that initiates the Abstraction's operation. The Abstraction is the PairwiseComparisonStrategy and the RefinedAbstractions are the implemented pairwise metrics. These RefinedAbstractions use the methods specified in the DataRepresentation, which takes the role of Implementor. The ConcreteImplementors are the data representations implemented in the system. The pattern is very similar for listwise metrics (seen below). Here, the client is the ListwiseCommand, and the Abstraction and RefinedAbstractions are the ListwiseComparisonStrategy and implemented listwise metrics, respectively. The Implementor and ConcreteImplementors are the same as those for the pairwise bridge.

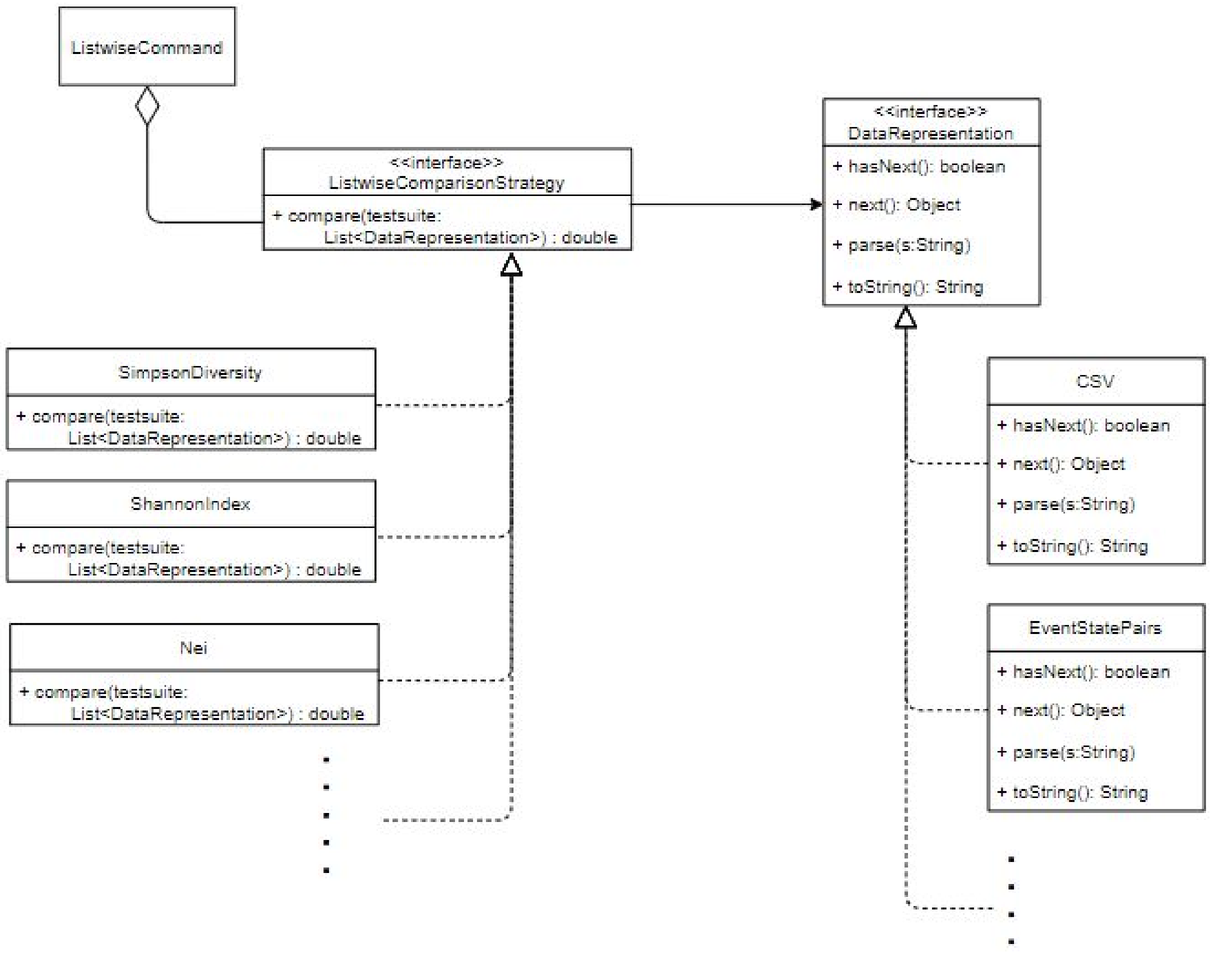


Figure _: Use of Bridge Pattern between listwise metrics and data representations.

Iterator-Like Interface for Data Representations

From the Bridge Patterns described above, it can be seen that the interface for a Representation is very similar to an Iterator. In a general Iterator Pattern (shown below), the Iterator and collection being iterated over (the Aggregate) are separate objects, which offers many benefits. [9] It should be noted that the diagram below is the general diagram for an Iterator. In Java, Iterators have a slightly different interface. Iterators in Java have 3 methods: next(); hasNext(), which replaces isDone(), and remove(), which is optional. [10] It can be seen in the diagrams above that DataRepresentations implement only next() and hasNext().

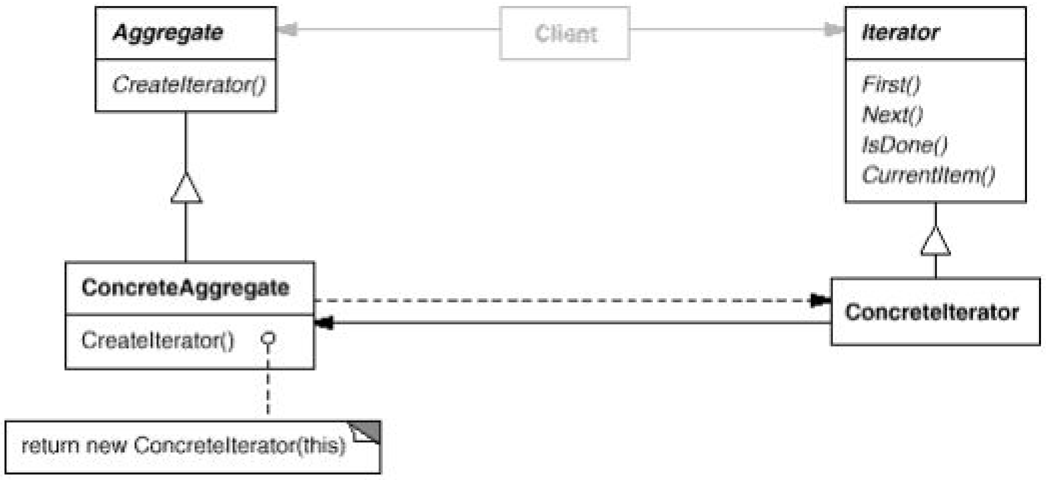


Figure _: The general design of an Iterator Pattern. [9]

The system does not get these benefits, because the system DataRepresentations do not iterate over a referenced object. Instead, the DataRepresentation interface methods iterate over an internal collection that is private to the DataRepresentation implementation. Iteration over an internal representation is chosen so that only a single file is needed to be added to the system to support a new test case format, otherwise two files would need to be added: one for the test case format, and another for iterating over that test case. By storing the test case in a private field in the DataRepresentation, there is also much more freedom in how the test case is stored. The goal of providing an Iterator-like interface for DataRepresentations is to provide an interface that looks familiar to those that want to provide new diversity metrics to the system.

Observer Pattern for Displaying Operation Progress

Operations on large test suites can be quite computationally intensive. Pairwise comparisons are particularly susceptible to this because as test suite size grows the number of pairs to make comparisons on grows rapidly. The number of pairs generated from a test suite is generated is equal to the number of subsets of size two that can be generated from that set of test cases, which is given by the formula below:

$$\frac{n(n-1)}{2}$$ [11]

Where n is the number of test cases in the test suite. With this it can be seen that a testsuite of 1000 test cases, there are almost half a million pairs of test cases. This means that half a million pairs must be generated, and this number of comparisons must be performed. Concurrency was introduced here since each pair generation and pairwise comparison is independent, but this can still take some time for large test suites.

The ConsoleOutputService is used to display error messages and results to the terminal, but while a successful operation is underway, the terminal was originally blank. It would be hard to tell for large operations if the system was performing properly before either results or error were displayed, and not indication of how long a comparison might take, so Observer Patterns were introduced between the ConsoleOutputService and PairingService, and between the ConsoleOutputService and ComparisonService in order to display a progress bar on the terminal. With this, the user can now tell that their operation is progressing properly, and they can get an idea of how much longer the operation will take.

It should be noted that in Java, there are interfaces for Observer and Observable, but these are deprecated. Instead, Observers implement ProperyChangeListener, and Observables must instantiate a PropertyChangeSupport object, add listeners to this object, and then fire a new PropertyChangeEvent that is sent to all added PropertyChangeListeners. [12]

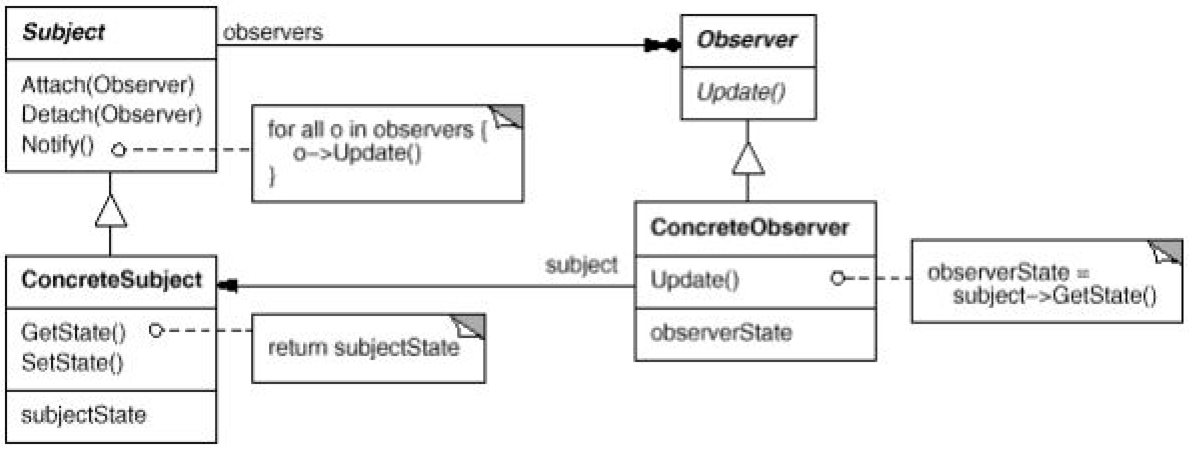


Figure _: The general design of an Observer Pattern. [13]

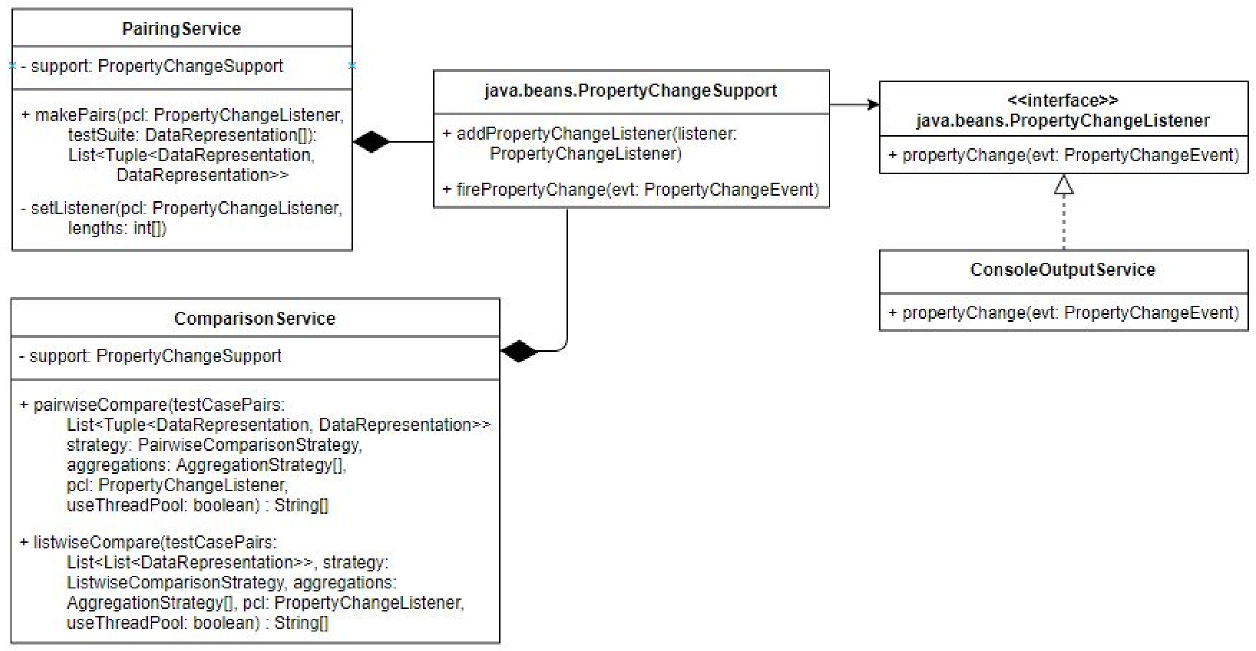

PairingService
- support: PropertyChangeSupport
+ makePairs(pcl: PropertyChangeListener, testSuite: DataRepresentation[]): List<Tuple<DataRepresentation, DataRepresentation>>
- setListener(pcl: PropertyChangeListener, lengths: int[])
java.beans.PropertyChangeSupport
+ addPropertyChangeListener(listener: PropertyChangeListener)
+ firePropertyChange(evt: PropertyChangeEvent)
<<interface>> java.beans.PropertyChangeListener
+ propertyChange(evt: PropertyChangeEvent)
ConsoleOutputService
+ propertyChange(evt: PropertyChangeEvent)
ComparisonService
- support: PropertyChangeSupport
+ pairwiseCompare(testCasePairs: List<Tuple<DataRepresentation, DataRepresentation>> strategy: PairwiseComparisonStrategy, aggregations: AggregationStrategy[], pcl: PropertyChangeListener, useThreadPool: boolean) : String[]
+ listwiseCompare(testCasePairs: List<List<DataRepresentation>>, strategy: ListwiseComparisonStrategy, aggregations: AggregationStrategy[], pcl: PropertyChangeListener, useThreadPool: boolean) : String[]

Figure _: The Observer Pattern implemented in the system.

In the above diagrams, it can be seen that PropertyChangeListener in the system takes the role of Observer in the general design, and ConsoleOutpurService is the ConcreteObserver. The PairingService and ComparisonService are ConcreteSubjects, but they do not implement a common interface. This is due to the difference in how Java implements an Observer Pattern. The PropertyChangeSupport, which is instantiated by the PairingService and ComparisonService, is what acts the Subject. The attach() method in the general design is replaced by the addPropertyChangeListener() method, and the notify() method in the general design is the firePropertyChange() method. On the observer side, the update() method in the general design is implemented as the propertyChange() method.

Command Pattern to Execute Comparisons in a Thread Pool

In order to improve the performance of large operations, a thread pool is implemented. The use of a thread pool requires encapsulation of parts of that large operation in objects that implement a similar interface, and so Command Pattern is inherent to their usage. The diagrams for the general design of a Command Pattern and for the implementation in the system's ComparisonService are shown below.

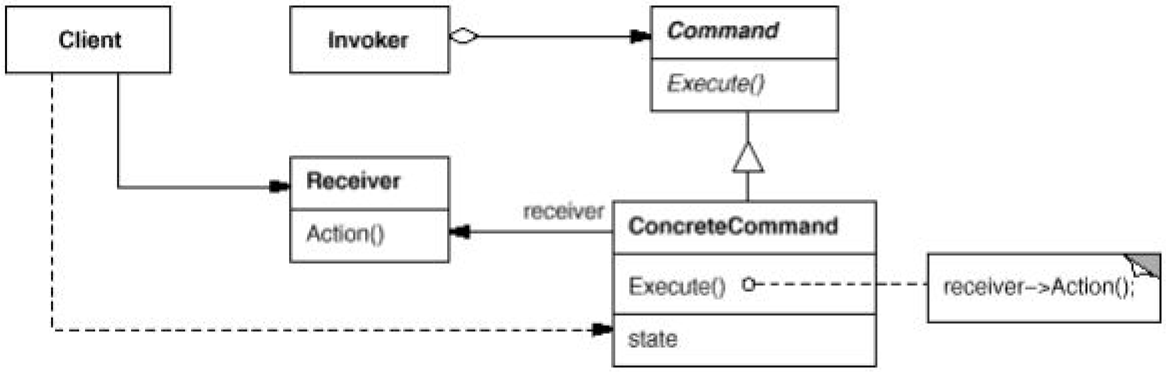
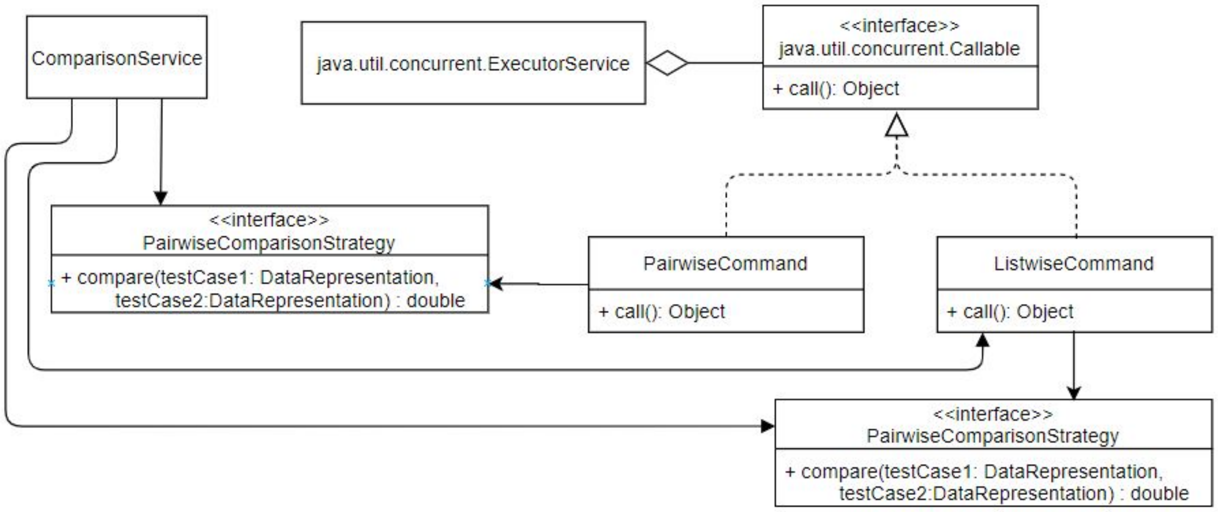


Figure _: The general design of a Command Pattern. [14]



ComparisonService
java.util.concurrent.ExecutorService
<<interface>>
java.util.concurrent.Callable
+ call(): Object
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
PairwiseCommand
+ call(): Object
ListwiseCommand
+ call(): Object
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double

Figure _: The implementation of Command Pattern for comparisons in the system.

When the comparison service is used, it generates several PairwiseCommands/ListwiseCommands that reference the diversity metric to use. Each of these command objects are invoked by the thread pool to generate comparison results. From the side-by-side diagrams it can be seen that the ComparisonService is the Client, the Invoker is the ExecutorService (ie. the thread pool), and the Command interface is Callable, the interface that an ExecutorService's tasks must implement. The ConcreteCommands are PairwiseCommand and ListwiseCommand, each of which have their own Receivers; for PairwiseCommand, it is the PairwiseComparisonStrategy, and for ListwiseCommand, it is the ListwiseComparisonStrategy.

The PairingService also uses a thread pool to aid in generating test case pairs, and so it also implements a Command Pattern. The implementation can be seen below, and is similar to the diagram above. The invoker and Command remain the same, but the ConcreteCommand is now a PairingCommand and the Client is the PairingService. There is no separate Receiver object for the PairingCommands. Instead the call() method of a PairingCommand contains the code to generate the test pairs. In the PairingService, each PairingCommand is created with a subset of the total pairs to create, and are all invoked by the thread pool to get the results.

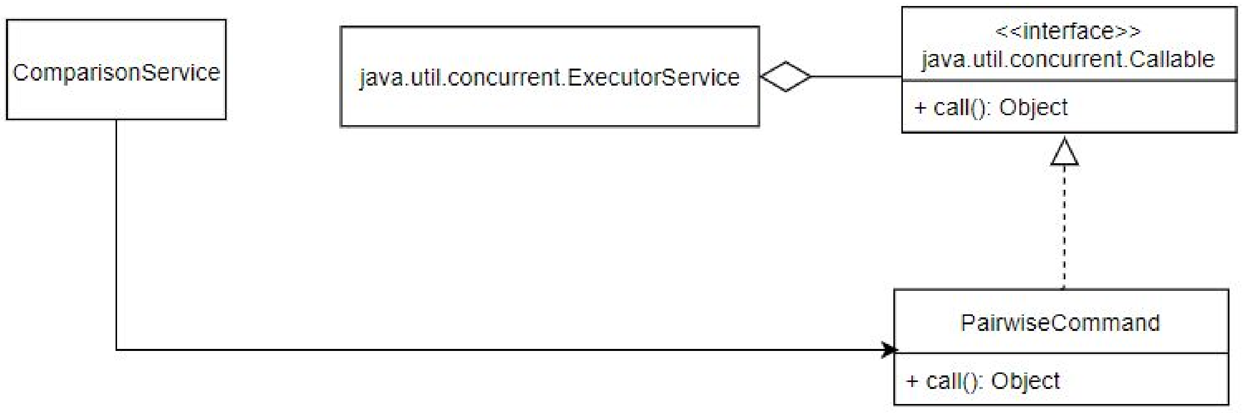


Figure _: The implementation of Command Pattern for pairing test cases in the system.

Software Testing

- Bottom up unit testing to reduce need for stubbing
- generate tests through black box technique: input domain modelling
  - Many tests end up being base block adequate, because we want to test different types of errors
- Aim to be all-edges adequate
  - Most free coverage tools only provide functionality this detailed

Case Study

-graph goes here

Challenges Encountered During the Project

- Discuss the issue where multiple pairs would try to consume the same instance of an object, leading to errors. This was fixed but performance was degraded

Next Steps

Allow Multiple Commands to be Queued

Reduce Controller Size with Aspect Oriented Programming

Use as a Plugin

Team Member Contributions

Contributions to Final Report

The following table splits this report into several sections, and lists the primary author for each to show team member contributions.

Report Section	Primary Author(s)
Introduction	Cameron Rushton
Problem Statement	Eric Bedard
Relation to Degree	Eric Bedard
Project Management	Eric Bedard
Software Tools Used	Cameron Rushton
Project Scheduling	Cameron Rushton
Requirements	Cameron Rushton, Eric Bedard
User Manual	Luke Newton
System Design	Luke Newton
Software Testing	Eric Bedard
Case Study	Luke Newton
Challenges Encountered	Luke Newton
Next Steps	Cameron Rushton
Conclusion	Eric Bedard

Contributions to Software System

The following table details the primary author of each part of the software system. Here, the software system is broken down into components for the central controller, the adjacent services, the model components, and implementations of diversity metrics, aggregation methods, and report formats.

System Component	**Primary Author**
Controller	Luke Newton
InputParser	Luke Newton
ReflectionService	Cameron Rushton
PairingService	Eric Bedard
ComparisonService	Eric Bedard
FileReaderService	Eric Bedard
FileWriterService	Eric Bedard
ConsoleOutputService	Luke Newton
DataTransferObjects	Cameron Rushton
Configuration File	Cameron Rushton
DataRepresentations	Luke Newton
PairwiseComparisonStrategies	Luke Newton
ListwiseComparisonStrategies	Luke Newton
AggregationStrategies	Cameron Rushton
ReportFormats	Cameron Rushton

Conclusion

References

[1] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge: Cambridge University Press, 1997.

[2] R. Feldt, S. M. Poulding, D. Clark, S. Yoo, Test set diameter: Quantifying the diversity of sets of test cases, CoRR, 2015.

[3] S. H. N. Asoudeh Khalajani, Test Generation from an Extended Finite State Machine as a Multiobjective Optimization Problem, Ottawa: Carleton University, 2016.

[4] H. Hemmati, L. Briand, An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection, 21st IEEE International Symposium on Software Reliability Engineering (ISSRE), 2010

[5] M. Fowler, "Data Transfer Object," in Patterns of Enterprise Application Architecture, Boston, Pearson Education, 2003, pp. 401-413.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Factory Method ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 106-114.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Strategy ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 292-300.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Bridge ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 146-155.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Iterator ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 241-264.

[10] Oracle, "Interface Iterator<E>," [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html. [Accessed 19 February 2020].

[11] H. J. Ryser, "Unordered Selections," in Combinatorial Mathematics, The Mathematical Association of America, 1963, p. 9.

[12] L. Vogel, "Observer Design Pattern in Java - Tutorial," 29 9 2016. [Online]. Available: https://www.vogella.com/tutorials/DesignPatternObserver/article.html. [Accessed 19 February 2020].

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Observer ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 273-282.

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Command ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 219-228.