Tool Framework to Measure Test Case Diversity: Final Report

Project Group 19

Submitted: April 6, 2020

Project Supervisor:

Yvan Labiche

Project Group Members:

Eric Bedard	101009607
Luke Newton	100999309
Cameron Rushton	101002958

Table of Contents

1 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 example, for a state machine, a test case may be seen as 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):

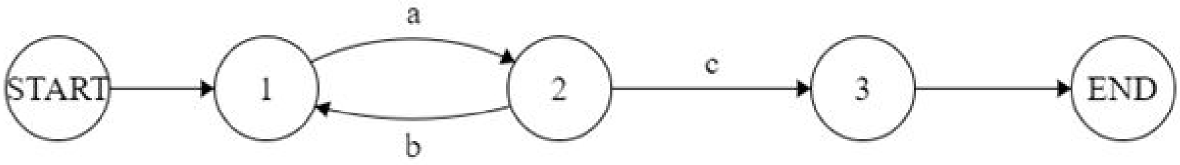


Figure 1: A simple state machine.

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.

In addition to metrics which compare sets versus sequences, there is another distinction to be made. Some metrics make “pairwise” comparisons, meaning every possible pairing of test cases are compared against each other. An example of this is a Hamming distance, which counts the number of elements in two sequences that are unequal. [4] Given the following four sequences:

1. 1,1,0,1,0,1,0,1
2. 1,0,1,0,1,1,1,1
3. 0,0,0,1,0,1,0,1
4. 1,0,1,1,1,1,1,1

Six individual calculations need to be made that compare the pairs (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), and (3, 4). For a Hamming distance, this gives the results [5, 2, 4, 5, 1, 4]. Other metrics make “listwise” comparisons, meaning that the metric considers all the test cases in a test suite are compared at once. An example of this is Simpson’s Diversity which is given by the formula:

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

where P is the set of sequences, q_r is the frequency of element r, and S is the total number of different elements in all sequences. [3] This describes the probability that, for any two elements randomly selected from a bag[1] of all elements in the sequences, the two elements selected are not equal. For the four sequences above, the Simpson Diversity is about 0.45.

[1] An unordered collection that contains duplicates

These listwise metrics would only produce a single value, but a pairwise metric for a test suite would produce as many values as there are possible pairings for the test cases within the suite. The pairwise results require an aggregation method to combine all the results into one representative value similar to what the listwise metrics provide. Different aggregation methods include the average pairwise diversity or median diversity, and further complicate things. A good pairwise metric could be chosen to find similarity, but the aggregation method may be incorrectly chosen and produce a value that is still not representative of the test suite's actual diversity.

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, aggregation method, 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.

2 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, aggregation methods, and test case representations. The software tool should be able to provide output in different forms to help interface with other tools in the future, and the addition of new output formats should also be easy.

The project will execute a full software lifecycle over the course of the academic year. The main use of the software developed is expected to be research to help answer some of the issues revealed in the *Introduction* section.

3 The Engineering Project

3.1 Health and Safety

The project itself is purely software, which means the usual health and safety does not apply. There was no handling of hazardous materials or requirement for personal protective equipment (PPE) for the development or usage of the product created for this project. There are some considerations on the safe use of computer resources, as well as ergonomic concerns.

The concern for safe use of computer resources deals with safe use of files and computational resources. If a command is issued to the system that takes too long, triggers an unknown fault resulting in infinite execution time, or if the command is triggered accidentally, there should be a way to stop the command so it does not unnecessarily consume system resources. The tool is designed to run within a terminal, and so it is easy to abort the tool at any time in the same manner any application can be stopped by a terminal. To maintain system file integrity, the system is defined in such a way that reading input and writing any necessary output are separate tasks. The handler for file input only has read permissions so input resources cannot be destroyed, and the handler for file output requires permission from a user if any attempted file writing would overwrite/append to an existing file. These concerns are expressed in the *Project Requirements* section later in the document, which were developed early on and carried through the project lifecycle.

Potential ergonomic issues during the development of the project are no different than the usual issues that can arise in software development. These include repetitive tasks, working in awkward positions for long periods of time, and eye strain. The members of the team are all familiar with these potential issues and thus take the usual precautions such as frequent breaks from working, sitting up straight with displays at eye level, and positioning hands to avoid wrist injury. There are measures specifically decided for this project as well. By selecting a software lifecycle where most planning is done upfront and by developing a strict schedule for work, as discussed later in *Project Management and Project Scheduling*, we aimed to avoid the need for a rush at the end of the project to complete all the work (ie. a hackathon to finish the project). Such a situation leaves little opportunity for the usual preventative measures for these ergonomic concerns. By doing accurate planning, ergonomic issues were avoided, as well as psychological concerns such as increased stress.

3.2 Engineering Professionalism

The main concerns for engineering professionalism in this project are the ability to work in teams and conflict resolution. In this project, there was little relevance of other professionalism concerns, such as conflict of interest, unauthorized disclosure, or mishandling of resources/money.

As with any project, team members were expected to act professional and be civil about any issues that arose. Since all the team members had worked together before, this was easier to maintain than it could be for a new group of students. Each team member is aware of the others' strengths and weaknesses, and we are familiar with each others' course schedules, so work could be assigned accordingly. With work assignments tailored to an individual's skills and availability, we mitigated the possibility for any conflict to arise.

Decisions on work, as well as other project decisions, were made collaboratively so the group members' preferences and ideas were heard. When there are no preferences from members, work would be assigned by one individual, but this was not commonly the case. In doing so, all group members feel that their ideas are heard, and there is less chance of conflict.

When conflict of ideas did arise, or when assigned work was not completed, the approach to resolution was not competitive or accusatory, but rather accommodating. For idea conflicts, group members made an effort to see the pros and cons of the different options instead of a “my idea is better no matter what” approach. When work was not completed on time, the approach was to determine why that was the case, and help in any possible way, rather than accuse and punish. An attempt was made to help members understand why a particular solution was not working, along with suggestions on how something a group member was stuck on could be solved. If the work took longer than expected because it was more complicated than originally thought, this was especially important to understand rather than blame the group member, because it could impact the design and solutions to other parts of the system. By taking this type of approach to solving problems, conflicts were resolved sooner, and the resolution of those conflicts did not cause resentment or fuel further conflicts later in the project.

3.3 Project Management

Before development started, The design of the system needed to be created. To do this, each team member was tasked with creating a class diagram of the system over the course of a week. Doing this prevented any ideas from being left out, and also confirmed that each member understood the goals of the project. When all the ideas were shared at the end of the week, all three diagrams were very similar. This was good, since it meant that each member had the same idea for the system and nobody misunderstood requirements for the system. The best ideas from these designs were then combined to create the real design for the system.

An incremental development cycle was chosen for this project with two increments, a minimum viable product to be ready for the presentation to the supervisor and second reader and a final product. The goal of the minimum viable product was to have a system that could calculate test case diversity according to one diversity metric, and contained all the infrastructure to be extensible. This was completed before the presentation so we could receive feedback on the system before we are locked in with too many implementations of aggregation and diversity metrics. We could then use the feedback to improve the system while working towards the final iteration, the final product. This iteration aims to be as complete as possible, containing many diversity metrics and output formats, and supporting several formats of test cases. This development cycle was very useful, because the feedback we received on the first iteration could be addressed earlier and gave us more confidence that the system we built matched the project supervisors expectations.

During each of the iterations, work was divided into issues. Each issue would represent a feature to be implemented into the system. These were all tracked on the projects Kanban board in github. When an issue is being worked on, it would be put into the development state and the developer would assign himself to the issue. A new “branch” will be used to develop the issue. Once the development is complete, the developer would create a “pull request”, which is a request for the other team members to review the code. Once all comments are addressed and the pull request is approved, the feature would be put into the main branch of the system. The issue will then be assigned to another developer, and that developer will test the feature and create unit tests. Once another pull request to review the tests is approved, the tests are put into the main branch and the issue is finished.

3.4 Justification of Suitability for Degree Program

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.

3.5 Individual Contributions

3.5.1 Table of Contributions to Final Report

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

Report Section	Primary Author(s)
Introduction	Cameron Rushton
Problem Statement	Eric Bedard
Health and Safety	Luke Newton
Engineering Professionalism	Luke Newton
Project Management	Eric Bedard
Suitability to Degree	Eric Bedard
Software Tools Used	Cameron Rushton
Project Scheduling	Cameron Rushton
Engineering Economics	Luke Newton
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

3.5.2 Table of 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

3.5.3 Contributions Summarized by Each Team Member

In a purely software project such as this, it can be difficult to give each team member well defined roles. In the case of this project, the software is also to be deployed on a single machine, so roles could not be divided by components this way either. The tables above list the primary authors for software components and aspects of the final report, but the majority of each of these are pieces that were worked on at least partially by two or all three of the team members. In light of this, each team member has included a short paragraph below to summarize what they feel their major contributions to the project were.

3.5.3.1 Contributions by Eric Bedard

The first major contribution I made during the implementation of the system was creating the comparisons service class (originally called comparator) with a thread pool. This involved researching how to handle giving a user the option to choose how many threads should be used for the calculations. Another contribution is the implementation of the file writer service, which was built in a way to avoid overwriting files unless specified by the user, which was one of our non-functional requirements. My last major contribution was finding a way to round the final result of the calculations by an amount specified by the user in the configuration file, and then implementing it in the system.

3.5.3.2 Contributions by Luke Newton

The process for writing each deliverable always involved every member, since our approach for most documents was to list the points for discussion in each section together, then individuals would write the sections out in full themselves. When it came to this, I usually would formalize more sections, as well as formalize more of the technical diagrams, but this is because I have a lighter course load than the other members, and thus more time to write. The ideas discussed in the deliverables and technical diagrams are still the result of a team effort and are more or less evenly contributed to by all members.

For the actual code written, the terminal I/O, and the Controller class which links everything together were mostly written by me, though others did contribute to those specific pieces as later features were added. I also wrote the majority of the implementations for the diversity metrics, both pairwise and listwise. This is because early on in the development lifecycle when we spent two weeks researching how we could build the system, I focussed most on what diversity metrics could be included, so I had more understanding/familiarity with them.

3.5.3.3 Contributions by Cameron Rushton

Besides collaborating with the group in various aspects of the system, the most individual tasks performed started with research and development on how we were going to handle reflection to make the project more scalable without relying on a framework. Further work was done on the reflection service, implementing the requirement of an interface that the instantiated object implements. Next, I figured out how we can leverage a JSON configuration file that can be easily editable through the command line or from the file itself. I proposed another solution by example using java's built-in properties API, but using gson to parse a JSON file met our requirements better. During this time, I also wrote a couple test suites for our data representations among others. I then started work on writing the aggregation strategies, implementing all the ones outlined in our research document and provided minimal test coverage for someone else to pick up and understand the new feature. Finally, the last large change was the report formatting where I implemented all the formats with test coverage and refactored small parts of the controller and comparison service to consolidate data in one spot to pass to the formatter.

4 Software Tools Used

The programming language used to develop the project is Java. This was part of the specification in the original project description by the project supervisor on the 4th year website. While it was not strictly required to be developed in this language, the supervisor's suggestion was used, because all team members are familiar with the language, and the system is meant to be extensible. As the primary user for the tool will be the project supervisor, the supervisor will be the primary one to extend the system, so the group resolved to use the language the supervisor suggested.

Some obvious tools to use then, are the java runtime environment (JRE) for running the application’s jar file and the java development kit (JDK) for development. The application was developed using version 1.8 since it is a well proven version of java and with this, we’re able to use more advanced functions such as streams without forcing the user to update to the latest version.

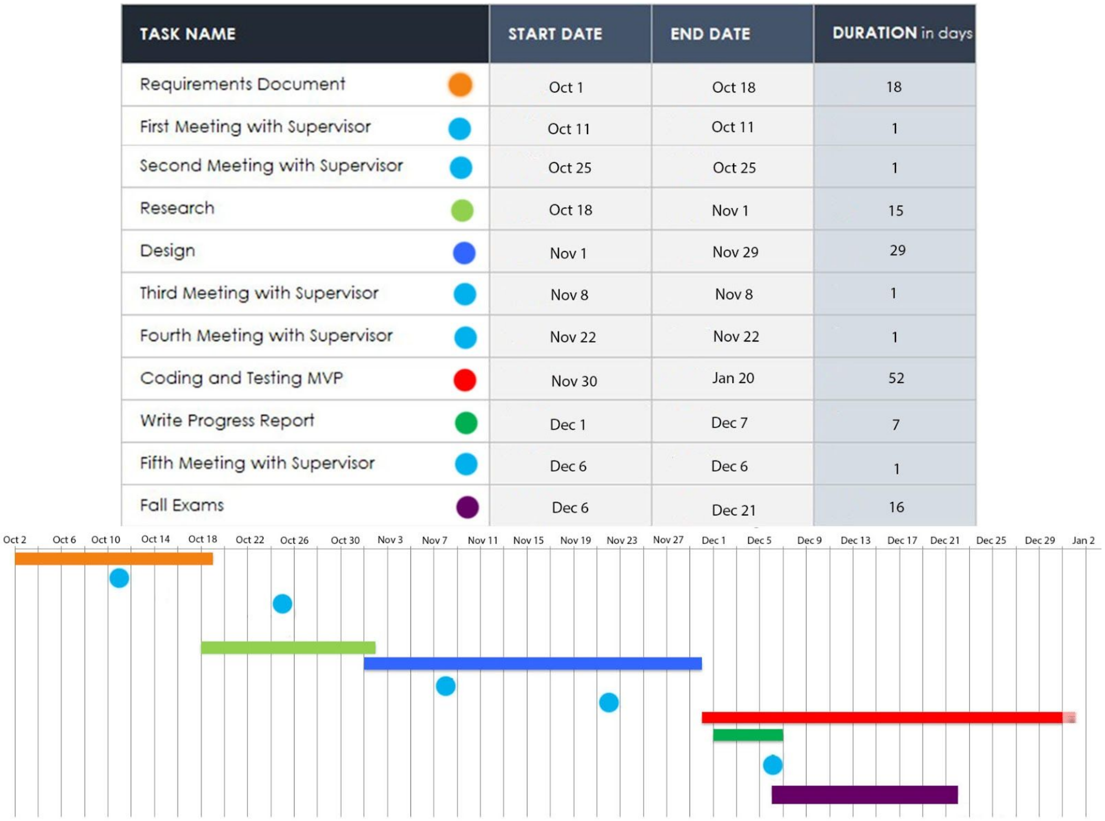
For dependency management, the best options include ant, maven and gradle. Ant is too primitive and isn’t as feature rich as maven and gradle. Maven is simple to use and is more familiar to the team than gradle, so it’s used instead. Jacoco was the only plugin used in the project. Jacoco is a free tool that measures branch coverage of our test suites using an all edges criterion. Jacoco generates a web page with our code highlighted depending on the coverage, making this process easy to visualize. The dependencies imported using maven are the JUnit test framework for creating our test suites and google’s gson library. Gson was used as a JSON parsing library over other parsers such as jackson because of its simple API and its ability to distinguish null and undefined values. Other libraries convert an empty value to null, but this behaviour isn’t always desirable when using the parser for a configuration file; sometimes an undefined value isn’t a mistake and shouldn’t throw an error.

Jetbrains’ IntelliJ IDEA IDE was used over other IDEs because of its maven lifecycle integration, java helper functions to make development easier, and great git integration. Git version control is important in applications to coordinate changes. Commits are pushed to a github repository where the project’s kanban board and branches reside. Github was used over other repositories because we’re more familiar with it and it also has a lot of project management features. If communication on github isn’t enough, we also communicated on slack for project meetings and non-code related discussion. Microsoft windows and unix based operating systems like Ubuntu linux and Apple’s OSX are used to confirm that the application works as intended on the most popular operating systems.

The project reports and presentation were written with Google Docs and Slides, since they offer concurrent editing online, and have their own version control that lets team members keep track of each other's progress. Technical diagrams were drawn using draw.io, which has since changed names to “diagrams.net”. This tool has a lot of built in support for UML diagrams, and also allows multiple members to edit the document concurrently.

5 Project Scheduling

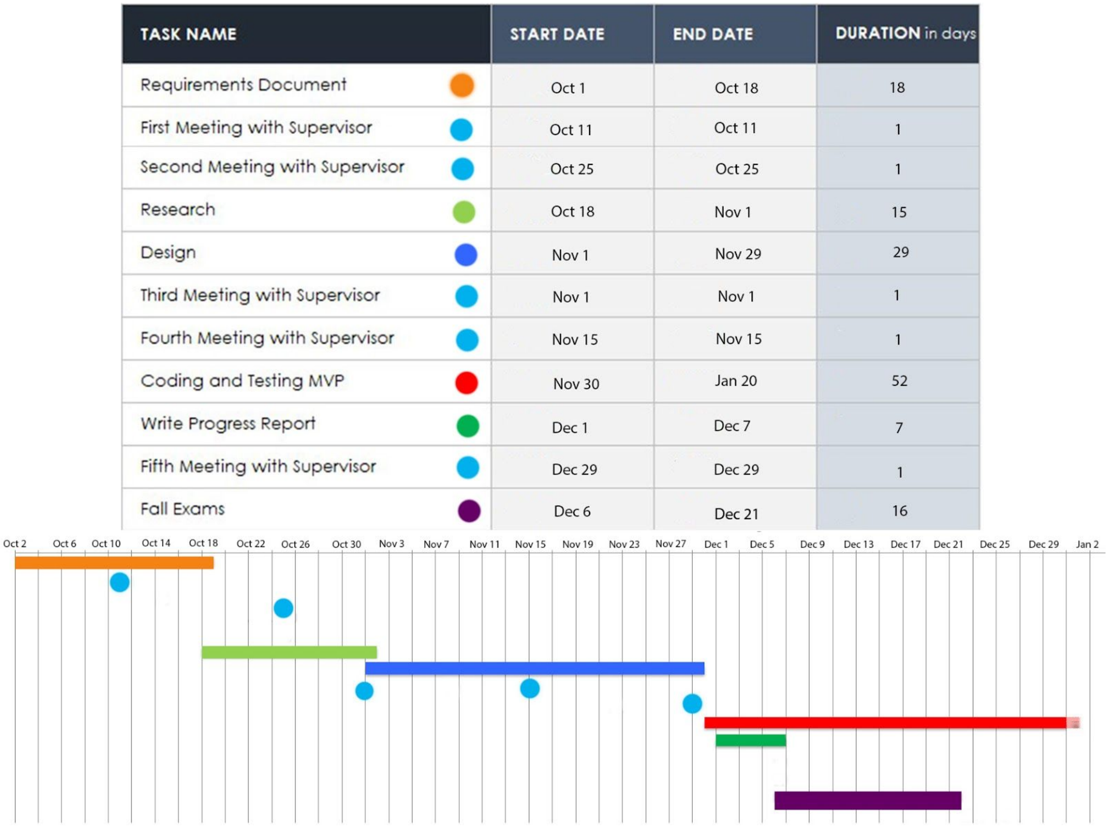
A strict schedule was maintained throughout development, covering two iterations. Our original schedule was followed closely with little change and included time for report writing, requirements elicitation, system design, programming and testing. The entire programming phase was divided into two goals. The first goal was to create the minimum viable product, the minimum amount of software required to satisfy the functional requirements. The final goal was to finish the application, satisfying all requirements. The only changes made to the fall schedule were the rescheduling of the biweekly meetings with the project supervisor due to reading week. Other meetings were shuffled to be sooner to keep the same number of expected meetings before and after key deliverables. The time after December first was kept free of meetings because of final examinations.



TASK NAME		START DATE	END DATE	DURATION in days
Requirements Document	●	Oct 1	Oct 18	18
First Meeting with Supervisor	●	Oct 11	Oct 11	1
Second Meeting with Supervisor	●	Oct 25	Oct 25	1
Research	●	Oct 18	Nov 1	15
Design	●	Nov 1	Nov 29	29
Third Meeting with Supervisor	●	Nov 8	Nov 8	1
Fourth Meeting with Supervisor	●	Nov 22	Nov 22	1
Coding and Testing MVP	●	Nov 30	Jan 20	52
Write Progress Report	●	Dec 1	Dec 7	7
Fifth Meeting with Supervisor	●	Dec 6	Dec 6	1
Fall Exams	●	Dec 6	Dec 21	16

Figure 2: Original fall timeline

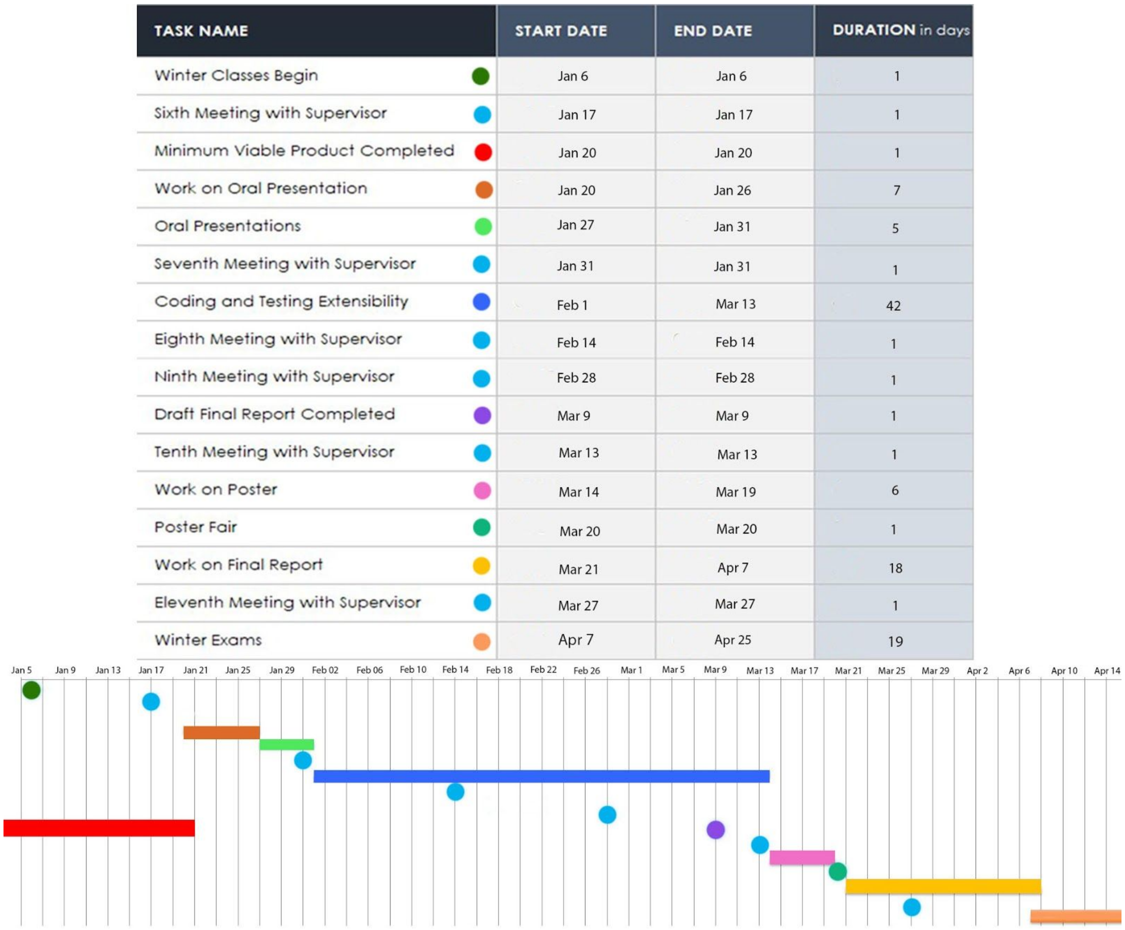
The figures above and below list colour coded important meetings, milestones and work schedules. Coding and testing the minimum viable product extended into the winter break when the team wasn't obligated to work on the software, but some work was done during this time.



TASK NAME		START DATE	END DATE	DURATION in days
Requirements Document	●	Oct 1	Oct 18	18
First Meeting with Supervisor	●	Oct 11	Oct 11	1
Second Meeting with Supervisor	●	Oct 25	Oct 25	1
Research	●	Oct 18	Nov 1	15
Design	●	Nov 1	Nov 29	29
Third Meeting with Supervisor	●	Nov 1	Nov 1	1
Fourth Meeting with Supervisor	●	Nov 15	Nov 15	1
Coding and Testing MVP	●	Nov 30	Jan 20	52
Write Progress Report	●	Dec 1	Dec 7	7
Fifth Meeting with Supervisor	●	Dec 29	Dec 29	1
Fall Exams	●	Dec 6	Dec 21	16

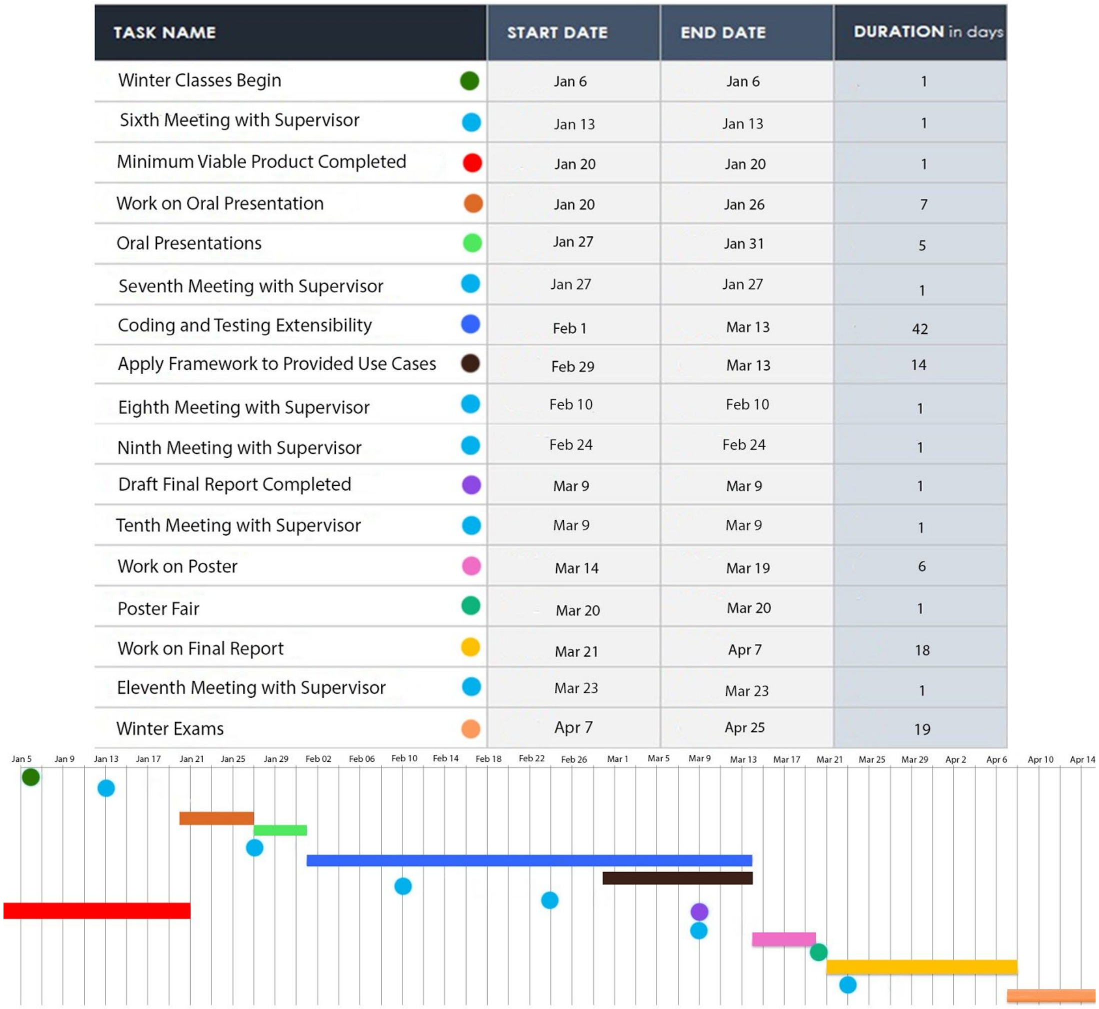
Figure 3: Revised fall timeline

By the end of the first half of development, the project was on time and the minimum viable product was nearly complete. Scheduling each phase beforehand so that development goes smoothly was beneficial and increased productivity. Very little changes were made to the winter timeline as well.



TASK NAME	START DATE	END DATE	DURATION in days
Winter Classes Begin ●	Jan 6	Jan 6	1
Sixth Meeting with Supervisor ●	Jan 17	Jan 17	1
Minimum Viable Product Completed ●	Jan 20	Jan 20	1
Work on Oral Presentation ●	Jan 20	Jan 26	7
Oral Presentations ●	Jan 27	Jan 31	5
Seventh Meeting with Supervisor ●	Jan 31	Jan 31	1
Coding and Testing Extensibility ●	Feb 1	Mar 13	42
Eighth Meeting with Supervisor ●	Feb 14	Feb 14	1
Ninth Meeting with Supervisor ●	Feb 28	Feb 28	1
Draft Final Report Completed ●	Mar 9	Mar 9	1
Tenth Meeting with Supervisor ●	Mar 13	Mar 13	1
Work on Poster ●	Mar 14	Mar 19	6
Poster Fair ●	Mar 20	Mar 20	1
Work on Final Report ●	Mar 21	Apr 7	18
Eleventh Meeting with Supervisor ●	Mar 27	Mar 27	1
Winter Exams ●	Apr 7	Apr 25	19

Figure 4: Original winter timeline



TASK NAME	START DATE	END DATE	DURATION in days
Winter Classes Begin ●	Jan 6	Jan 6	1
Sixth Meeting with Supervisor ●	Jan 13	Jan 13	1
Minimum Viable Product Completed ●	Jan 20	Jan 20	1
Work on Oral Presentation ●	Jan 20	Jan 26	7
Oral Presentations ●	Jan 27	Jan 31	5
Seventh Meeting with Supervisor ●	Jan 27	Jan 27	1
Coding and Testing Extensibility ●	Feb 1	Mar 13	42
Apply Framework to Provided Use Cases ●	Feb 29	Mar 13	14
Eighth Meeting with Supervisor ●	Feb 10	Feb 10	1
Ninth Meeting with Supervisor ●	Feb 24	Feb 24	1
Draft Final Report Completed ●	Mar 9	Mar 9	1
Tenth Meeting with Supervisor ●	Mar 9	Mar 9	1
Work on Poster ●	Mar 14	Mar 19	6
Poster Fair ●	Mar 20	Mar 20	1
Work on Final Report ●	Mar 21	Apr 7	18
Eleventh Meeting with Supervisor ●	Mar 23	Mar 23	1
Winter Exams ●	Apr 7	Apr 25	19

Figure 5: Revised winter timeline

Again, in the winter term the meeting dates with the project supervisor had to be changed due to lab scheduling conflicts, but they were rescheduled and never completely removed. After the seventh meeting with the supervisor, we realised we never explicitly gave time to apply the software to a set of provided use cases. Ample time near the end of development was given to try out the product so that we can still make adjustments and be on schedule. At the time of writing, all major development has been completed and only small quality of life changes remain, meaning we are slightly ahead of schedule.

6 Engineering Economics

The project is purely software, which cost nothing in terms of money to produce. All the technologies and tools used were free/open-source. The only cost for project development was time. Each team member spent on average six to ten hours a week on the project, including during reading weeks and over the winter break. Due to the project scheduling performed at the beginning of the project and in weekly team meetings, there were no instances where the amount of work needed drastically increased in one week compared to another. It is difficult to quantify the value added by the project. The team is not looking to market the product of the project; it can be downloaded from the public repository listed in the *User Manual* in the Appendix, which includes the source code.

The project may not explicitly add value with its intended purpose, but certainly saves time. The goal of this project is to automate the process of measuring test case diversity, which to do manually is incredibly time consuming. The project is particularly useful if it already supports a user's test case formats and contains their required diversity metric, but can be extended quite easily. The *User Manual* describes how exactly this can be done, but support for new diversity metrics can be implemented very quickly with no changes required to the rest of the system. Making the system support new test case formats may take a little longer, but the time this takes mostly depends on how complicated the format is, since the main task to support a new format is to write a parser for it.

Without a tool like this, the calculations would likely require the use of a spreadsheet for anything but the smallest test suites, but this has a major disadvantage. The act of performing a diversity calculation can be broken into three parts: converting a test suite into a form that allows test cases to easily be compared, performing the comparison, and then taking those comparison results and putting them into some meaningful form. A spreadsheet can help to organize test case data and somewhat automate comparisons once the comparison formula has been set up, but the test cases still need to be put into a form usable by a spreadsheet, and this must be done for every test suite that a calculation is to be performed on. The use of this tool requires only writing a script to read test cases of a certain form once, and can then be used for any number of test suites. The use of a programming tool also allows for more complicated comparison metrics than what could easily be written in a spreadsheet.

7 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.

7.1 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 may 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 one 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.

7.2 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.

7.3 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.

7.4 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.

7.5 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.

7.6 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.

8 System Design

8.1 Use Cases

Once we determined requirements, we developed four use cases. These are: performing a diversity calculation, configuring the system, asking the system for help, and extending the system's functionality. Each of these use cases are described here.

Performing a diversity calculation is the primary use for the system, because the automation of this process is the motivation behind creating the system in the first place. This can be further broken down into sub-cases for pairwise and listwise metrics, as these behave slightly differently. In these cases, the user should supply the test suite to use in the calculation, the representation of those test cases so the system knows how to parse the files, the diversity metric to use, the method to aggregate the results of that diversity metric, and the format to report results.

Configuring the system will allow the system to store values for parameters that would be unchanged over several diversity calculations. In diversity calculations, there are a lot of parameters to specify, so it made sense to allow the user to give default values to these to reduce the length of system commands. This means less typing for the user when entering commands, but also less room for error in how the user writes a command.

While there is a user manual that describes in detail each of the different possible values for parameters, it would be nice to be able to quickly consult the system to see what those possible values are, and have a shortened description to avoid having to consult a separate document for everything. The help use case is split into several sub cases, each for different types of listings that may be useful. These sub cases are listing the available system commands, supported test cases representations, diversity metrics, aggregation methods, and reporting formats.

The final use case, extending the system functionalities, is a little different from the rest. Functionalities that can be added are the test case representations, diversity metrics, aggregation methods, and reporting formats. This will be a case of writing extra code that is then added to the system, and so this use case is something that pertains to the system, but is largely done outside of the system itself.

The diagram below is a use case diagram which summarizes the discussion here.

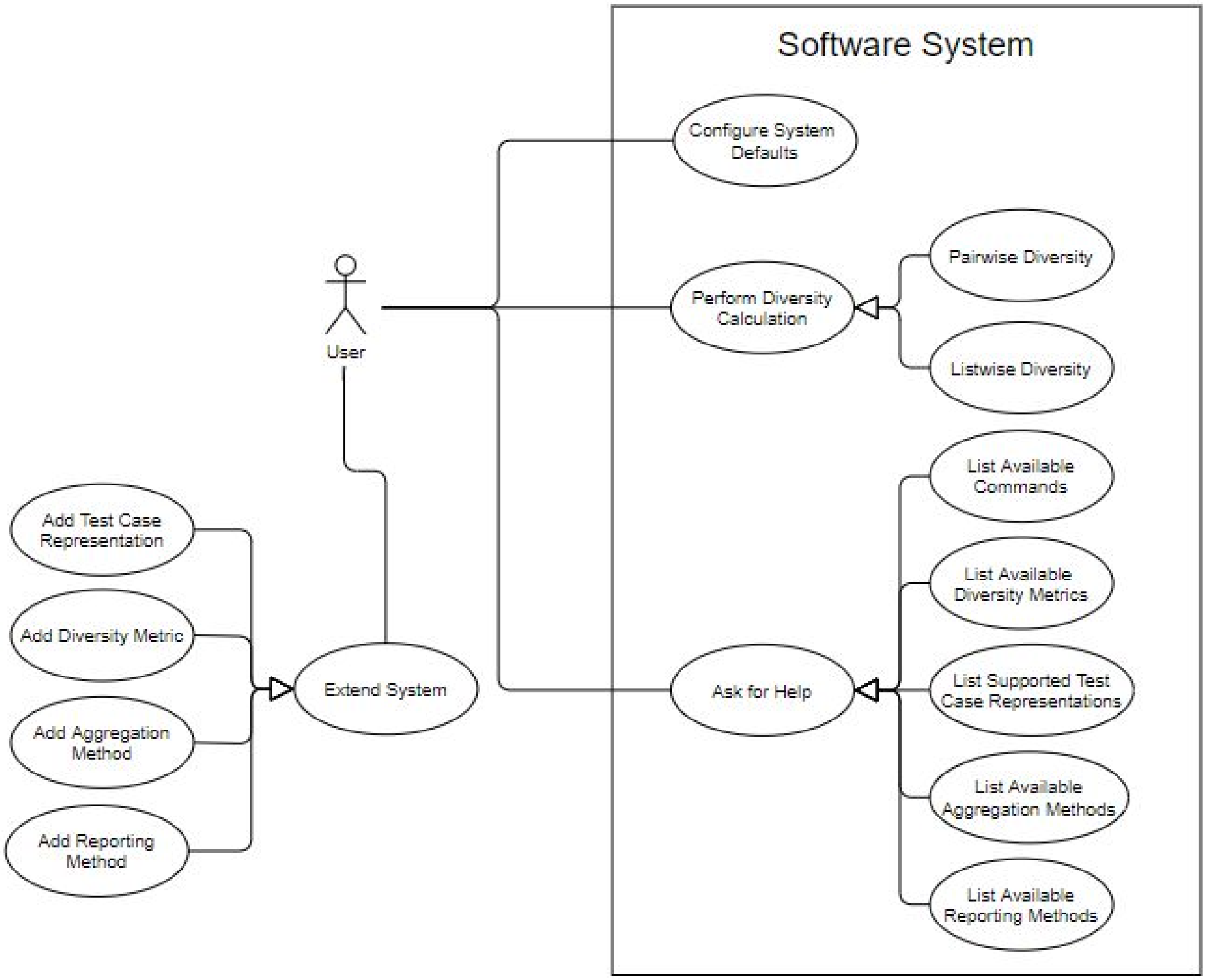

Software System
Configure System Defaults
User
Perform Diversity Calculation
Pairwise Diversity
Listwise Diversity
List Available Commands
List Available Diversity Metrics
List Supported Test Case Representations
List Available Aggregation Methods
List Available Reporting Methods
Ask for Help
Add Test Case Representation
Add Diversity Metric
Add Aggregation Method
Add Reporting Method
Extend System

Figure 6: Use Case Diagram for the system.

8.2 High Level Description

At a high level, the system can be described by the following diagram:

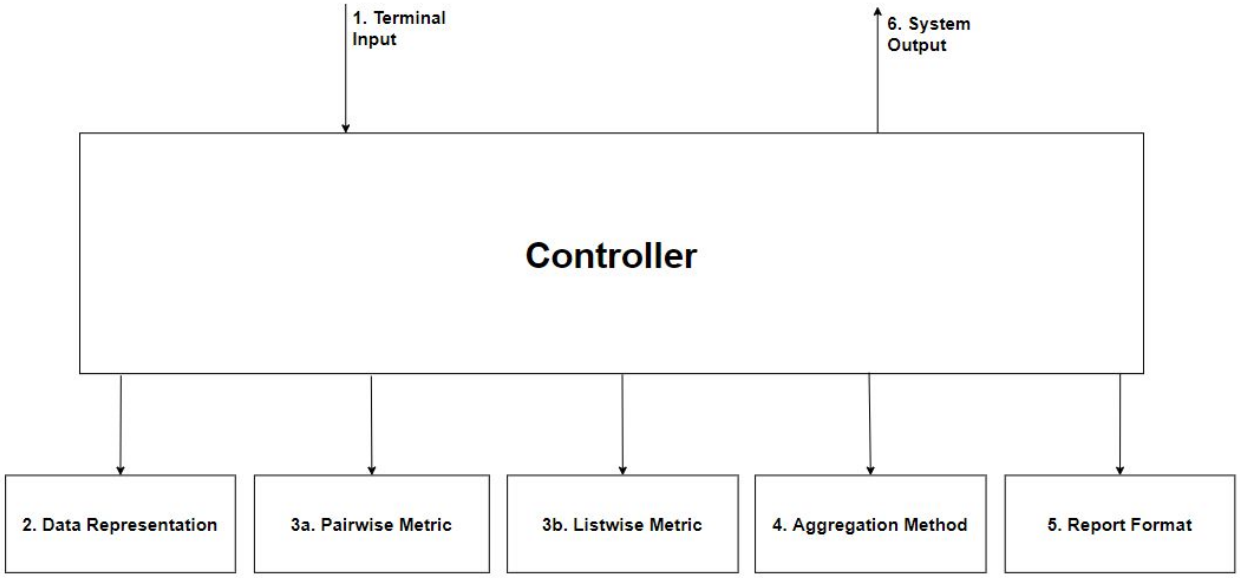


Figure 7: A high level depiction of the system.

This also shows the primary use case of the system, performing a diversity calculation. First, the user supplies input to the system as command line arguments; this input specifies the test suite file(s) to use in the diversity calculation, the data representation to use that specifies how to read the provided test cases, the diversity metric to use, the method(s) used to aggregate the diversity metric results, and the format in which to display the aggregated results. The system’s controller uses these one by one to perform the calculation. The data representation is used to read in the test cases, which are then compared according to either a pairwise or listwise metric. The results of the diversity metric are then aggregated in one or more ways, and those aggregated results are formatted according to the specified report format. Finally, the formatted results are output to the terminal, and optionally to a file if specified in the users command.

8.3 Overall Architecture

Originally, the system was designed with a Model-View-Controller[2] (MVC) approach. In this design, the model was for the test cases, the view was to be a simple text-based read-eval-print-loop[3] (REPL), and the controller would connect the model and view. However, to achieve the extensibility requirements and make greater reuse of existing code, the controller was refactored into several objects, each of which performs a few well defined functions similar to each other but different from the functionality offered by other control objects. This allowed separate elements of the controller to be decoupled from each other, and connected via a central Controller class. Pairing this change with the simplification of the view from a REPL to just command line arguments, and the resulting architecture ended up resembling a Service Oriented Architecture[4] (SOA), where control objects are services, and the central Controller class acts as a service bus. An SOA approach was also beneficial for development, because it allowed for more clear boundaries between components so work could be assigned to each team member more easily.

[2] A software architecture for interactive systems that separates the data, user interface, and business logic from each other.

[3] A type of execution flow where instructions are received one at a time from the user. Each instruction is processed and completed before receiving the next instruction.

[4] A modular software architecture which separates each distinct function into its own component, which are networked together to create the application.

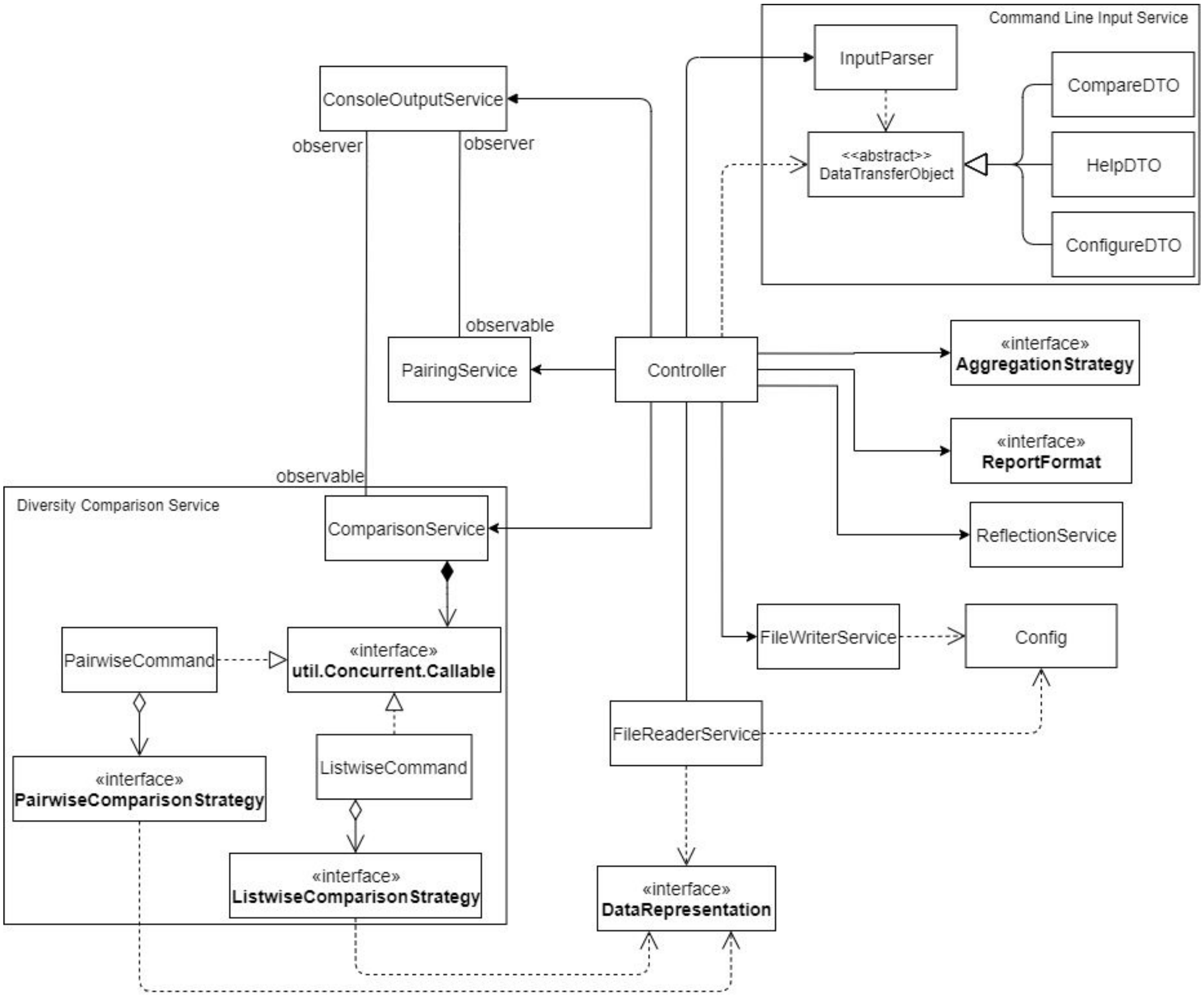


Figure 8: Diagram of the overall system, separated into its services.

There are still some elements of MVC in the system. For example, the ConsoleOutputService, which was originally the system view, observes elements of the controller, namely the PairingService and ComparisonService. The observer relationship here however is now used to display the progress of operations to the user, as they are the most computationally heavy parts of the system, and is discussed more in later sections.

8.4 Responsibilities

The figure above introduces many more components to the description of the system. The following sections provide a brief description of each service, and class, to summarize their responsibility in the system.

8.4.1 Service Responsibilities

Command Line Input Service: This component transforms input data from the user into a form usable by the rest of the system. By making this modular, it would be possible to use the system in some other environment (eg. as a web service) and only need to modify this component for the rest of the system to work.

Console Output Service: This component handles text based output to the terminal the program runs in. It displays progress bars as the system executes and outputs the final result.

Diversity Comparison Service: This component performs the actual calculation of diversity metric values.

File Reader Service: This component provides file reading capabilities for the system. This includes reading configuration information and reading the files containing test cases into the system.

File Writer Service: This component performs file output, including updates to the system configuration and writing comparison results to a file if necessary.

Pairing Service: This component generates pairs from a provided list of test cases, which is required for performing pairwise comparisons.

Reflection Service: This component enables our extensibility requirements. By use of reflection, this component searches the system files for the required diversity metrics, aggregation methods, etc. so that they do not need to hard coded in a list anywhere to be referenced.

8.4.2 Class Responsibilities

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

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

ComparisonService: Performs the actual comparison of test cases.

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

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

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

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

DataTransferObject: The abstract class extended by system command representations.

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

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

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

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

InputParser: Converts command line input into a form usable by the system.

ListwiseCommand: A container for a ListwiseComparisonStrategy.

ListwiseComparisonStrategy: The interface to be used by 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 used by pairwise diversity metrics.

ReflectionService: Searches the system files for the required diversity metrics, aggregation strategies, report formats, and data representations.

ReportFormat: The interface to be used by all output formats.

8.5 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.

Unfortunately, this means that the Controller object is more tightly coupled to other objects in the system, but this is necessary for its function of coordination and error handling, and decouples all the other service objects from each other. The Controller itself does not do much more than taking the results of one adjacent object and passing them as parameters to the next, so the risk of this object being the single point of failure is fairly low due to its simplicity.

Any errors that could occur are more likely to happen in the other controller objects, which do more complex tasks.

A similar functionality could be achieved through a pipe and filter architecture, but this approach would miss the flexibility and code reuse that is gained through the method actually used. By organizing the system as is done above, each system command can be performed using the same set of controller objects, organized by the Controller in different ways, and those other objects the Controller coordinates do not need to know anything about which type of command the overall system is executing. How exactly the Controller orders calls to adjacent objects for the existing commands is elaborated in the following sections.

8.5.1 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.

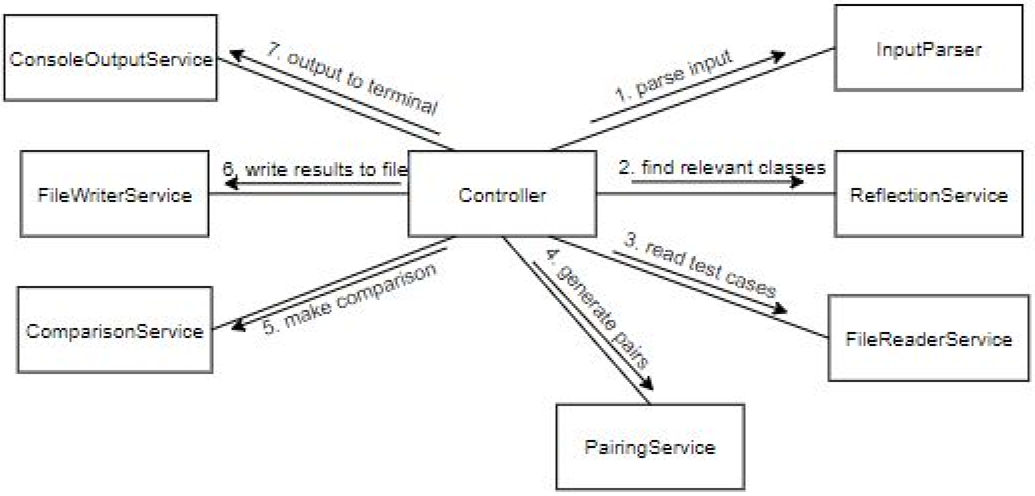


Figure 9: Collaboration diagram of pairwise comparisons in the system.

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, aggregation method, and report format. 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.

8.5.2 Listwise Comparisons

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

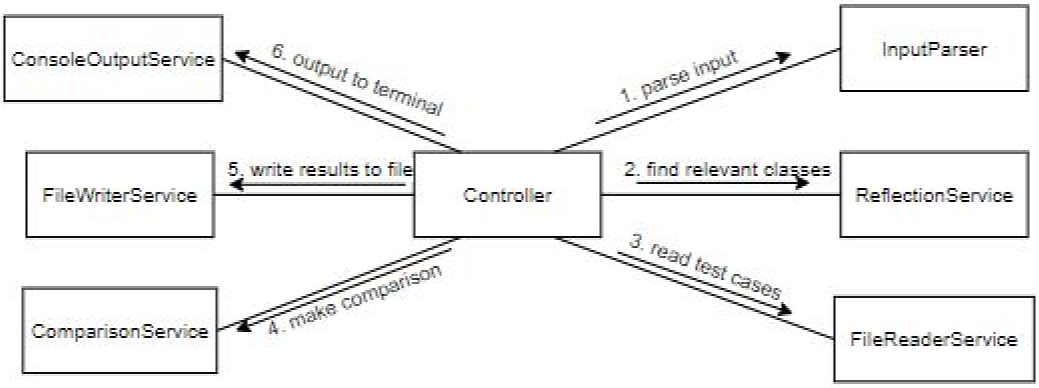


Figure 10: Collaboration diagram of listwise comparisons in the system.

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, except the test cases read into the system by the FileReaderService are immediately passed to the ComparisonService.

8.5.3 Configuration Commands

Another use case for the system is the ability to specify default values for the parameters of a comparison, so that they would not need to be explicitly typed with every command. The parameter defaults are stored in a JSON file. While it is possible to edit this directly, the system allows for safe configuration of the system defaults through the “configure” command; exactly how this is done is detailed below.

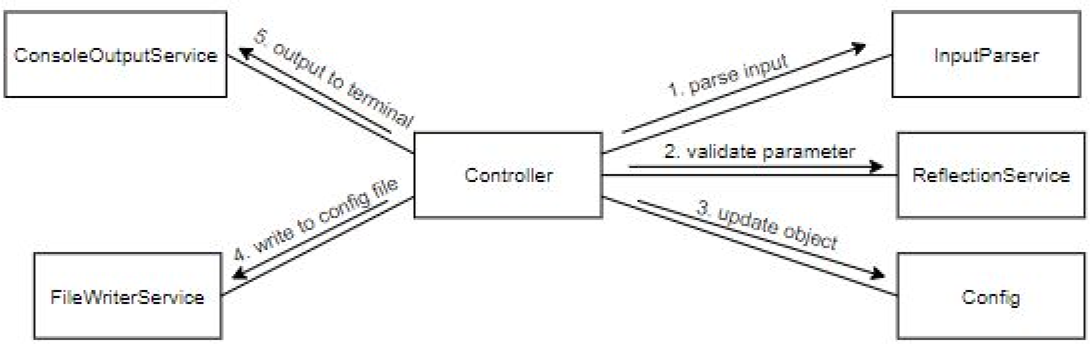


Figure 11: Collaboration diagram of configure commands in the system.

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.

8.5.4 Help Commands

Help commands offer the ability to gain information about the system's capabilities, without needing to open up the user manual.

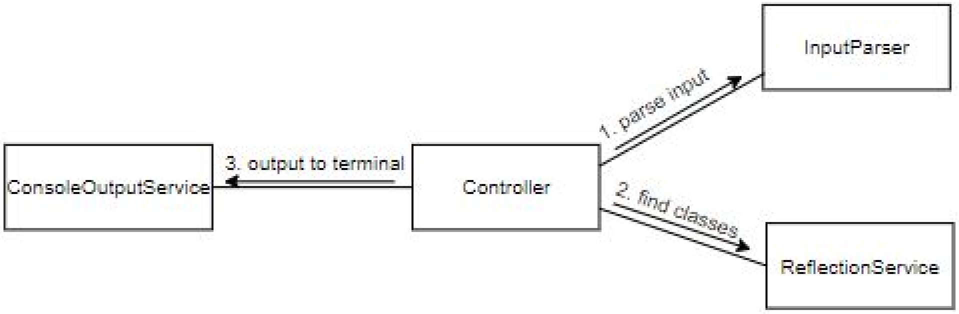


Figure 12: Collaboration diagram of help commands in the system.

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.

8.6 Design Patterns

Usage of design patterns has heavily influenced the overall system design. These patterns are 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.

Before going into each design pattern in detail, a short summary of each pattern used and why it is used is provided here:

Bridge Pattern: The bridge relationship between data representations and diversity metrics allows us to satisfy the need to compare any type of test case with any diversity metric.

Command Pattern: This pattern is used to encapsulate comparison commands so that they can be used in a thread pool for concurrent execution as per our performance requirements. This also makes it easier to substitute the function by which comparisons are made if needed in the future.

Data Transfer Object: This pattern is used to encapsulate the information from user commands into a generic form usable by the rest of the system. This means that the method by which the system receives user commands can be changed more easily if necessary.

Factory Method: A Factory Method is used to instantiate all the diversity metrics, data representations, etc. in an extensible way to satisfy the maintainability requirements.

Observer Pattern: An observer pattern is used to keep track of the progress of large operations in the system to meet usability requirements. With this relationship between the console output and services that are most computationally heavy, the user is provided with progress bars to see their command's progress.

Strategy Pattern: This pattern is used for the diversity metrics, aggregation methods, data representations, and report formats to swap out each based on the user command, as per the functional requirements.

The interface for the data representations also resembles an iterator pattern interface to make the implementation more intuitive, but an actual iterator pattern is not used here. This fact, along with the other patterns, is elaborated on below.

8.6.1 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.

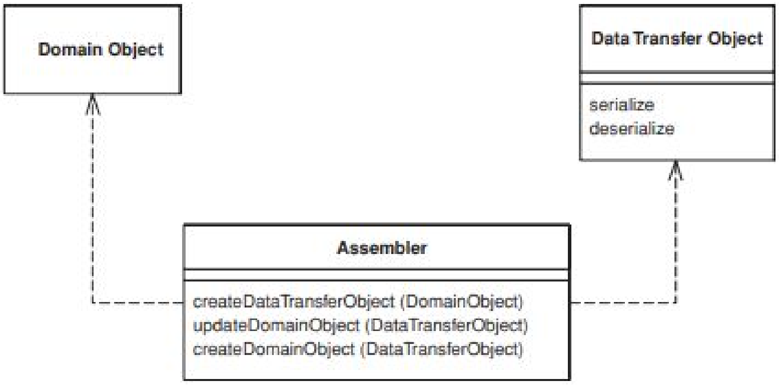

Domain Object
Data Transfer Object
serialize
deserialize
Assembler
createDataTransferObject (DomainObject)
updateDomainObject (DataTransferObject)
createDomainObject (DataTransferObject)

Figure 13: The general design of a data transfer object pattern. [5]

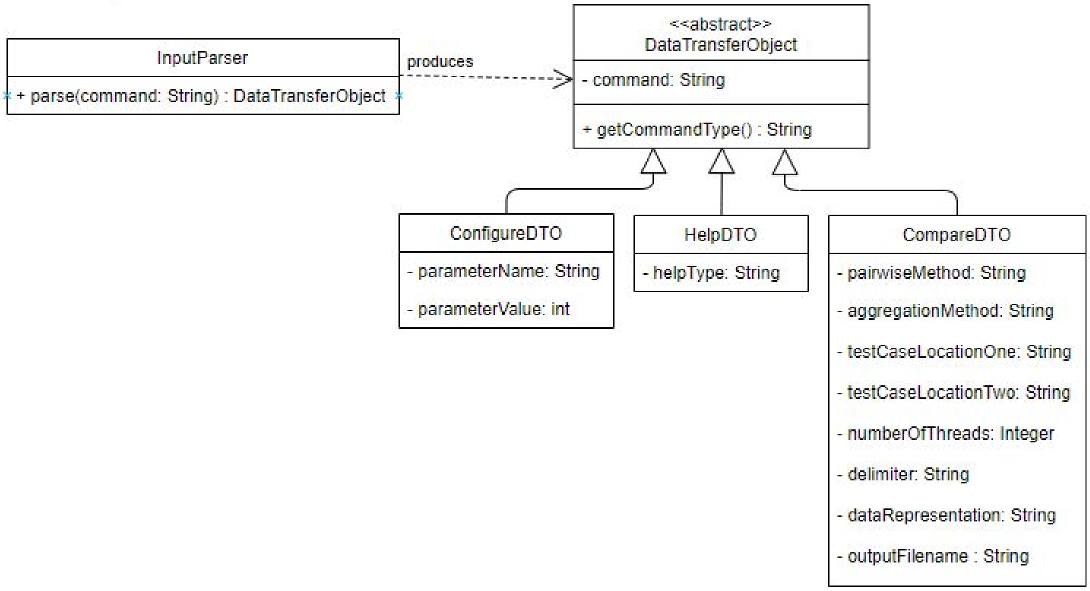

InputParser
+ parse(command: String) : DataTransferObject
produces
<<abstract>>
DataTransferObject
- command: String
+ getCommandType() : String
ConfigureDTO
- parameterName: String
- parameterValue: int
HelpDTO
- helpType: String
CompareDTO
- pairwiseMethod: String
- aggregationMethod: String
- testCaseLocationOne: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

Figure 14: 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.

8.6.2 Instantiating Classes with a Factory Method

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

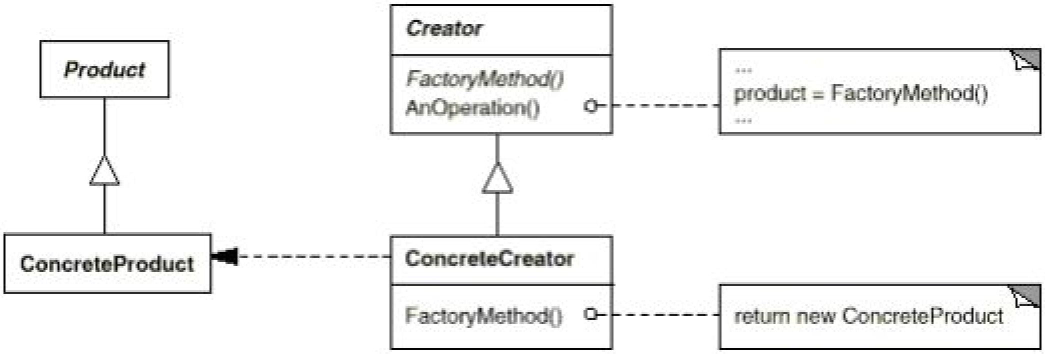


Figure 15: The general Factory Method design. [6]

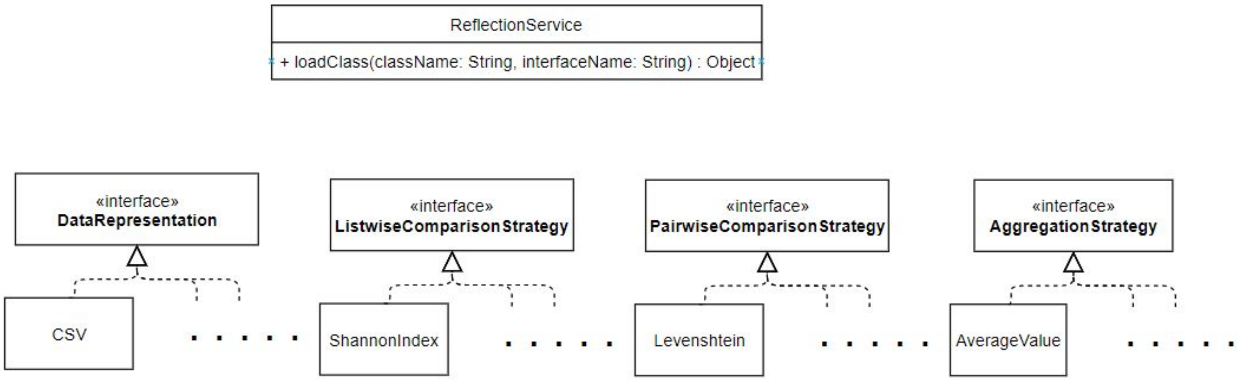


Figure 16: The use of Factory Method in the system.

Each of the extensible 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 Products, instead of requiring a separate factory method for each. This also means the factory method could be used to instantiate other types of Products if the project were continued in the future.

8.6.3 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 are all instantiated through the Factory Method discussed above. The exhaustive list of all the ConcreteStrategies for each Strategy in the system are or what each does is not particularly important to the design discussion, but they can be seen in the *User Manual* in the Appendix.

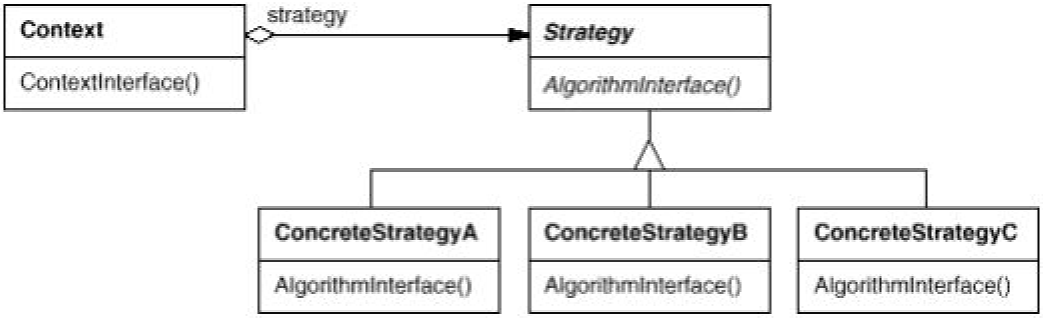


Figure 17: The general Strategy Pattern design. [7]

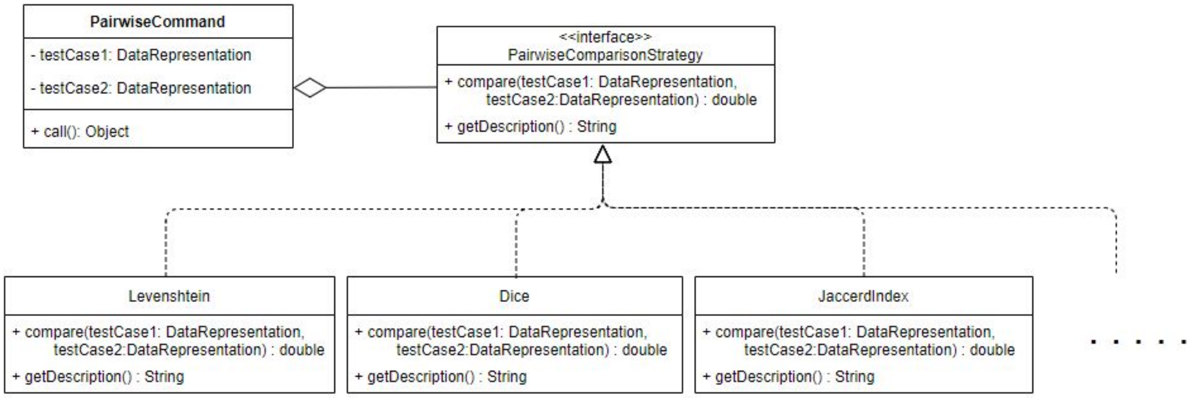

PairwiseCommand
- testCase1: DataRepresentation
- testCase2: DataRepresentation
+ call(): Object
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
+ getDescription() : String
Levenshtein
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation)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

Figure 18: Strategy usage for pairwise comparisons in the system.

The usage of the strategy pattern in pairwise comparisons (above) and listwise comparisons (below) strongly follow the general design for the pattern. For pairwise diversity metrics, the 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.

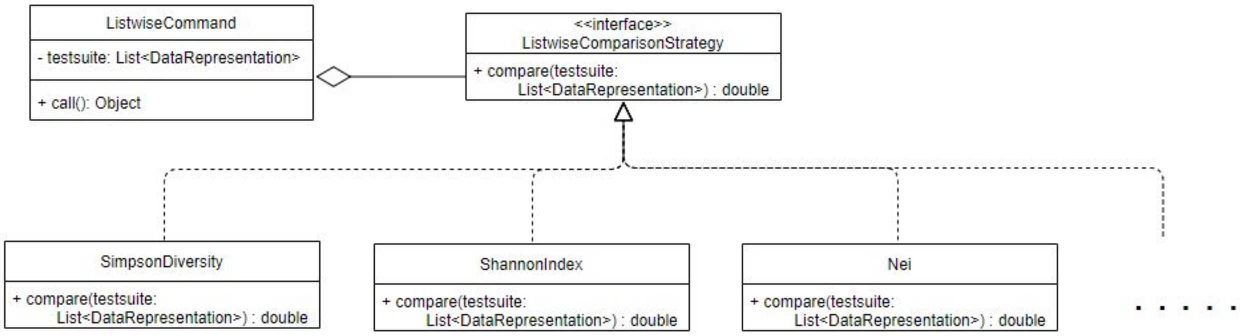

ListwiseCommand
- testsuite: List<DataRepresentation>
+ call(): Object
<<interface>>
ListwiseComparisonStrategy
+ compare(testsuite: List<DataRepresentation>) : double
SimpsonDiversity
+ compare(testsuite: List<DataRepresentation>) : double
ShannonIndex
+ compare(testsuite: List<DataRepresentation>)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

Figure 19: Strategy usage for listwise comparisons in the system.

The aggregation methods are implemented in a similar way. Once the ComparisonService completes its calculation using a 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.

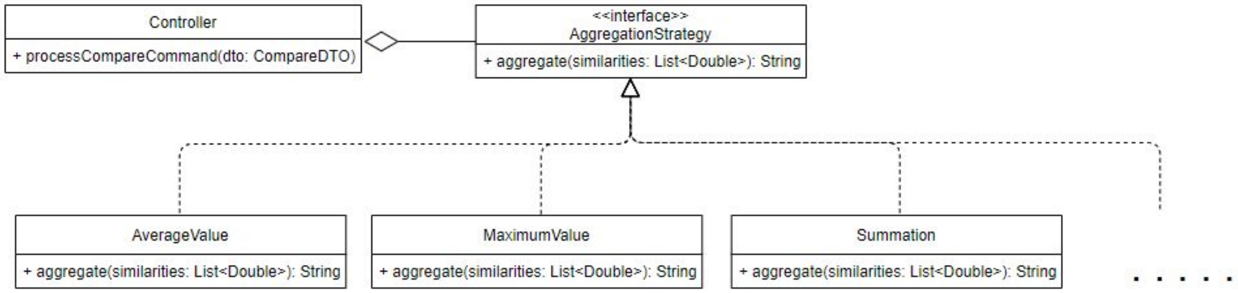

Controller
+ processCompareCommand(dto: CompareDTO)
<<interface>>
AggregationStrategy
+ aggregate(similarities: List<Double>): String
AverageValue
+ aggregate(similarities: List<Double>): String
MaximumValue
+ aggregate(similarities: List<Double>): String
Summation
+ aggregate(similarities: List<Double>):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

Figure 20: Strategy usage for aggregation methods in the system.

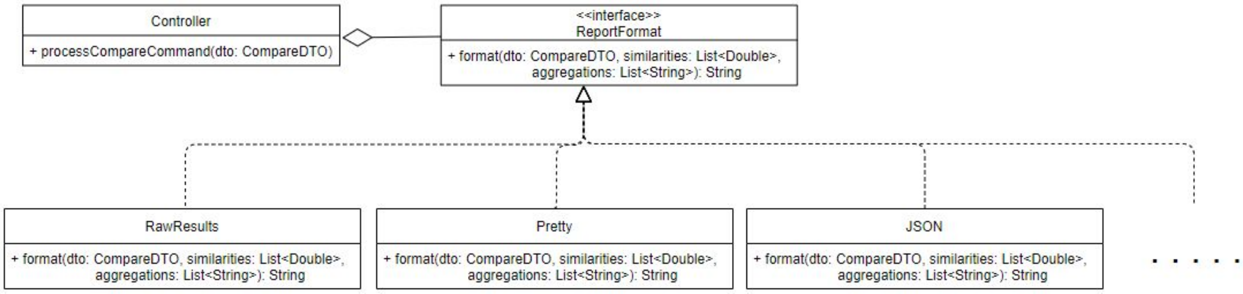

Controller
+ processCompareCommand(dto: CompareDTO)
<<interface>>
ReportFormat
+ format(dto: CompareDTO, similarities: List<Double>, aggregations: List<String>): String
RawResults
+ format(dto: CompareDTO, similarities: List<Double>, aggregations: List<String>): String
Pretty
+ format(dto: CompareDTO, similarities: List<Double>, aggregations: List<String>): String
JSON
+ format(dto: CompareDTO, similarities: List<Double>, aggregations: List<String>): String
. . . . .

Figure 21: Strategy usage for aggregation methods in the system.

Finally, there is the ReportFormats. Once the results of a comparison have been aggregated, they are passed to a ReportFormat implementation. The selection of ReportFormat to use specifies what information to include in the system output, and how the output should be structured. For example, RawResults simply outputs the aggregation values, while the implementations Pretty and JSON also include information about which diversity metric and aggregation methods were used (from the provided data transfer object). Pretty simply outputs text, while JSON output is formatted as the name suggests. Here, the Controller is the Context, ReportFormat is the Strategy, format is the AlgorithmInterface, and the implementations RawResults, Pretty, JSON, etc. are the ConcreteStrategies.

8.6.4 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, helps to decouple, and allows greater extensibility of, the diversity metrics and data representations [8].

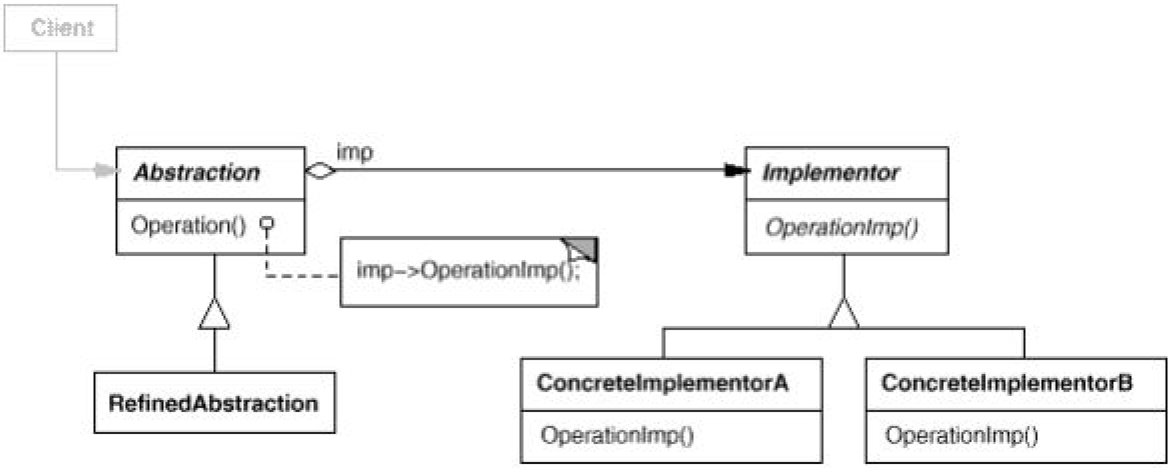


Figure 22: The general structure of a Bridge Pattern. [8]

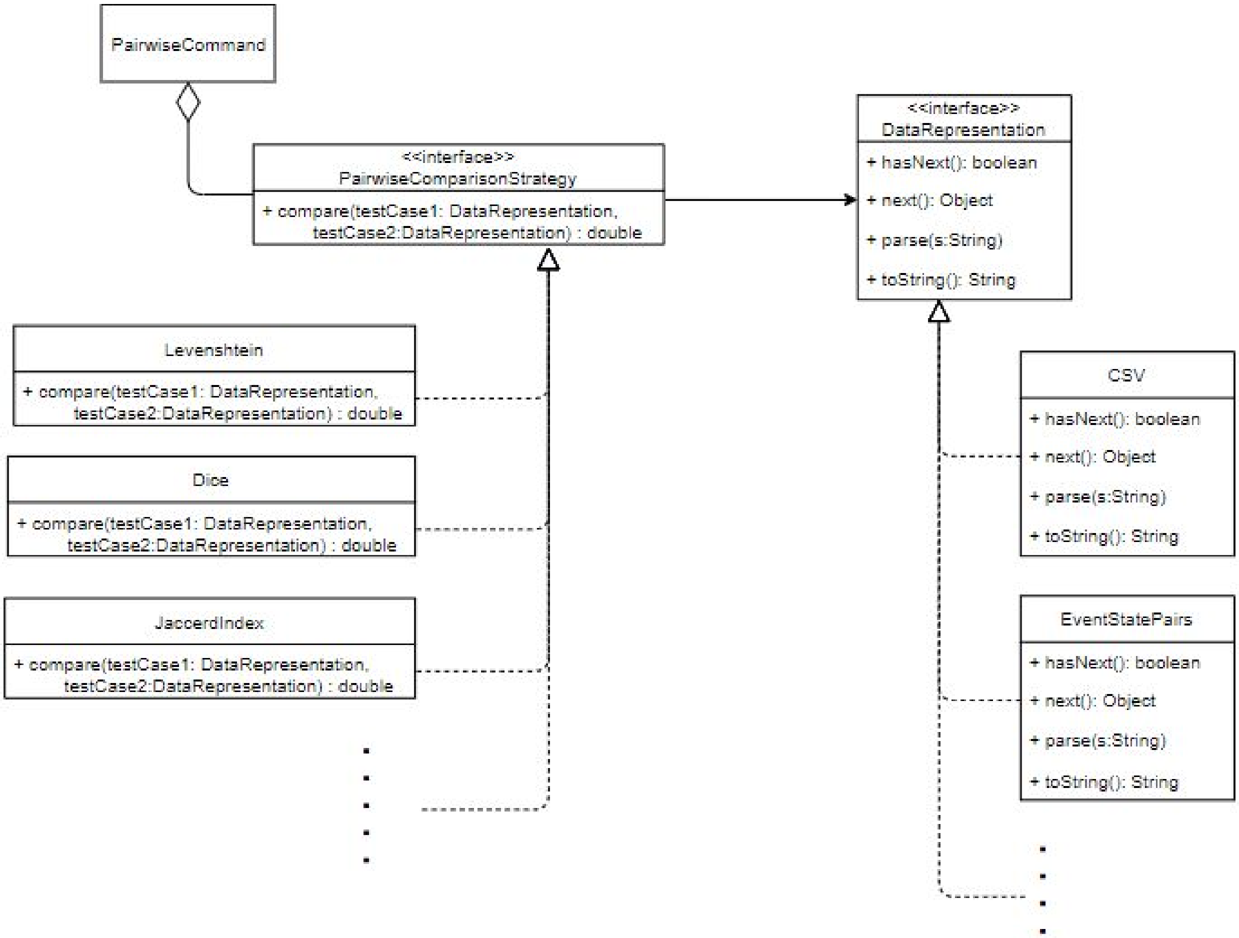

PairwiseCommand
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : double
<<interface>>
DataRepresentation
+ hasNext(): boolean
+ next(): Object
+ parse(s:String)
+ toString(): String
Levenshtein
+ compare(testCase1: DataRepresentation, testCase2:DataRepresentation) : 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 23: 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.

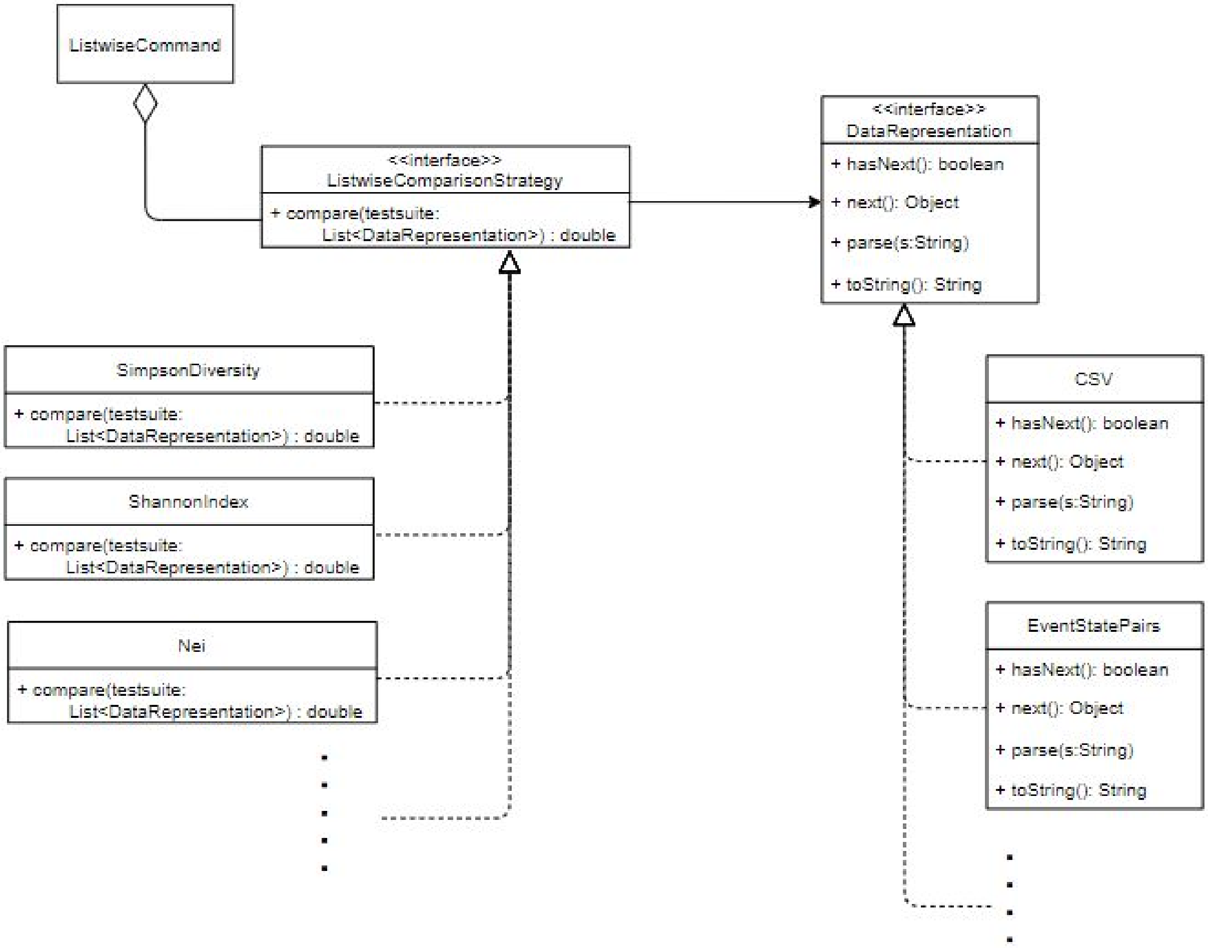


Figure 24: Use of Bridge Pattern between listwise metrics and data representations.

8.6.5 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().

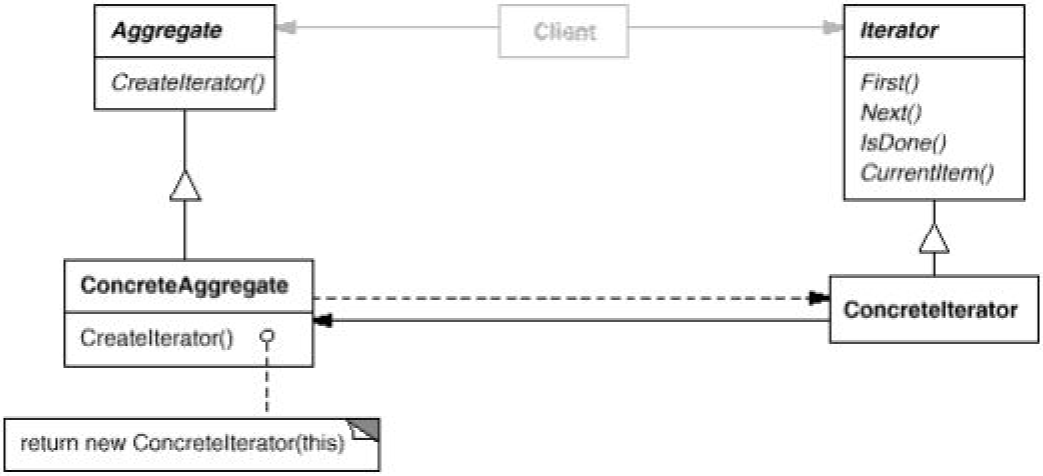


Figure 25: The general design of an Iterator Pattern. [9]

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 a test case in a private field in the DataRepresentation, there is also much more freedom in how a 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.

8.6.6 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 needed to make comparisons grows rapidly. The number of pairs generated from a test suite 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 in a test suite 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]

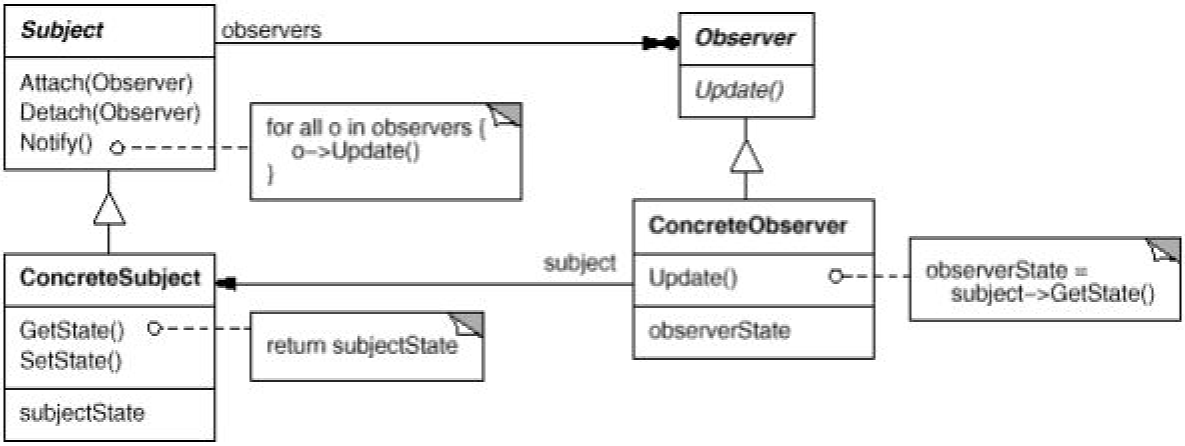


Figure 26: The general design of an Observer Pattern. [13]

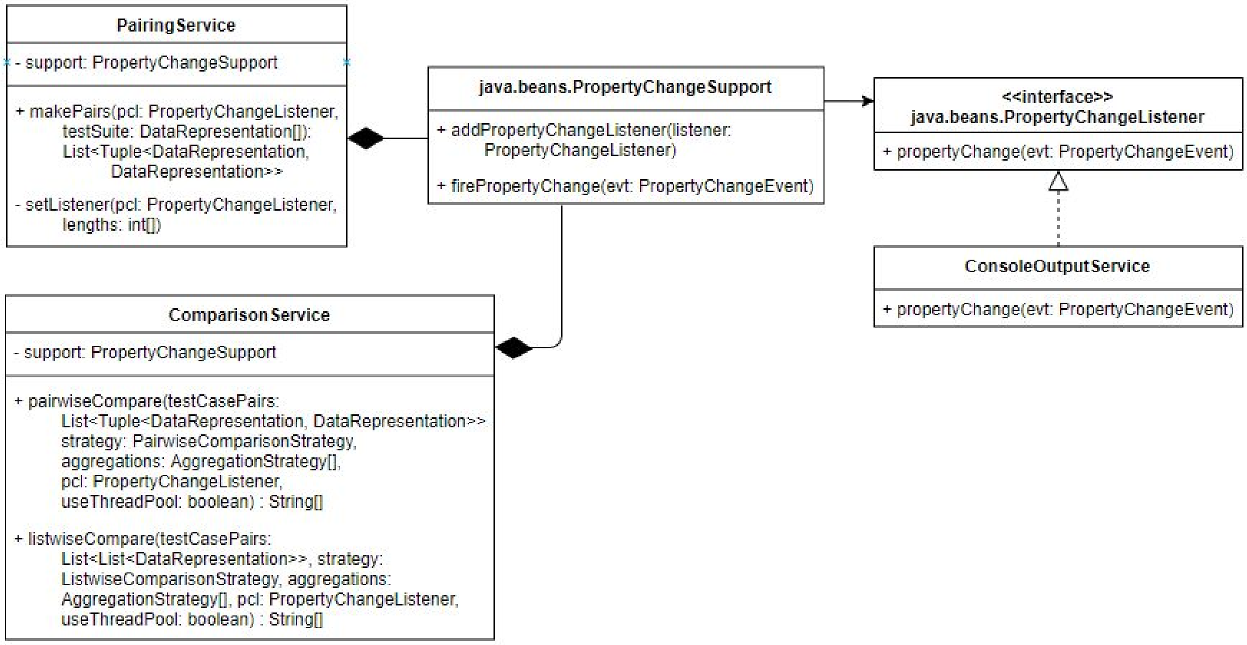

PairingService
- support: PropertyChangeSupport
+ makePairs(pcl: PropertyChangeListener, testSuite: DataRepresentation[]): List<Tuple<DataRepresentation, DataRepresentation>>
- setListener(pcl: PropertyChangeListener, lengths: int[])
java.beans.PropertyChangeSupport
+ addPropertyChangeListener(listener: 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 27: 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 ConsoleOutputService 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 as 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.

8.6.7 Command Pattern to Execute Comparisons in a Thread Pool

In order to improve the performance of a large number of operations, we decided to implement a thread pool. The use of a thread pool requires encapsulation of parts of that large number of operations in objects that implement a similar interface, and so the 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.

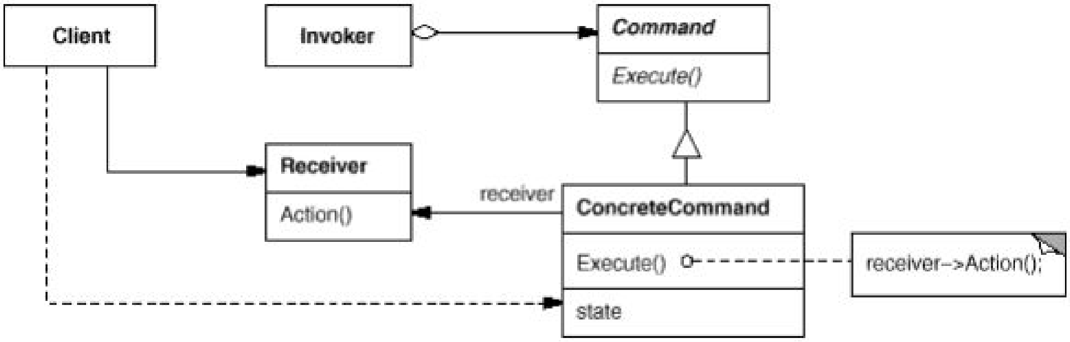


Figure 28: The general design of a Command Pattern. [14]

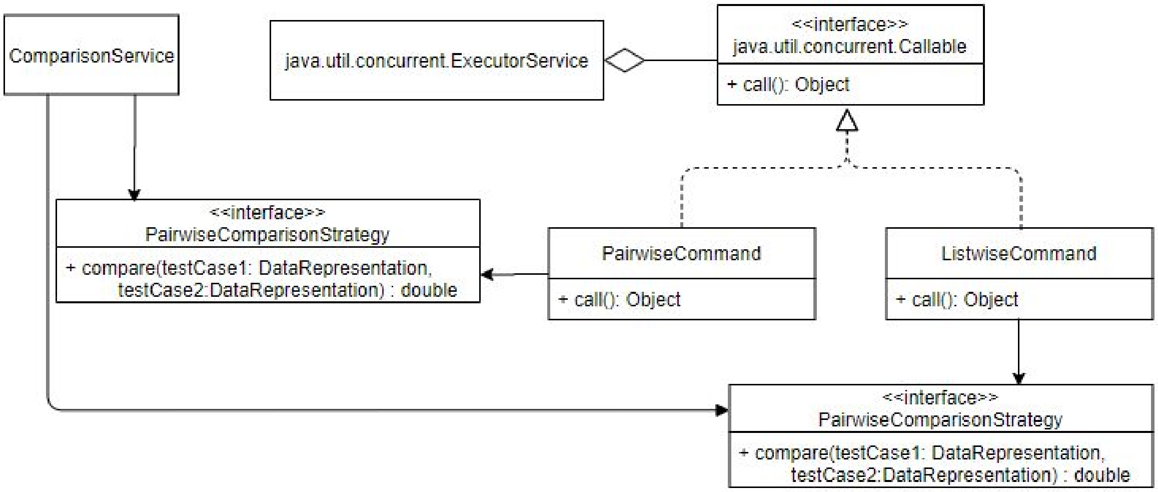

ComparisonService
java.util.concurrent.ExecutorService
<<interface>>
java.util.concurrent.Callable
+ call(): Object
<<interface>>
PairwiseComparisonStrategy
+ compare(testCase1: DataRepresentation,
testCase2:DataRepresentation) : double
PairwiseCommand
+ call(): Object
ListwiseCommand
+ call(): Object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

Figure 29: The implementation of Command Pattern for comparisons in the system.

When the comparison service is used, it generates several PairwiseCommands or ListwiseCommands that reference the diversity metric to use. Each of these command objects are invoked by the thread pool to generate comparison results. From figures 28 and 29, 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.

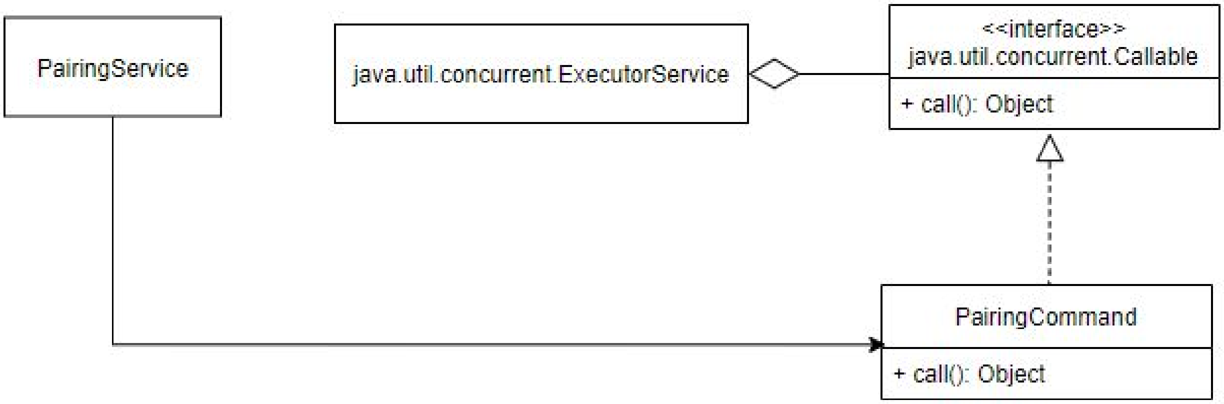


Figure 30: The implementation of Command Pattern for pairing test cases in the system.

9 Software Testing

Since the system was implemented from the bottom up, the system was also tested in the same way. What this means is that classes which do not depend on other untested classes are tested first. Once all the classes without dependencies had been tested, classes which depended on these classes were tested. The process is then repeated until every class has been tested. In one testing sequence, a first data representation object was tested, followed by a diversity metric, then an encapsulating command object, and then the ComparisonService. At the same time, the Conifg model object could be tested, and then the FileWriterService and FileReaderService.

This method of testing allowed us to test the system as it was being built, alerting us to problems that might have affected later classes. It also allows us to avoid the issue of stubbing classes, so time was saved by not needing to write extra code for stubs. Due to the architectural approach that was taken (see figure 8 in section 8.3), there were many classes with few dependencies, and so many classes could immediately be tested in parallel with each other.

The tests were created using a black box technique by creating tests without looking at the code past the method headers. This meant that the tester would test what the method should be doing instead of trying to test every line of code. This was made simpler by assigning the creation of tests to a developer who had not worked on the code being tested, preventing familiarity with the code hiding an issue. Category Partition was the technique used to generate these tests, and in several of the larger operations, testing required the use of the base block criterion, since for those methods that can cause exceptions, it is easiest to test one type of exception at a time. Once tests were generated in this way, a white-box criterion was used to evaluate the coverage of those tests, and more tests would be created based on that coverage if necessary.

The goal of our test suite was to be all-edges adequate. The coverage of our tests, that is the code tested by our tests, was verified by using Jacoco. This allowed us to ensure the test suite was in fact all-edges adequate and showed which area of the code was missing tests. The All-edges coverage criterion is the most demanding criterion of free test coverage tools we could find, and a more detailed test coverage application would have been more expensive.

Our final test suite achieves full coverage (ie. 100% edge coverage) everywhere except the for Controller and the ReflectionService classes. The test suite achieves 87% for the Controller class, and 90% edge coverage for the ReflectionService class, due primarily caused by certain exception catching, and due to some code that only executes when running the program from the jar file, which is elaborated later in the *Challenges* section of this report. The untested exceptions are either caused by conditions that are difficult to replicate or can be thrown by Java methods, but cannot logically occur.

An example of difficult to replicate exceptions are certain exceptions that can occur in the reflection mechanism used for extensibility. It is possible to find a class successfully but fail to instantiate it. The simplest way this could occur if the class found is an abstract class, but this requires the creation and compilation of a whole class for testing, which we just did not have time to do. I/O exceptions can also be difficult to replicate; it is easy to cause the conditions for an I/O error due to a non-existent file, but more complicated to simulate exceptions for when the operation is interrupted or the operation is on a target file the system is not authorized to perform operations on.

The types of exceptions that could occur, but logically cannot in our system not only decrease the coverage of the test suite for a class, but also clutter the code more. There are very few of these in the system, but it would be possible to use a library such as Lombok which, among other capabilities, provides annotations precisely to prevent these types of exception catches from being necessary. This was not implemented in the system because the group members did not become aware of this library until late in the winter semester when the project timeline was almost complete.

One of the requirements for the system was that it should run properly on Windows, Linux, and Apple machines. As the system runs on the Java Virtual Machine (JVM) no challenges were expected here, but the test suite for the system was executed on each platform to ensure everything works as expected. The system was also tested manually with the deployed jar through the command line on each platform to ensure the software behaves as expected and gives the same results on all systems.

10 Case Study

Once the system was largely completed, the project supervisor provided some sample test suites. These were used to test the ability of the system to handle new test case formats, and to measure the performance of the system.

The test cases supplied are all in the same format. This format is a path through a state machine, denoted as a list of events and resulting states separated by dashes:

Start-<event>-<state>-<event>-<state>. . .

From this format, it would be possible to consider three things when studying test case similarity: the sequence of states traversed, the sequence of events, or the sequence of pairs of events and resulting states. As described in the user manual, this requires writing three new DataRepresentation classes. Despite all three DataRepresentation classes operating on the same test case format, different classes are required because each DataRepresentation defines what information is extracted from a test case, which is different for each consideration above. This means that the three DataRepresentations are largely similar in their implementation, but this is not an issue because there is opportunity for code reuse in the system (ie. there is nothing to prevent super/subclassing in the DataRepresentation implementations). A DataRepresentation was created for each consideration and, once placed in the appropriate package, could be discovered and used by the rest of the system without any other changes necessary.

When analyzing performance, the sample test suites were used to generate several test suites of various sizes. The execution times for a command on each differently sized test suite were recorded and used to plot regression lines that indicate the performance of the system for other test cases sizes not explicitly tested here. The test cases and measurements recorded are done via another Java script, so the execution times may not be exact. The execution times are likely to be slightly lower than actual times since a JVM does not need to start up for each execution, but the remainder of the system operations are the same. The main method for the system makes a call to a single public method in the Controller, and this method is called several times in the test script to perform calculations. This script is included with the rest of the codebase in the testing folder so the process here can be repeated, but we have ensured that the execution of this test script did not contribute to the Jacoco coverage reports discussed in the previous section.

These test suites were made by randomly selecting samples from the largest of the test suite files (approximately 1400 test cases), and each sized test suite was run through the system under one pairwise and listwise metric. The script to generate different sized test suites creates 30 different sized test suites[5], with the largest suite being the full 1400 test cases. These are evenly spaced apart, so each sized test suite is in increments of 46 test cases. The actual result of these diversity calculations was not considered here, only the execution time of the operation. Each different sized test suite was executed five times on each metric to obtain average execution times, which were plotted with different numbers of threads used by the system.

[5] In accordance with the Central Limit Theorem, 30 sample points is the threshold at which the collection of sample points is considered to sufficiently represent the data.

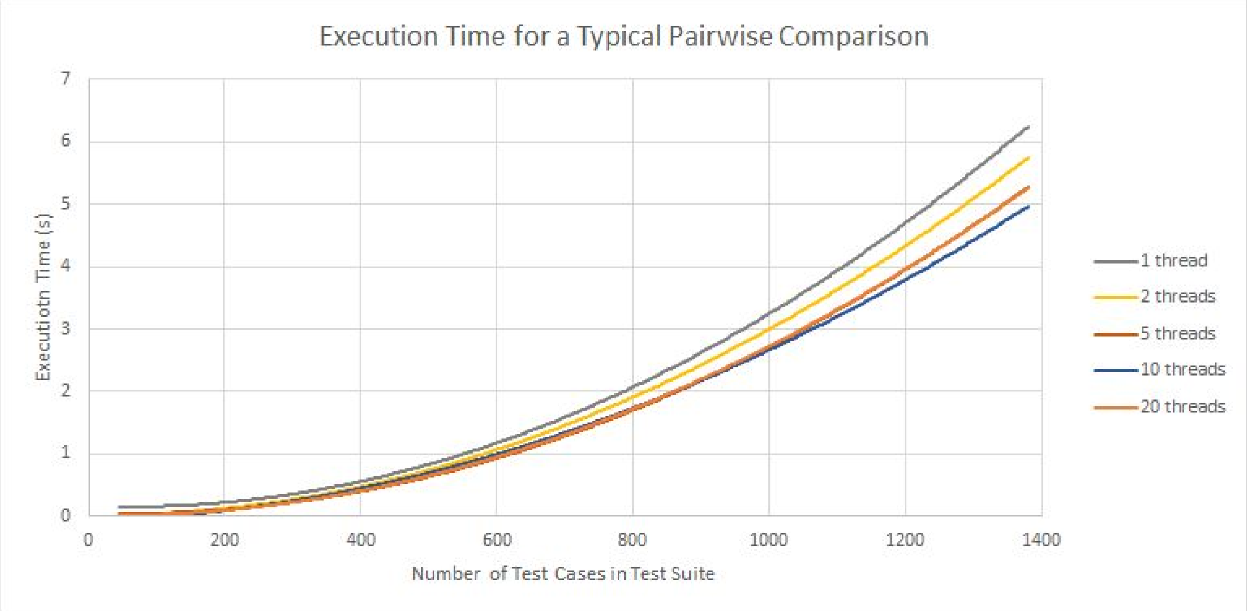


Figure 31: The execution time of various sized test cases for a pairwise comparison with different numbers of execution threads.

The graph above shows the average execution times of several different sized test suites using a Levenshtein distance pairwise metric. [1] As it can be seen, while the execution times for any number of threads is around 5 to 6 seconds, the execution time is growing according to n^2. This is because the number of pairs to compare grows at the same rate, specifically, the number of pairs grows by

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

Multithreading the execution ultimately did not do too much to improve system performance, although the improvements will become greater with even larger test suites. This is possibly because the additional overhead from adding threads cancels out a lot of the speedup, or because the many pairwise comparisons did not end up being as dominant of a workload as anticipated. As the number of threads increases, the performance improves to a point, but the performance for five threads and twenty threads are more or less identical, with ten threads providing better performance than either. At some point between ten and twenty then, the extra overhead needed for that number of threads outweighs any speedup from concurrency.

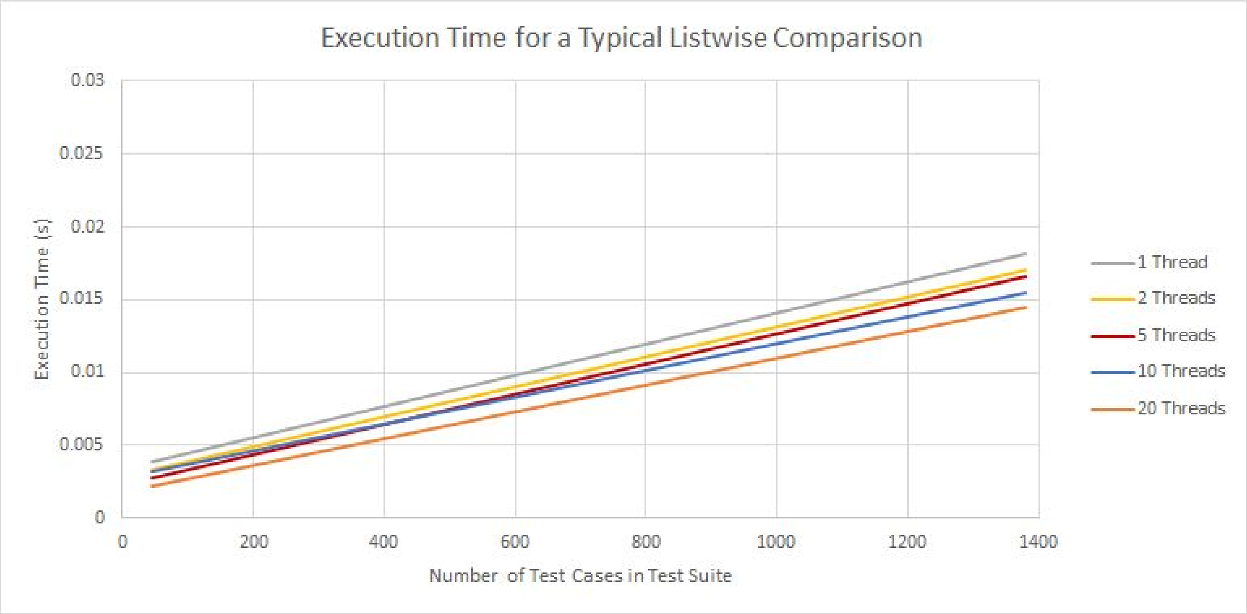


Figure 32: The execution time of various sized test cases for a listwise comparison with different numbers of execution threads.

This graph shows the execution times for variously sized test suites using Nei's measure, a listwise diversity metric. It can be seen that the average execution times for test suites here are much lower compared to pairwise metrics of a similar size. This is because a listwise metric only needs to look at each test case once while performing a calculation, while pairwise metrics must look at each test case several times. The execution time then increases much less with the size of the test suite, which can be seen by the execution times' linear trend. Different numbers of threads used here do not significantly impact the execution time, which is as expected since there is not much that can be parallelized in a listwise metric. This is because regardless of the size of the test suite, there is still only one calculation being made. In our solution, each calculation is encapsulated in an object, and those objects are parallelized, so without refactoring how the parallelization is done, threading will not grant much speedup. This refactoring is not really necessary however, since from the graph in figure 32, it can be seen that these times are all well under one second and so are already quite fast.

It should be noted that the execution time for the system can also vary in a way not captured by these graphs: the size of each individual test case. In these test suites, the number of elements to compare in each test suite were for the most part between five and ten, meaning that most paths traversed between five and ten states in a state machine. If the average length of test cases in the test suite increases, the execution time will increase as well.

11 Challenges Encountered During the Project

11.1 System Performance

The main challenge faced in the implementation of this project was handling DataRepresentation objects in pairwise comparisons. In a comparison operation, the system originally produced one DataRepresentation object for each test case in the specified file(s), which are consumed by whatever metric is used for the diversity calculation. This worked fine for listwise metrics, but the problem for pairwise metrics was that one test case would belong to several pairwise comparisons, so there was a single instance of each test case being consumed by several concurrently executing methods, which yielded incorrect results. This was not noticed early on because many of the initial tests written for performing pairwise comparisons only included two test cases, so there was no opportunity to see more than one calculation access the same instance of a DataRepresentation and yield incorrect results.

This was a result of the design of the DataRepresentation objects. As discussed in the *Design Patterns* section, the design choice to give users the freedom to choose the format they store their test case within their implementation of a DataRepresentation, means that there is no decoupling the test case and the iteration over the test case. The result here is that each DataRepresentation object could only be iterated over once without providing more work for a user to do when implementing a new DataRepresentation. Pushing more work on to the user was to be avoided, so the team looked into ways to easily make copies of the DataRepresentations that exist in the system.

The DataRepresentation objects ended up being cloned for each pairwise comparison, so each calculation could iterate over a separate instance of a test case. This made the system work properly, and produce correct results, but there was a performance issue. The cloning methods used were implemented though serialization, which meant the system had to do a lot more work. This yielded the execution times shown in a graph below:

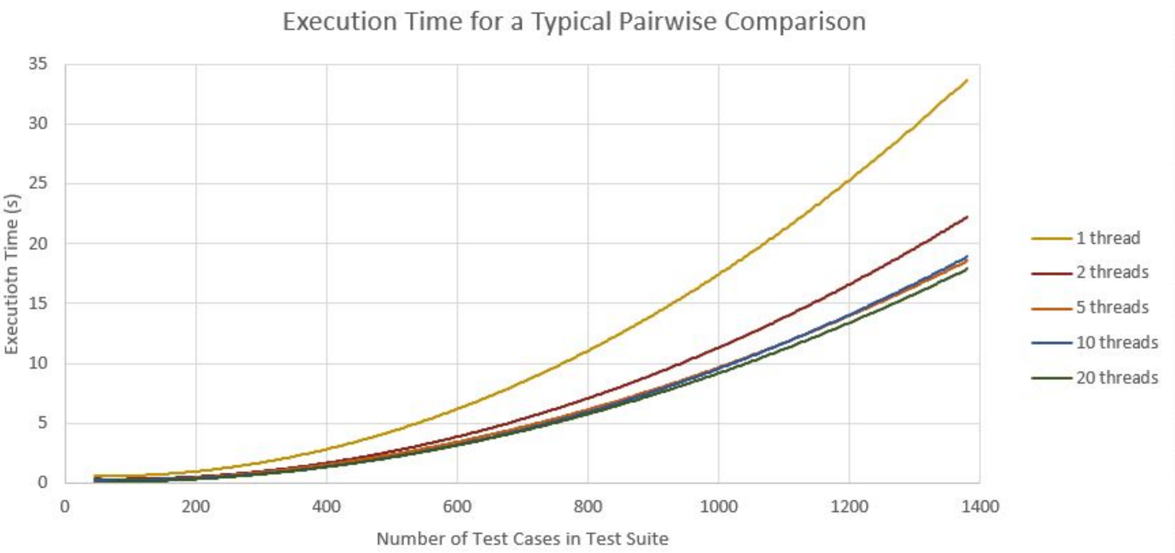


Figure 33: Execution time for different sized test suites in an early version of the system.

Compared to the final performance of the system, this is much worse. In this version of the system, multithreading the comparisons achieved much more speedup than the final version, but the multithreaded system still was two to three times slower than the final version of the system executing with a single thread.

To solve this, the design of the system needed to be tweaked a bit. Rather than directly read each test case into a DataRepresentation in the FileReaderService, and then need to clone those objects, the design was changed to read in the test cases as strings. While only one diversity measurement can use an instance of a DataRepresentation, multiple identical DataRepresentations can be created from a single test case string, so the creation of the DataRepresentation objects was deferred to allow multiple identical DataRepresentations to be instantiated. This change yielded the same diversity calculation results, but ended up being much faster and gives the performance graphs shown in the *Case Study* section.

11.2 Saving User Default Choices

Another issue that the team faced was with the way that we let users persist their default choices for diversity calculation parameters. The system was to have a configuration file where these would be stored, which worked fine while the system was being written in an IDE, but behaved differently when the project was deployed as a JAR. This was because the configuration file was packaged into the JAR, and while files inside a JAR can be read during execution, the writing process is much more complicated. The team wanted the whole system to be confined to the single file so the tool did not populate the user's file system with several files, but after several attempts this could not be achieved, so it was settled that the configuration file would live outside the project's JAR in whatever directory the JAR is placed. When the system starts up, if there is no configuration file present in the directory, a configuration file is copied out of the JAR for the system to use.

12 Next Steps

Now that the application is completed, there are many small quality of life changes that could be added. During normal operation of the system, there are two loading bars that display the current task progress, one for creating pairs of items to compare and one for the comparisons. Displaying these progress bars has an impact on performance since the tasks need to be monitored. Giving the option to disable the progress bars and other verbose logging may be beneficial for extremely large datasets as this would improve performance. The user could be given an option of whether they want verbose logging, or greater performance. Importing a logging library like log4j may also help with controlling output.

The application expects a single command, executes it, and prints the results, but there may be cases where a user wants to submit many commands at once for different run configurations and view the results at the end. It would be possible for a user to write their own script to execute multiple commands, but this requires further programming knowledge and it would be nice if the system could accept a file containing multiple commands that it would execute in series.

Another change made to the application is reducing the size of the Controller by relieving some responsibility. The Controller is the only component that has two responsibilities - coordination of services and error handling. To remove the error handling functionality, a specialized service can be created to handle errors or aspect oriented design could be used. The aspect oriented approach may be implemented using Spring, AspectJ or another similar library or framework that enables static code to run before and after certain functions defined in a configuration class. Common error reporting tasks can be delegated to an aspect, reducing the responsibility of the controller and the amount of code.

Late in the development cycle, the team became aware of the Spring framework in another course. The use of this framework for the system could be used to reduce the size of the tool, since Spring would handle some of the reflection code the team wrote themselves. This could simplify the design, but most of the other functions a framework has included aren't necessary for this type of application, which would complicate how the application runs, builds and deploys unnecessarily.

Since the compiled jar is small (packaged with all its dependencies, the jar is 317KB), it can work better as a plugin or dependency on another project that requires an extensible comparison library. Making the project available on maven central for other developers to use and updating whenever possible is one of the final tasks that can be done for the project.

Some of the niche exceptions not explicitly tested, for reasons discussed in the *Software Testing* section, could be simulated using a tool like Mockito. With a better test coverage tool, more tests could be added to satisfy other coverage criteria such as edge-pair and round trip paths.

13 Conclusion

Over the course of two terms, a research tool has been produced that can perform diversity calculations on the test cases of a test suite according to several diversity metrics. The system can support different formats of test cases and output results in different human and machine readable formats. The system can be extended to support new test case formats, diversity metrics, and output formats by writing short Java classes to implement simple, well-defined interfaces such as the DataRepresentation interface, Pairwise or Listwise ComparisonStrategy, AggregationStrategy and ReportFormat interfaces. Extending the system this way requires no change to the existing system code. The project followed an iterative software lifecycle and was completed on time, although there are some features discussed in the *Next Steps* section that could be added. The Appendix contains the *User Manual* for the system, which includes a link to the repository containing the software.

14 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.

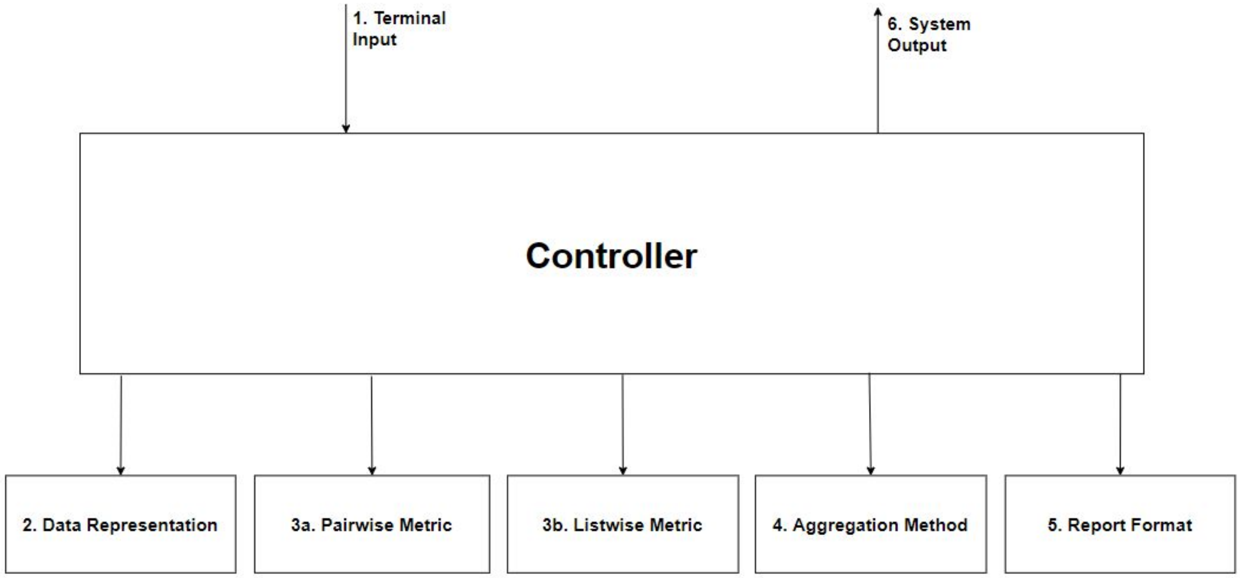
[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Command ," in Design Patterns: Elements of Reusable Object-Oriented Software, Boston, Addison-Wesley, 1994, pp. 219-228.

15 Appendix: User Manual

Project Description

The system allows a user to measure the diversity within 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 it 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	The default listwise diversity metric to use in the system if no metric is specified.	ShannonIndex
listwiseMethodLocation	The path to the folder containing listwise comparison methods.	metrics.comparison.listwise
pairwiseMethod	The default pairwise diversity metric to use in the system if no metric is specified.	CommonElements
pairwiseMethodLocation	The path to the folder containing pairwise 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	HALF_UP

	/docs/api/java/math/RoundingMode.html for valid options here.	
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 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

In this section, there are a number of examples of instructions that can be entered into the system along with the results of the command. All these examples use the following files described in the following table:

Filename	**Contents**
test	1,1,0,1,0,1,0,1 1,0,1,0,1,1,1,1 0,0,0,1,0,1,0,1 1,0,1,1,1,1,1,1
suite	Start-1-StateA-4-StateB-12-StateC Start-1-StateB-4-StateD-13-StateB-24-StateF Start-1-StateB-4-StateA-13-StateE-25-StateF Start-0-StateA-4-StateB-13-StateE-26-StateF Start-0-StateA-4-StateB-13-StateE-27-StateD-56-StateC Start-1-StateB-4-StateC-13-StateE-27-StateD-57-StateC Start-1-StateB-4-StateC-13-StateE-27-StateD-58-StateF

each instruction is listed below in using the following template:

Instruction	This section will show the input into the terminal.
Explanation	This section will show a text description of the input and output for the instruction.
Console Display	This section will show the contents displayed to the terminal once the instruction has completed.
File(s) Affected	This section will show any files that are altered as a result of the command, including the filename and contents of the file.

the output of these instructions is based on the contents of the system JAR at the time of writing. If any metrics, etc. have been added by a user, or if values in the configuration file have changed, the results may not be identical if these exact instructions are entered.

Help Examples

Instruction	help
Explanation	Displays the commands available to the system
Console Display	compare <filename> [<filename>] <data-representation> performs a diversity calculation within a test suite, or between test suites at the specified filename(s) -m <metric>: set the diversity metric to use in the calculation. Available metrics can be found with 'help -m' -a <method>: set the method to use for aggregating results. Available methods can be found with 'help -a' -r [<format>]: set the report formats to use to display results. Available formats can be found with 'help -r' -d <delimiter>: set the delimiter that separates test cases within the passed test suite file(s). This can be a character, string, or regular expression -s <filename>: denote that the results of the operation should be saved to a file named <filename> -t [<integer>]: denote that the operation should use a thread pool for concurrency, and optionally specify the number of threads config <parameter> <value> sets the value of a parameter read from the configuration file help lists information on the requested topic -m: lists the available comparison metrics in the system -a: lists the available aggregation methods in the system -f: lists the available data representations in the system -r: lists the available report formats in the system
File(s) Affected	none

Instruction	help -m
Explanation	Displays the available pairwise and listwise diversity metrics in the system
Console Display	Available pairwise metrics are: CommonElements: counts the number of elements that appear at the same index in both sequences CompressionDistance: calculates the similarity of two test cases by the ratio of size between the individual test cases compressed, and the compression of the test cases concatenated Dice: a measure of overlap between two sets, with 0 being distinct and 1 being complete equality Hamming: compares two test cases by the number of positions in the test cases that are different JaccardIndex: provides a ratio between the intersection of two sets and the union of the sets Levenshtein: Calculates the minimum number of insertion/deletion/modification operations to transform one test case into another LongestCommonSubstring: returns a value indicating the longest common sequence of contiguous elements in a pair of test cases Available listwise metrics are: Nei: yields the average of the simpson diversity calculated across each position in the test case sequences. ShannonIndex: A measure of the entropy for sets. Commonly used to measure diversity within a population in life sciences SimpsonDiversity: a value between 0 and 1 indicating diversity within a set. Values closer to 1 are more diverse StoddardIndex: report the relative frequency of test case elements as a value >= 1. Larger values represent more diverse suites.
File(s) Affected	none

Configuration Examples

Instruction	config numThreads 10
Explanation	Set the default number of threads for the system to use to ten
Console Display	Successfully set numThreads to the value 10
File(s) Affected	numThreads value in config.json modified to 10

Comparison Examples

Instruction	compare test CSV
Explanation	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). For the default configuration file, this will be the average value of all CommonElements pairwise comparisons.
Console Display	4.5
File(s) Affected	None

Instruction	compare suite EventSequence -m Hamming -a AverageValue -t 5 -s
Explanation	Perform a diversity calculation on a file/folder of files called ‘suite’, 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
Console Display	1.48
File(s) Affected	This creates a file called “defaultName” with the contents “1.48”

Instruction	compare test CSV -s out -t -r XML -m Levenshtein -a MaximumValue
Explanation	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. 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.

Console Display	<report>
	    <username> </username>
	    <hostname> </hostname>
	    <datetime> </datetime>
	    <parameters>
	        <outputfilename> out </outputfilename>
	        <usethreadpool> true </usethreadpool>
	        <comparisonmethod> Levenshtein </comparisonmethod>
	        <delimiter>
	 </delimiter>
	        <reportformats> [XML] </reportformats>
	        <aggregationmethods> [MaximumValue]
	</aggregationmethods>
	        <numberofthreads> 15 </numberofthreads>
	        <save> true </save>
	        <testcaselocationone> test.txt </testcaselocationone>
	        <datarepresentation> CSV </datarepresentation>
	    </parameters>
	    <similarities>
	        <similarity> 3.0 </similarity>
	        <similarity> 1.0 </similarity>
	        <similarity> 4.0 </similarity>
	        <similarity> 3.0 </similarity>
	        <similarity> 1.0 </similarity>
	        <similarity> 3.0 </similarity>
	    </similarities>
	    <results>
	        <maximumvalue> 4.0 </maximumvalue>
	    </results>
	</report>

File(s) Affected	A file called “out” is created with the contents as what is displayed to the console.

Instruction	compare suite CSV -r JSON Pretty -a AverageValue MedianValue -s out
Explanation	Perform a diversity calculation between the test suite named ‘test’. All test cases in the test suite are formatted according to the CSV data representation. The results will be saved to a file called ‘outJSON’ in JSON format, and called ‘outPretty’ in a human readable format. The results reported will be the average and median of all the pairwise comparisons made.
Console Display	{"username":"","hostname":"","datetime":"","parameters":{"outputfilename":"out","usethreadpool":"false","delimiter":"\r\n","reportformats":"[JSON, Pretty]","aggregationmethods":"[AverageValue, MedianValue]","save":"true","testcaselocationone":"test.txt","datarepresentation":"CSV"},"similarities":["3.0","3.0","4.0","4.0","6.0","7.0"],"results":{"medianvalue":"[4.0, 4.0]","averagevalue":"[4.5]"}} /****************** * Comparison Report * User: * Host: * Date: ****************** Run parameters: outputFilename: out useThreadPool: false delimiter: reportFormats: [JSON, Pretty] aggregationMethods: [AverageValue, MedianValue] save: true testCaseLocationOne: test.txt dataRepresentation: CSV Results: MedianValue: [4.0, 4.0] AverageValue: [4.5]

File(s) Affected	A file is created called outJSON with the following content: {"username":"","hostname":"","datetime":"","parameters":{"outputfilename":"out","usethreadpool":"false","delimiter":"\r\n","reportformats":"[JSON, Pretty]","aggregationmethods":"[AverageValue, MedianValue]","save":"true","testcaselocationone":"test.txt","datarepresentation":"CSV"},"similarities":["3.0","3.0","4.0","4.0","6.0","7.0"],"results":{"medianvalue":"[4.0, 4.0]","averagevalue":"[4.5]"}} Another File called outPretty is created with the following content: /***************** * Comparison Report * User: * Host: * Date: ***************** Run parameters: outputFilename: out useThreadPool: false delimiter: reportFormats: [JSON, Pretty] aggregationMethods: [AverageValue, MedianValue] save: true testCaseLocationOne: test.txt dataRepresentation: CSV Results: MedianValue: [4.0, 4.0] AverageValue: [4.5]

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 computing 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]:

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

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 longest 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:

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

where, P is the set of test cases, T is the number of unique elements across all test cases in the test suite, 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 greater than or equal to 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:

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

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 pairwise 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:

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

where each x_i is a pairwise comparison result.

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

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

MedianValue yields the median from a set of pairwise 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 pairwise diversity results.

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

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

Summation simply sums together every pairwise 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 a 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. The filename specified in the user command is used for each file, with the report format name appended to it. For example, if a user wants to save results to “out.txt” and selects both JSON and XML output formats, the files created will be “outJSON.txt” and outXML.txt”.

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>”.