COMP SCI 2ME3 and SFWR ENG 2AA4 Final Examination
McMaster University

DAY CLASS, **Version 1** Dr. S. Smith
DURATION OF EXAMINATION: 2.5 hours (+ 30 minutes buffer time)
MCMASTER UNIVERSITY FINAL EXAMINATION April 28, 2021

NAME: [Enter your name here —SS]

Student ID: [Enter your student number here —SS]

This examination paper includes 23 pages and 8 questions. You are responsible for ensuring that your copy of the examination paper is complete. Bring any discrepancy to the attention of your instructor.

By submitting this work, I certify that the work represents solely my own independent efforts. I confirm that I am expected to exhibit honesty and use ethical behaviour in all aspects of the learning process. I confirm that it is my responsibility to understand what constitutes academic dishonesty under the Academic Integrity Policy.

Special Instructions:

1. For taking tests remotely:
   - Turn off all unnecessary programs, especially Netflix, YouTube, games like Xbox or PS4, anything that might be downloading or streaming.
   - If your house is shared, ask others to refrain from doing those activities during the test.
   - If you can, connect to the internet via a wired connection.
   - Move close to the Wi-Fi hub in your house.
   - Restart your computer, 1-2 hours before the exam. A restart can be very helpful for several computer hiccups.
   - Use a VPN (Virtual Private Network) since this improves the connection to the CAS servers.
   - Commit and push your tex file, compiled pdf file, and code files frequently. As a minimum you should do a commit and push after completing each question.
   - Ensure that you push your solution (tex file, pdf file and code files) before time expires on the test. The solution that is in the repo at the deadline is the solution that will be graded.
   - If you have trouble with your git repo, the quickest solution may be to create a fresh clone.
2. It is your responsibility to ensure that the answer sheet is properly completed. Your examination result depends upon proper attention to the instructions.

3. All physical external resources are permitted, including textbooks, calculators, computers, compilers, and the internet.
4. The work has to be completed individually. Discussion with others is strictly prohibited.
5. Read each question carefully.
6. Try to allocate your time sensibly and divide it appropriately between the questions. Use the allocated marks as a guide on how to divide your time between questions.
7. The quality of written answers will be considered during grading. Please make your answers well-written and succinct.
8. The set $\mathbb{N}$ is assumed to include 0.

Question 1 [5 marks] What are the problems with using "average lines of code written per day" as a metric for programmer productivity?

[Provide your reasons in the itemized list below. Add more items as required. —SS]

- Productivity when it comes to the software development process is a measure of how efficiently the process produces software. However, it is important to note that efficiency does not directly correlate to the number of lines of code written by the developer. There are points in time in the development process where writing an abundance of lines of code is actually frowned upon. In terms of core software qualities, writting more lines of code than necessary often sacrifies essentiality, as there will be times in which lines of code that are not necessarily needed could have been omitted. Additionally, when it comes to writting test cases, it is not necessarily about the number of test cases that are written, but rather how significant the test cases written are for testing all boundaries of the code. This is the key principle of white box testing, where time must be taken to develop test cases that are as close to 100 percent code coverage as possible through specific coverage testing. However, this has nothing to do with the number of lines of code that is written for these test cases, but rather the amount of boundaries and edge cases that the written test cases actually cover. Finally, if a developer is caught up on the number of lines of code they must write, they might sacrifice the crucial time required in the early stages of development when establishing requirements and design for a product. Going directly into implementation may end up increasing the amount of time taken as there was no real effort to consider what the actualy problem is, and how it should be solved. Therefore, although productivity is a difficult measure when it comes to software development, it should not be measured by the number of lines a developer writes.

Question 2 [5 marks] Critique the following requirements specification for a new cell phone application, called CellApp. Use the following criteria discussed in class for judging the quality of the specification: abstract, unambiguous, and validatable. How could you improve the requirements specification?

"The user shall find CellApp easy to use."

[Fill in the itemized list below with your answers. Leave the word in bold at the beginning of each item. —SS]

- **Abstract** - This specification is abstract because it sticks to stating that the user finds the CellApp easy to use, but does not state how or why this is the case.
- **Unambiguous** - This specifification is ambigious because the word ”easy” can be interpreted in many different ways and does not give a clear indication of how the user finds the app.
- **Validatable** - Because the specification is ambigious and does not give a proper measurement of how easy the user should find the CellApp, this specification is not validatable.
- **How to improve** - In this case, making the specification unambigious is very important as it will ensure that it can be validatable. This can be done by providing more information on what it means for the user to find the app ”easy to use”. An example would be ”The user can run the CellApp in 4 commands”. This will allow the specification to be validatable as well.

Question 3 [5 marks] The following module is proposed for the maze tracing robot we discussed in class (L20). This module is a leaf module in the decomposition by secrets hierarchy.

Module Name find_path

Module Secret The data structure and algorithm for finding the shortest path in a graph.

[Fill in the answers to the questions below. For each item you should leave the bold question and write your answer directly after it. —SS]

A. **Is this module Hardware Hiding, Software Decision Hiding or Behaviour Hiding? Why?**

   - This module would be considered software decision hiding. This is because the data structure and algorithm used to find the shortest path in a graph is a software product that can be implemented generally for any graph related problem, and would not necessarily be specific for this maze tracing robot.

B. **Is this a good secret? Why?**

   - This is not a good secret. This is because the secret describes both a data structure AND algorith, however a good secret should be specific to one module. To make it better, both the algorithm and data structure for the shortest path should be separated into two different secrets.

C. **Does the specification for maze tracing robot require environment variables? If so, which environment variables are needed?**

   - The maze tracing robot specification would benefit from an environment variable in order to store the state of the screen itself, the state of the motor of the robot, the position of the robot, etc. This is because the module may interact with the outside environment specifically for the maze tracing robot example.

Question 4 [5 marks] Answer the following questions assuming that you are in doing your final year capstone in a group of 5 students. Your project is to write a video game for playing chess, either over the network between two human opponents, or locally between a human and an Artificial Intelligence (AI) opponent.

[Fill in the answers to the questions below. For each item you should leave the bold question and write your answer directly after it. —SS]

A. **You have 8 months to work on the project. Keeping in mind that we usually need to fake a rational design process, what major milestones and what timeline for achieving these milestones do you propose? You can indicate the time a milestone is reached by the number of months from the project's start date.**

   - In order to produce quality software, the rational process described by Parnas will act as a great model for my project. We will think about the requirements that the game would like to have, and overall what type of behvaiours we expect the game to have as well. We will then move on to thinking more specifically about the design itself, and how exactly the problem will be solved, including how the requirements will be satisfied. This will be better documented through the use of an MIS, and should be aimed to be completed within 1 month of starting on the project. Once we've gotten a general, but not necessarily conrete idea of the desired implementation, we will begin to code the solution to the problem. In parallel with coding, we should pay attention to potential test cases in order to verify the behaviour and requirements of the code. These steps will be done incrementally in order to eventually build up the software product. Specifically, it would be good practice to make efforts to verify the code through testing every week or so. It is quite possible that while implementing or verifying the product, we will need to make changes to the intiail design and requirements that were written. This is where the idea of "faking the rational design process" comes into play, where we will modify the previously written MIS over the course of time as we encounter more edge cases and design problems within our code. As referenced by the third question, there will also be efforts to berify the installability of the game by testing on external environments such as VMs. This should not be done at the end of the 8 months, but rather incrementally as well, potentially every 2-3 weeks. By maintaining these steps, we can eventually move on to the final stage and think about how the game will be delivered and how we wish to maintain it going forward, potentially 3-4 weeks prior to the end deadline.

B. **Everything in your process should be verified, including the verification. How might you verify your verification?**

   - Mutation testing would be best for verifying your verification. Test cases will be built by intentionally adding faults. These test cases will be run and that these test cases also pick up on the faults that we've added as well. If the test cases are not sufficient, they will not pick up on the seeded faults that were artificially added.

C. **How do you propose verifying the installability of your game?**

   - In order to verify the installability of the game, it is important that I test the game on different environments. This can either be done by asking other users to test the game on their machine, or to use my own virtual machine. This will ensure that the game is able to run on different machines effectively without any unexpected faults. Additionally, this external testing should not be done at the very end of the 8 months, it is important to test on the virtual machine incrementally as time goes on to avoid facing large issues at the very end of development.

Question 5 [5 marks] As for the previous question, assume you are doing a final year capstone project in a group of 5 students. As above, your project is to write a video game for playing chess, either over the network between two human opponents, or locally between a human and an Artificial Intelligence (AI) opponent. The questions below focus on verification and testing.

[Fill in the answers to the questions below. For each item you should leave the bold question and right your answer directly after it. —SS]

A. **Assume you have 4 work weeks (a work week is 5 days) over the course of the project for verification activities. How many collective hours do you estimate that your team has available for verification related activities? Please justify your answer.**

   - Verification of the code is just as important as the implementation itself, because it does not only ensure that the output of the product is the expected/desired result, it also uncovers potential mistakes in both the implementation and indirectly, the initial requirements and design process and allows for an increase in building confidence in the software. It is important that verification is not simply left to the end, but rather integrated throughout the implementation process. For this reason, in accordance with best practices in production environments, it would be my best interest to dedicate 2-3 hours at the end of every work week for such verification activities that involve the entire group, potentially increasing this number as the project becomes more intensive and more code has been written. There should also be individual efforts for verification independent of the team that would also increase this time. Therefore, an estimation of about 20 hours total would be reasonable for verification activities.

B. **Given the estimated hours available for verification, what verification techniques do you recommend for your team? Please list the techniques, along with the number of hours your team will spend on each technique, and the reason for selecting this technique.**

   - Verification techniques can include analysis, inspection and testing of the software product written. There should be both static and dynamic attempts to verify the software product. First, static techniques such as Rubber Duck testing and code walkthroughs can be used. This includes either the developer sitting by themselves, reading the lines of code to ensure that all lines work as desired, as well as getting in a group environment and talking about the lines of code. This will easily uncover basic faults in the code just be vocalizing the process of the product. Additionally, dynamic testing should be conducted, that involves test cases that are designed specifically for the software product. Although dynamic testing does not guarentee correctness, these tests will still uncover many errors and build confidence in the software. Techniques described by both white-box testing specifically for the code, and black-box testing specifically for the created requirements. The static testing individually, along with the individual test cases run will account for about one third of the 20 allocated hours. The rest will be allocated to more group-based verification techniques, both through more intensive dynamic testing and the code walkthroughs.

C. **Is the oracle problem a concern for implementing your game? Why or why not? If it is a concern, how do you recommend testing your software?**

   - For a game that we are developing from scratch, chances are there will be components of the expected output/behaviour that is not necessarily known, especially when components of AI technology are desired to be integrated. In this case, oracles will be requierd at each stage of the testing process to tell us what the right answer is, but an oracle itself does not necessarily have to be used. Instead, metamorphic testing will come in handy where an independent program will be used to approximate the oracle. We will manufacture solutions in order to continue on our development process by examining properties of the expected values to determine whether the output itself is wrong. This can be done in addition to previously explained whitebox and blackbox testing.

Question 6 [5 marks] Consider the following natural language specification for a function that looks for resonance when the input matches an integer multiple of the wavelengths 5 and 7. Provided an integer input between 1 and 1000, the function returns a string as specified below:

- If the number is a multiple of 5, then the output is "resonance 5"
- If the number is a multiple of 7, then the output is "resonance 7"
- If the number is a multiple of both 5 and 7, then the output is "resonance 5 and 7"
- Otherwise, the output is "no resonance"

You can assume that inputs outside of the range 1 to 1000 do not occur.

A. What are the sets D_i that partition D (the input domain) into a reasonable set of equivalence classes?

- There would need to be 4 sets created in order to cover all of the input domain, while ensuring that all sets are disjoint. The first set would be all integers from 1 to 1000 that are a multiple of 5. The next set would be all integers from 1 to 1000 that are a multiple of 7. The next set would be all integers that are multiples of both 5 and 7. The final set would be all integers that are multiples of neither 5 or 7.

B. Given the sets D_i, and the heuristics discussed in class, how would you go about selecting test cases?

- Test cases can be chosen based on the sets created, where suitable test cases must be created for each set. These test cases would be created in accordance to the whitebox coverage testing. In order to achieve both statement and condition coverage, there must be a test case that invokes each of the 4 return statements. There should be additional testing to pay attention to edge coverage. I would draw out a control graph to view all possible routes that the path can be taken, and all possible values of the consituents of compound conditions will be exercised at least once. For example, for the part of the code that should output resonance 5, there should be a case where the number isn't a multiple of 5, and there should be a case for when there is a number of multiple of 5. Many of the cases already created to satisfy the conditions of other parts of the code will already cover part of the edge coverage tests, and so I should make an effort to ensure that there aren't any repeats of tests. Overall, this should include cases in which the value themselves are chosen, values that are very close to 5 and 7, as well as extreme edge cases. This will ideally improve the robustness of the code to test as many cases as possible that are considered to be unexpected.

Question 7 [5 marks] Below is a partial specification for an MIS for the game of tic-tac-toe (`https://en.wikipedia.org/wiki/Tic-tac-toe`). You should complete the specification.

[The parts that you need to fill in are marked by comments, like this one. You can use the given local functions to complete the missing specifications. You should not have to add any new local functions, but you can if you feel it is necessary for your solution. As you edit the tex source, please leave the `wss` comments in the file. You can put your answer immediately following the comment. —SS]

Syntax

Exported Constants

SIZE = 3 *//size of the board in each direction*

Exported Types

cellT = { X, O, FREE }

Exported Access Programs

Routine name	In	Out	Exceptions
init			
move	$\mathbb{N}$, $\mathbb{N}$		OutOfBoundsException, InvalidMoveException
getb	$\mathbb{N}$, $\mathbb{N}$	cellT	OutOfBoundsException
get_turn		cellT	
is_valid_move	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	OutOfBoundsException
is_winner	cellT	$\mathbb{B}$	
is_game_over		$\mathbb{B}$	

Semantics

State Variables

b: boardT
$Xturn$: $\mathbb{B}$

State Invariant

[Place your state invariant or invariants here —SS]

number of Xs + number of Os + number of FREEs = 9
$\forall i, j \in \mathbb{N}$ such that $b[i][j]$ is a valid cell,$0 \leq i, j \leq 2$

Assumptions

The init method is called for the abstract object before any other access routine is called for that object. The init method can be used to return the state of the game to the state of a new game.

Access Routine Semantics

init():

- transition:

$$Xturn, b := \text{true}, < \begin{array}{c} <\text{FREE}, \text{FREE}, \text{FREE}> \\ <\text{FREE}, \text{FREE}, \text{FREE}> \\ <\text{FREE}, \text{FREE}, \text{FREE}> \end{array} >$$

- exception: none

move(i, j):

- transition: $Xturn, b[i, j] := \neg Xturn, (Xturn \Rightarrow \text{X} | \neg Xturn \Rightarrow \text{O})$
- exception

  $exc := (\text{InvalidPosition}(i, j) \Rightarrow \text{OutOfBoundsException} | \neg \text{is\_valid\_move}(i, j) \Rightarrow \text{InvalidMoveException})$

getb(i, j):

- output: $out := b[i, j]$
- exception $exc := (\text{InvalidPosition}(i, j) \Rightarrow \text{OutOfBoundsException})$

get_turn():

- output: $Xturn \Rightarrow \text{X} | \neg Xturn \Rightarrow \text{O}$
- exception: none

is_valid_move(i, j):

- output: $out := (b[i][j] = \text{FREE})$
- exception $exc := (\text{InvalidPosition}(i, j) \Rightarrow \text{OutOfBoundsException})$

is_winner(c):

- output: $out := \text{horizontal\_win}(c, b) \vee \text{vertical\_win}(c, b) \vee \text{diagonal\_win}(c, b)$
- exception: none

is_game_over():

- output: $is\_winner(b) \vee \forall i \in b | i \neq \text{FREE} \Rightarrow \text{True} | \text{True} \Rightarrow \text{False}$
- exception: none

Local Types

boardT = sequence [SIZE, SIZE] of cellT

Local Functions

InvalidPosition: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

InvalidPosition$(i, j) \equiv \neg((0 \leq i < \text{SIZE}) \wedge (0 \leq j < \text{SIZE}))$

count: cellT $\rightarrow \mathbb{N}$

count$(c) \equiv +(i : \mathbb{N} | 0 \leq i \leq \text{SIZE} : +(j : \mathbb{N} | 0 \leq j \leq \text{SIZE} \wedge b[i, j] = c : 1))$

horizontal_win : cellT $\times$ boardT $\rightarrow \mathbb{B}$

horizontal_win$(c, b) \equiv \exists(i : \mathbb{N} | 0 \leq i < \text{SIZE} : b[i, 0] = b[i, 1] = b[i, 2] = c)$

vertical_win : cellT $\times$ boardT $\rightarrow \mathbb{B}$

vertical_win$(c, b) \equiv \exists(j : \mathbb{N} | 0 \leq j < \text{SIZE} : b[0, j] = b[1, j] = b[2, j] = c)$

diagonal_win : cellT $\times$ boardT $\rightarrow \mathbb{B}$

diagonal_win$(c, b) \equiv (b[0, 0] = b[1, 1] = b[2, 2] = c) \vee (b[2, 0] = b[1, 1] = b[0, 2] = c)$

Question 8 [5 marks] For this question you will implement in Java an ADT for a 1D sequence of real numbers. We want to take the mean of the numbers in the sequence, but as the following webpage shows, there are several different algorithms for doing this: `https://en.wikipedia.org/wiki/Generalized_mean`

Given that there are different options, we will use the strategy design pattern, as illustrated in the following UML diagram:

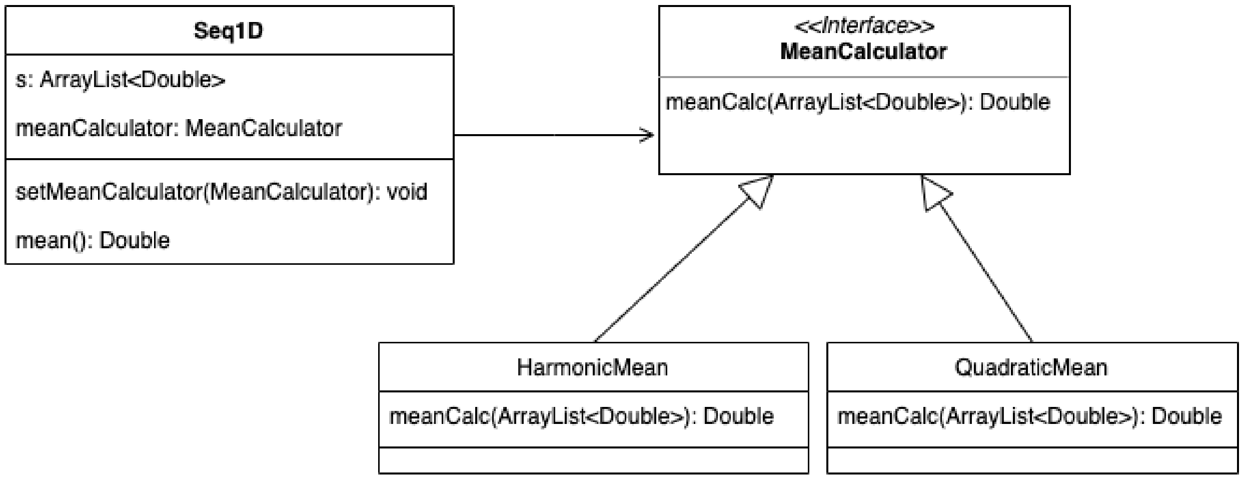
Seq1D
s: ArrayList<Double>
meanCalculator: MeanCalculator
setMeanCalculator(MeanCalculator): void
mean(): Double
<<Interface>>
MeanCalculator
meanCalc(ArrayList<Double>): Double
HarmonicMean
meanCalc(ArrayList<Double>): Double
QuadraticMean
meanCalc(ArrayList<Double>): Double

Figure 1: UML Class Diagram for Seq1D with Mean Function, using Strategy Pattern

You will need to fill in the following blank files: `MeanCalculator.java`, `HarmonicMean.java`, `QuadraticMean.java`, and `Seq1D.java`. Two testing files are also provided: `Expt.java` and `TestSeq1D.java`. The file `Expt.java` is pre-populated with some simple experiments to help you see the interface in use, and do some initial testing. You are free to add to this file to experiment with your work, but the file itself isn't graded. The `TestSeq1D.java` is also not graded. However, you may want to create test cases to improve your confidence in your solution. The stubs of the necessary files are already available in your `src` folder. The code will automatically be imported into this document when the `tex` file is compiled. You should use the provided Makefile to test your code. You will NOT need to modify the Makefile. The given Makefile will work for `make test`, without errors, from the initial state of your repo. The `make expt` rule will also work, because all lines of code have been commented out. Uncomment lines as you complete work on each part of the modules relevant to those lines in `Expt.java` file. As usual, the final test is whether the code runs on mills. You do not need to worry about doxygen comments.

Any exceptions in the specification have names identical to the expected Java exceptions; your code should use exactly the exception names as given in the spec.

Remember, your code needs to implement the given specification so that the interface behaves as specified. This does NOT mean that the local functions need to all be implemented, or that the types used internally to the spec need to be implemented exactly as given. If you do implement any local functions, please make them private. The real type in the MIS should be implemented by `Double` (capital D) in Java.

[Complete Java code to match the following specification. —SS]

Mean Calculator Interface Module

Interface Module

MeanCalculator

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
meanCalc	seq of $\mathbb{R}$	$\mathbb{R}$	

Considerations

meanCalc calculates the mean (a real value) from a given sequence of reals. The order of the entries in the sequence does not matter.

Harmonic Mean Calculation

Template Module inherits MeanCalculator

HarmonicMean

Uses

MeanCalculator

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
meanCalc	seq of $\mathbb{R}$	$\mathbb{R}$	

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

meanCalc(v)

- output: $out := \frac{|x|}{+(x:\mathbb{R}|x\in v:1/x)}$
- exception: none

Quadratic Mean Calculation

Template Module inherits MeanCalculator

QuadraticMean

Uses

MeanCalculator

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
meanCalc	seq of $\mathbb{R}$	$\mathbb{R}$	

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

meanCalc(v)

- output: $out := \sqrt{\frac{+(x:\mathbb{R}|x\in v:x^2)}{|x|}}$
- exception: none

Seq1D Module

Template Module

Seq1D

Uses

MeanCalculator

Syntax

Exported Types

Seq1D = ?

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Seq1D	seq of $\mathbb{R}$, MeanCalculator	Seq1D	IllegalArgumentException
setMaxCalculator	MaxCalculator		
mean		$\mathbb{R}$	

Semantics

State Variables

s: seq of $\mathbb{R}$
meanCalculator: MeanCalculator

State Invariant

None

Assumptions

- The Seq1D constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once. All real numbers provided to the constructor will be zero or positive.

Access Routine Semantics

new Seq1D(x, m):

- transition: $s, \text{meanCalculator} := x, m$
- output: $out := self$
- exception: $exc := (|x| = 0 \Rightarrow \text{IllegalArgumentException})$

setMeanCalculator(m):

- transition: $\text{meanCalculator} := m$
- exception: none

mean():

- output: $out := \text{meanCalculator.meanCalc}()$
- exception: none

Code for MeanCalculator.java

```
package src;

import java.util.ArrayList;

public interface MeanCalculator {
    public double meanCalc(ArrayList<Double> v);
}
```

Code for HarmonicMean.java

```java
package src;

import java.util.ArrayList;

public class HarmonicMean implements MeanCalculator {
    public double meanCalc(ArrayList<Double> v) {
        double sum = 0;
        for (int x = 0; x < v.size(); x++) {
            sum = sum + (1/v.get(x));
          }
        return v.size()/sum;
    }
}
```

Code for QuadraticMean.java

```
package src;

import java.util.ArrayList;
import java.lang.Math;

public class QuadraticMean implements MeanCalculator {
    public double meanCalc(ArrayList<Double> v) {
        double sum = 0;
        for (int x = 0; x < v.size(); x++) {
            sum = sum + Math.pow(v.get(x), 2);
        }
        return Math.sqrt(sum/v.size());
    }
}
```

Code for Seq1D.java

```
package src;

import java.util.ArrayList;

public class Seq1D {

    private MeanCalculator meanCalculator;
    private ArrayList<Double> s;

    public Seq1D(ArrayList<Double> x, MeanCalculator m) {
        if (x.size() == 0) {
            throw new IllegalArgumentException("Empty sequence not
               allowed");
        }
        this.s = x;
        this.meanCalculator = m;
    }

    public void setMeanCalculator(MeanCalculator m) {
        meanCalculator = m;
    }

    public double mean() {
        return meanCalculator.meanCalc(s);
    }


}
```

THE END.