# HarmonyMuse

A system for organizing and categorizing musical harmony

# Michael Kramer

# Assignment 1 --1/12/2021 7:08 AM

For this assignment, you will describe and implement release 1 of your term project. You will incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*. You are free to choose a project that interests you but if you prefer, the instructor and your facilitator will be happy to suggest a topic. If you are already an experienced developer, this is an opportunity to build a challenging application (check with your facilitator if it requires significant API’s) or discuss research with the instructor. It is OK to name a project with much more scope than you can accomplish in the course (as in the example above): we will not hold you to completing everything associated with it. What we do expect is that you specify and implement a set of do-able requirements within such scope.

Submit this completed Word document. Replace as indicated. Please observe and retain the gray text. Your materials—in black 12-point Times New Roman—should not exceed 5 pages excluding references, figures, and appendices. Use the Appendix sections for additional material if you need to. These will be read only on an as-needed basis.

We want you to develop in Eclipse preferably or else IntelliJ (talk to your facilitator about exceptions). As you code, use JUnit tests whenever possible but certainly by week 2—package-by-package, class-by-class, and method-by-method, except for trivial methods and those requiring I/O. Use non-Junit classes for testing the latter. Keep the evaluation criteria in mind, listed at the end.

For this assignment, you do not need to read data from a file—you can build all data into the code.

Include a ReadMe file describing where to run the application from, and including necessary execution notes. All JUnit tests will be assumed runnable.

# 1.1 SUMMARY DESCRIPTION *EVALUATION CRITERION (i) APPLIES*

One- or two-paragraph overall description of your proposed term project—half-page (12-point Times New Roman) limit. By the end, term projects will incorporate most of the techniques discussed in the course. To do this, you may need to alter the direction of your project or introduce an additional project in future. You may alter this or even replace it as the semester progresses. You will probably find it useful to use your project acronym.

This project concerns a system for organizing and categorizing harmony (chords, a collection of musical notes). To use this system, called *HarmonyMuse*, a musician inputs either a collection of notes or a chord quality with an accompanying complexity and a context for the sonority of the chord. *HarmonyMuse* analyzes the given input and provides a performance solution based on the provided context.

## 1.2 I/O EXAMPLE FROM *PROJECTED* COMPLETED PROJECT *EVALUATION CRITERION (i) APPLIES*

Provide an example of projected *concrete* output for designated input. You will not be held to fulfilling exactly this—it is just explanatory at this point, to indicate where your project is going. We recognize that project direction and details will change as the term progress. This section refers to the project as a whole, not just to what you will produce this week, so we can gain an idea of what you have in mind overall.

**Musician:**

C, E, G

**System:**

* Simple Possibilities:
  + C major triad (root position)
  + E minor b6 (omit 5th)
  + G6sus4
* Would you like to see complex possibilities?

**Musician:**

Y

**System:**

* Complex Possibilities:
  + Db maj7(#11, #9)
  + D7sus4
  + Eb 13(b9)
  + Fmaj9
  + F#7alt
  + Abmaj7(#5)
  + Amin7
  + Bbmaj13(#11)
  + Bphrygian
* What is your context? type “piano” or “guitar”

**Musician:**

piano

**System:**

<Outputs PDF of music notation on grand staff including all possibilities listed above>

<https://lilypond.org/doc/v2.22/Documentation/learning/index#top>

## 1.3 REQUIREMENTS IMPLEMENTED IN THIS RELEASE *EVALUATION CRITERION (ii) APPLIES*

Supply [functional requirements](https://docs.google.com/document/d/1eU7eINLDxmrf793D4OF2yGT4ry_SW3GQGoVDYzecGHc/edit?usp=sharing) statement that you accomplished for this assignment, i.e., functionality that the application provides for the user. Please state requirement in declarative form, as illustrated in the examples, because here we want to know the functionality intended (*what*, not *how*). For example, the following is *not* a proper functional requirement: *TicTac will have a class for O’s and a class for X’s.* It is common to mistake design elements like this for requirements. To get started, state what the application will accept as input, like requirement 1.3.1 below.

Keep in mind that the implementation of your requirements will incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*; that will probably influence the requirements you choose to implement in this assignment. The example material supplied should be deleted before you submit.

### 1.3.1 Create and Classify Chord Objects from raw input

*HarmonyMuse* should accept raw data (currently hard-coded String arrays, accepting Midi data arrays is the eventual goal in addition to String arrays), build a Chord object and classify that object. At this stage the classification process is arbitrary, later to be implemented dynamically by the system.

### 1.3.2 Report Chord classification and associated attributes

Each chord object should know its classification (i.e. triad, seventh chord etc.), quality (i.e. major, minor etc.) and its degrees (i.e. root, third, fifth, etc.) and the intervals it contains (i.e. major third, perfect fifth) and report all of this information to the console.

### 1.3.3 Create and Classify Note Objects from raw input

Given raw String (currently hard-coded) input *HarmonyMuse* should clean data and create a Note Object such that Interval Objects may be accurately determined. This includes accounting for enharmonic (two notes with different spelling that sound the same frequency i.e. “b#” and “c”) pitch spelling possibilities.

### 1.3.3 Create and Classify Interval Objects from raw input

Perhaps the most important data structure to this system is the Interval as the basic unit of musical harmony. After creating two Note Objects, *HarmonyMuse* should determine the interval quality (and enharmonic intervals where applicable, two intervals with the same distance in half-steps but different spellings, i.e. “minor third” and “augmented second”).

## 1.4 ILLUSTRATIVE OUTPUT FROM IMPLEMENTATION *EVALUATION CRITERION (ii) APPLIES*

### Provide illustrative output from your implemented application (so far) showing that the requirements have been met. Explain what class.method(s) produce it.

The following is produced by

* the respective types overridden *toString()* methods
* the respective types *getRoot(), getThird. getFifth()* and *getSeventh()* methods (where applicable)
* A substantial amount of thought, work and design went into the non-trivial task of determining how to successfully implement the output that results from *Chord.getIntervals(),* see the Note class and the Interval class
  + - In the Note class constructor the active attributes are name, intValue, and enharmonics
      * A Map structure of an Integer object and an ArrayList (containing all of the enharmonic spellings for a given pitch class) is built in the buildNotesMap() function and this map is used to determine the enharmonics attribute for any given note.
    - The Interval class attributes firstly track the number of half steps in each interval (within one octave – this is a natural compression function used for classification)
      * The indices of the elements of the arrays intervalQualities and enharmonicIntervals correspond to the number of half steps in the respective interval for which these names are representations (with a few exceptions thoroughly noted in code comments)
    - The distance attribute is acquired via the .setDistance() method which calls the .distance() method of the CircularlyLinkedList class in the package (code modified from sources noted in the file, the .distance() function is self-authored).
      * The .distance() function determines the interval family (i.e. how many “steps” in the musical alphabet the bottom note name is from the top).
        + The Circularly linked list is an optimal data structure here as the musical alphabet wraps around like so (a, b, c, d, e, f, g -> a…) and the distance from a to c (3) is the same as from g to b (3, wrapping around).
      * Once the distance is set the appropriate interval can be determined from the individual notes intValues producing the intValue attribute for the Interval class.
        + With the intValue of the Interval Class, the number of half steps in each “proper” interval (the first 13 Interval class attributes) and the two arrays, intervalQualities and enharmonicQualities (recall each element index corresponds to the number of half steps in the name string representation at that index) we can assign the Interval attribute quality (and enharmonic when applicable) as seen in the setQuality() function.

Text

Description automatically generated

## 1.5 YOUR DIRECTORY

### Show a screenshot of your directory. This should include a parallel directory of JUnit tests where possible—package-by-package, class-by-class, and method-by-method, except for trivial ones.

Text

Description automatically generated

Please note:

The Interfaces Extensions, AlteredExtensions and Voicing are rough sketches and essentially are placeholder “TODO” at this current stage of development.

* Extensions builds the ninth, eleventh and thirteenth on-top of SeventhChord
* AlteredExtensions does something similar but allows for building both the FlatNinth and the SharpNinth, a common requirement for Dominant Seventh chords in practice.

## 1.6 TECHNIQUES IMPLEMENTED *EVALUATION CRITERION (iii) APPLIES*

Your implementation should include *inheritance*, *polymorphism*, and *either an abstract class or interface* at least once, and in a manner that is useful to your application. Explain where and how you applied these, using the headings below.

### 1.6.1 Class model and Sequence Diagram

Identify where you included *inheritance*, *polymorphism*, and *abstract classes* or *interfaces* in your class model. Make classes and members *static* or not as per their intended usage. To do this use tools (e.g., Visio and Lucidchart), PowerPoint, or a combine models as in [this example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to cut and paste from). Insert indications in red (as in [this example](https://docs.google.com/spreadsheets/d/1ZvkerE9FkWHWwVGdzuy7YMBU6oBMFGZbA4sotFETs8Y/edit?usp=sharing)) to show where the three features below apply. Diagram

Description automatically generated

Class Model for HarmonyMuse (Above):

* Chord is Abstract – polymorphism occurs in the versions getRoot()
* ChordBuilder, MajorTriad, MinorTriad and DominantSeventhChord inherit from Chord.
* ChordBuilder should serve as the intermediary between input to the system and the creation and classification of any resultant chord object – cleaning and analyzing data as necessary.
* MajorTriad and MinorTriad are both additionally dependencies of the Triad Interface
* DominantSeventhChord is additionally a dependency of the SeventhChord Interface – downcasting is shown with getSeventh()
* Interval aggregates with a many to one relationship to chord
* Note aggregates to Interval with a 2 to 1 relationship and further to Chord with a 3 (or greater) to 1 relationship
* Voicing (not shown) is an interface that will voice (arrange and draw) chords that utilize it as a dependency

Diagram

Description automatically generated

Sequence Diagram 1) Use Case: Create and classify chord from raw input (above)

Diagram

Description automatically generated with low confidence

Sequence Diagram 2) Use Case: Report class associated attributes (above)

### 1.6.2 Code showing an abstract class or interface

Show the relevant code (only) and explain why an abstract class or interface is appropriate here. It should be clear where the code is located (class and method).

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Abstract Class Chord (Above)

* A chord is by nature one of the most fundamental abstractions in conceiving of harmony in modern music
  + It is simply a collection of notes regardless of classification
    - It deserves noting, this collection should be greater than two notes
      * Two notes constitute an Interval
  + Every chord regardless of specific classification needs to be contextualized with a root (line 42)
    - Although the design for this system should be flexible enough to be able to examine any of the input data as the root and classify the resultant chord accordingly
      * We want the ability and flexibility to handle and contextualize all permutations of a given data

Text

Description automatically generated

Interface Triad (above)

* A Triad is an abstraction on a three-note chord
  + Not every three-note chord is a triad – these are very specific arrangements of notes built on the Interval of a third (the basis for Tertian, third based harmony, the harmony of the Western world).
    - There are 4 specific qualities of triads
      * Major
      * Minor
      * Diminished
      * Augmented
    - Each of the 4 qualities have the following common attributes
      * A root (method inherited from Chord)
      * A third (method to be implemented from Triad Interface dependency)
      * A fifth (method to be implemented from Triad Interface dependency)

Text

Description automatically generated

Interface SeventhChord (above)

* A Seventh chord, in Tertian harmony (based on the Interval of a third) is simply adding one more third to the base Triad (one of the 4 qualities)
  + The DominantSeventhChord class thusly demonstrates multiple inheritance from the abstract Chord class, and the Interfaces Triad and SeventhChord

### 1.6.3 Code showing polymorphism

Show the relevant code (only) and explain why *polymorphism* is appropriate here. Recall that polymorphism is implemented in one of two ways – overriding methods in subclasses or overloading methods in the same class where the method signatures are different – and allowing the language runtime to dynamically invoke the correct method. It should be clear where the code is located (class and method).

The method *getRoot()* in the following for each loop illustrates polymorphic calls to objects of types MajorTriad, MinorTriad, and DominantSeventhChord, each of whom have inherited and overridden this function from the abstract Chord class and additionally have been stored in variables of names descriptive of their respective types. The below code snippet additionally demonstrates downcasting (explained below in Section 1.6.4).

Text

Description automatically generated

### 1.6.4 Code showing upcasting or downcasting

Show the relevant code (only) and explain why upcasting or downcasting is appropriate here. It should be clear where the code is located (class and method).

Because a DominantSeventhChord type object is the only object in the Chord array with the method *getSeventh()* it is necessary to **downcast** in order to reach the DominantSeventhChord.seventh attribute of the appropriate object (determined by the instanceof query at line 64).

Text

Description automatically generated

## 1.7 YOUR CODE

Unless your facilitator requests another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request an alternative means.

Project and Test folders are zipped in a folder named metcs622\_Assignment1\_mgkramer.zip

\*\*\*Please Note\*\*\*

The directory referenced in section [1.5](file:///C:\Users\Michael%20Kramer\Downloads\metcs622_Assignment1_mgkramer.docx#_1.5_YOUR_DIRECTORY) in this document belongs in the src project directory (in IntelliJ this is a dark blue folder)

TO RUN TESTS:

The directory in the zipped folder named ***HarmonyMuseTests*** belongs in the test root directory (in IntelliJ this is a green folder traditionally named test) and needs to be renamed to ***HarmonyMuse*** as shown below:

Graphical user interface, text

Description automatically generated

## 1.8 EVALUATION OF ASSIGNMENT 1



## Appendix 1 (if needed; should be referenced above, and will be read as-needed only)

## Appendix 2 (if needed; should be referenced above, and will be read as-needed only)

--1/20/2021 9:07 AM

# Assignment 2

Implement the next release of your term project. You will incorporate exception handling and file I/O, as specified below.

Submit this completed Word document, including your name within the file name, observing and retaining the gray text like this. Retain the headings or edit them as indicated. Your Assignment 2 materials—in black 12-point Times New Roman—should not exceed 5 pages excluding references, figures, and appendices. Use the Appendix sections for additional material if you need to. These will be read on an as-needed basis.

As you code, use JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones and ones requiring I/O that cannot be unit tested. Use non-Junit classes for testing the latter, as in the worked example.

Include a ReadMe file describing where to run the application from, and including notes as necessary (not more).

Keep the evaluation criteria in mind, listed at the end and referred to in the headings.

## 2.1 SUMMARY DESCRIPTION, UPDATED AS APPLICABLE THIS

## *EVALUATION CRITERION (i) APPLIES*

One- or two-paragraph overall description of your whole proposed term project. Edit your last description as needed.

This project concerns a system for organizing and categorizing harmony (chords, a collection of musical notes). To use this system, called *HarmonyMuse*, a musician inputs either a collection of notes or a chord quality with an accompanying complexity and a context for the sonority of the chord. *HarmonyMuse* analyzes the given input and provides a performance solution based on the provided context.

## 2.2 I/O EXAMPLE FROM PROJECTED COMPLETED PROJECT, UPDATED AS APPLICABLE *EVALUATION CRITERION (i) APPLIES*

Provide an example of projected *concrete* output for example input, indicating how users will interact with your application. You will not be held to fulfilling exactly this—it intended to help us understand the probable direction of your application as a whole

As with section 2.1, I’m still happy with the I/O example from Assignment1 1.2 however, I’m still in the process of refining this at this stage. Below is the example from 1.2:

**Musician:**

C, E, G

**System:**

* Simple Possibilities:
  + C major triad (root position)
  + E minor b6 (omit 5th)
  + G6sus4
* Would you like to see complex possibilities?

**Musician:**

Y

**System:**

* Complex Possibilities:
  + Db maj7(#11, #9)
  + D7sus4
  + Eb 13(b9)
  + Fmaj9
  + F#7alt
  + Abmaj7(#5)
  + Amin7
  + Bbmaj13(#11)
  + Bphrygian
* What is your context? type “piano” or “guitar”

**Musician:**

piano

**System:**

<Outputs PDF of music notation on grand staff including all possibilities listed above>

<https://lilypond.org/doc/v2.22/Documentation/learning/index#top>

## 2.3 REQUIREMENTS IMPLEMENTED IN THIS RELEASE NOT IMPLEMENTED BEFORE *EVALUATION CRITERION (i) APPLIES*

### 2.3.1 Implement an MVC Design Pattern of Data Flow for HarmonyMuse THIS

Create the necessary classes to separate the Model (business logic, state of the application and retrieval and storage of data) from the View (for now a command line application that prints data to the console) and the Controller (responsible for calling the model to get and set the data and updating the view based on these state changes --- currently, controller = Main.java, view = ChordEntryView.java, and the remaining classes are model classes).

### 2.3.2 Dynamically Classify a Chord given raw data

Based on system cleaned and validated raw data (currently in the form of string arrays written to and read from a file) HarmonyMuse should dynamically classify Chords (currently only Closed Position Triads of all inversions and qualities).

### 2.3.3 Accept User Input, Validate and write to database

User input is currently accepted at the command line through only one view class at this stage which is called by the controller. The view only accepts clean data that can be handled properly by the systems current business logic and notifies the user of false input with appropriate customized Exceptions. Currently, the database is modeled as a .txt file, this is to be expanded in subsequent iterations.

### 2.3.4 Read Input from a database, classify and deliver analysis to the user

HarmonyMuse should read raw data from the database, currently modeled as a text file, classify this data and deliver the classified data to the view such that it can be presented to the end user.

### 2.3.5 Your title replaces this (add more requirements as needed)

Your content response replaces this

## 2.4 I/O EVIDENCE THAT THE ABOVE FUNCTIONALITY WAS ACHIEVED

## *EVALUATION CRITERION (ii) APPLIES*

## This typically consists of screen shots of input and output, together with text explaining their context.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

## 2.5 YOUR DIRECTORY

Show a screenshot of your directory. This should include a parallel directory of JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones.

Your response replaces this.

## 2.6 TECHNIQUES IMPLEMENTED *EVALUATION CRITERION (iii) APPLIES*

Your implementation should exploit *file IO* and *exceptions* at least once, in as natural a manner as possible. Using the headings below, explain where and how you applied these.

### 2.6.1 Class model and Sequence Diagram

Indicate clearly in your class model where you applied file IO and exception handling, including a user-defined exception if possible. “Enforce what you intend.” For example, make classes and members *static* or not as per their intended usage. To do this use tools, PowerPoint, or combine models as in [this RUML example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to copy, cut and paste from). Insert indications in red (as in the example) to show where the three features below apply.

Your response replaces this.

### 2.6.2 Code showing *file I/O THIS*

Show the relevant code (only). It should be clear where the code is located (class and method). Specify nontrivial methods with pre- and postconditions (and examples if this clarifies).

#### Example WriteToFile.Java

package HarmonyMuse;  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.Formatter;  
  
  
public class WriteToFile {  
  
 public static void clearFile(){  
 try {  
 File file = new File("data\\input.txt");  
 FileWriter fileWriter = new FileWriter(file);  
 fileWriter.flush();  
 }catch (IOException e){  
 System.*out*.println("Error flushing file");  
 e.printStackTrace();  
 }  
  
 }  
 public static void writeToFile(String[] rawInput){  
  
 try{  
 File file = new File("data\\input.txt");  
 FileWriter fileWriter = new FileWriter(file, true);  
 //BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);  
 Formatter outfile = new Formatter(fileWriter);  
 for(int i = 0; i < rawInput.length; i++){  
 System.*out*.println(rawInput[i]);  
 outfile.format("%s ", rawInput[i]);  
 //bufferedWriter.write(rawInput[i]);  
 }  
 outfile.format("%n");  
 outfile.close();  
 //bufferedWriter.close();  
 }catch (IOException e){  
 System.*out*.println("Error writing to file");  
 e.printStackTrace();  
 }  
  
 }  
}

#### Example ReadFromFile.Java

package HarmonyMuse;  
  
import java.io.IOException;  
import java.nio.file.Paths;  
import java.util.ArrayList;  
import java.util.NoSuchElementException;  
import java.util.Scanner;  
import java.util.StringTokenizer;  
  
public class ReadFromFile {  
  
 public static ArrayList<Chord> readFile(){  
 ArrayList<Chord> chordsOnFile = new ArrayList<>(0);  
 ArrayList<String> container = new ArrayList<>(0);  
 // open input.txt, read its contents and close the file  
 try(Scanner input = new Scanner(Paths.*get*("data\\input.txt"))) {  
 // read record from file  
 while (input.hasNext()) { // while there is more to read  
 StringTokenizer st = new StringTokenizer(input.nextLine());  
 while (st.hasMoreTokens()){  
 container.add(st.nextToken());  
 }  
 String[] data = new String[container.size()];  
 for(int j = 0; j < container.size(); j++) {  
 data[j] = container.get(j);  
 }  
 Note[] notes = new Note[data.length];  
 for (int i = 0; i < data.length; i++) {  
 try {  
 notes[i] = new Note(data[i]);  
 } catch (IllegalArgumentException e) {  
 System.*out*.println(e);  
 }  
 }  
 ChordBuilder rawData = new ChordBuilder(notes); // build a chord from input data  
 Chord chord = rawData.classify(rawData);  
 chordsOnFile.add(chord);  
 container.clear(); // clear for the next chord  
 }  
 // try(with recources IMPLICITLY calls input.close()  
 } catch (IOException | NoSuchElementException |  
 IllegalStateException e) {  
 e.printStackTrace();  
 }  
 return chordsOnFile;  
 }  
}

### 2.6.3 Code showing *exception*, preferably a user-defined exception

Show the relevant code (only) and explain why *exceptions* are appropriate here. It should be clear where the code is located (class and method). Specify nontrivial methods with pre- and postconditions (and examples if this clarifies).

#### Example InvalidNoteException.Java

package HarmonyMuse;  
  
public class InvalidNoteException extends Exception{  
  
 public InvalidNoteException(String str){  
 super(str);  
 }  
}

##### is thrown by the ChordEntryView.java class, getNotes() function:

public static String[] getNotes(int num\_notes, String[] notes) throws InvalidNoteException{  
 boolean goodNote = false;  
 Scanner sc = new Scanner(System.*in*);  
 for(int i = 0; i < num\_notes; i++){  
 System.*out*.println("Please enter note #" + (i + 1));  
 String next\_note = sc.nextLine();  
 if(next\_note.toString().matches("[a-gA-G]")){  
 goodNote = true;  
 notes[i] = next\_note.toString();  
 }  
 else if(next\_note.toString().matches("[a-gA-G]+[#-]")){  
 goodNote = true;  
 notes[i] = next\_note.toString();  
 }  
 else if(next\_note.toString().matches("[a-gA-G]##")){  
 goodNote = true;  
 notes[i] = next\_note.toString();  
 }  
 else if(next\_note.toString().matches("[a-gA-G]--")){  
 goodNote = true;  
 notes[i] = next\_note.toString();  
 }  
 else throw new InvalidNoteException("invalid note entry");  
 }  
 return notes;  
}

###### and caught in the controller Main.java

package HarmonyMuse;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
  
public class Main {  
 public static void main(String[] args) {  
 ChordEntryView.*welcome*();  
 WriteToFile.*clearFile*(); // reset file, this is temporary  
 int num\_notes = -1;  
 boolean moreInput = true;  
 while (moreInput) {  
 while (num\_notes < 0) {  
 num\_notes = ChordEntryView.*getNumNotes*();  
 }  
 System.*out*.println(num\_notes);  
 String[] chordInput = new String[num\_notes];  
 boolean goodInput = false;  
 while (!goodInput) {  
 try {  
 String[] rawInput = ChordEntryView.*getNotes*(num\_notes, chordInput);  
 System.*out*.println(Arrays.*toString*(rawInput));  
 goodInput = true;  
 WriteToFile.*writeToFile*(rawInput);  
  
 } catch (InvalidNoteException e) {  
 System.*out*.println(e);  
 }  
 }  
 moreInput = ChordEntryView.*moreChords*();  
 }  
 ArrayList<Chord> chordsOnFile = ReadFromFile.*readFile*();  
 ChordEntryView.*displayChordsOnFile*(chordsOnFile);  
 }  
}

#### Example InvalidTriadException.java

package HarmonyMuse;  
  
public class InvalidTriadException extends Exception{  
  
 public InvalidTriadException(String str){  
 super(str);  
 }  
}

##### is thrown in TriadClassifier.java in the classifyTriad() function

public Chord classifyTriad(ChordBuilder chordBuilder, Interval bottom\_mid, Interval bottom\_top) throws InvalidTriadException{  
 Chord classified = chordBuilder; // re-point chordBuilder to classify it  
 /\*  
 Initialize chordBuilder to each possible quality and let below conditions  
 handle which gets pointed at classified  
 \*/  
 DiminishedTriad dimTriad = new DiminishedTriad(chordBuilder);  
 MinorTriad minTriad = new MinorTriad(chordBuilder);  
 MajorTriad majTriad = new MajorTriad(chordBuilder);  
 AugmentedTriad augTriad = new AugmentedTriad(chordBuilder);  
  
 /// CLOSED POSITION CHECKS ///  
 if(bottom\_mid.getIntValue() == 3){ // 4 possibilities that have minor third on bottom  
 if(bottom\_top.getIntValue() == 6){ // root position diminished triad (c, e-, g-)  
 dimTriad.setQuality(triadQualities[0]);  
 dimTriad.setInversion(chordBuilder.getInversions()[0]);  
 classified = setRootPosClosed(dimTriad, chordBuilder);  
  
 }  
 else if(bottom\_top.getIntValue() == 7){ // root position minor triad (c, e-, g)  
 minTriad.setQuality(triadQualities[1]);  
 minTriad.setInversion(chordBuilder.getInversions()[0]);  
 classified = setRootPosClosed(minTriad, chordBuilder);  
 }  
 else if(bottom\_top.getIntValue() == 8){ // first inversion major triad (e, g, c)  
 majTriad.setQuality(triadQualities[2]);  
 majTriad.setInversion(chordBuilder.getInversions()[1]);  
 setFirstInvClosed(majTriad, chordBuilder);  
 classified = majTriad;  
 }  
 else if(bottom\_top.getIntValue() == 9){ // first inversion diminished triad (e-, g-, c)  
 dimTriad.setQuality(triadQualities[0]);  
 dimTriad.setInversion(chordBuilder.getInversions()[1]);  
 classified = setFirstInvClosed(dimTriad, chordBuilder);  
 }  
 }  
 else if(bottom\_mid.getIntValue() == 4){ // 3 possibilities have major third on bottom  
 if(bottom\_top.getIntValue() == 7){ // major triad root position (c, e, g)  
 majTriad.setQuality(triadQualities[2]);  
 majTriad.setInversion(chordBuilder.getInversions()[0]);  
 classified = setRootPosClosed(majTriad, chordBuilder);  
 }  
 else if(bottom\_top.getIntValue() == 8){  
 /\* augmented triad symmetrical shape (root determined by context) closed position  
 (c, e, g#)  
 \*/  
 augTriad.setQuality(triadQualities[3]);  
 augTriad.setInversion(chordBuilder.getInversions()[0]);  
 classified = setRootPosClosed(augTriad, chordBuilder);  
 }  
 if(bottom\_top.getIntValue() == 9){ // minor triad first inversion (e-, g, c)  
 minTriad.setQuality(triadQualities[1]);  
 minTriad.setInversion(chordBuilder.getInversions()[1]);  
 classified = setFirstInvClosed(minTriad, chordBuilder);  
 }  
 }  
 else if(bottom\_mid.getIntValue() == 5){ // 2 possibilities have perfect fourth on bottom  
 if(bottom\_top.getIntValue() == 8){ // minor triad second inversion (g, c, e-)  
 minTriad.setQuality(triadQualities[1]);  
 minTriad.setInversion(chordBuilder.getInversions()[2]);  
 classified = setSecondInvClosed(minTriad, chordBuilder);  
 }  
 else if(bottom\_top.getIntValue() == 9){ // major triad second inversion (g, c, e)  
 majTriad.setQuality(triadQualities[2]);  
 majTriad.setInversion(chordBuilder.getInversions()[2]);  
 classified = setSecondInvClosed(majTriad, chordBuilder);  
 }  
 }  
 else if(bottom\_mid.getIntValue() == 6){ // two possibilities have tritone on bottom  
 if(bottom\_top.getIntValue() == 9){ // second inversion diminished triad (g-, c, e-)  
 dimTriad.setQuality(triadQualities[0]);  
 dimTriad.setInversion(chordBuilder.getInversions()[2]);  
 classified = setSecondInvClosed(dimTriad, chordBuilder);  
 }  
 /// END CLOSED POSITION CHECKS ///  
  
 /// OPEN POSITION CHECKS ///  
 else if(bottom\_top.getIntValue() == 3){ // root position diminished triad (c, g-, e-)  
 dimTriad.setQuality(triadQualities[0]);  
 dimTriad.setInversion(chordBuilder.getInversions()[0]);  
 classified = setRootPosOpen(dimTriad, chordBuilder);  
 }  
 /\*  
 *TODO continue classifying open triads*  
 \*/  
 }  
 // if classified hasn't been re-assigned triad by now classification failed  
 if(classified instanceof ChordBuilder){  
 throw new InvalidTriadException(classified + " is not a valid triad");  
 }  
 return classified;  
}

###### and caught in ChordBuilder.java classify() function

public Chord classify(ChordBuilder chordBuilder){  
 Chord classified = chordBuilder; // container to return classified chord  
 ArrayList<Interval> intervals = chordBuilder.getIntervals();  
  
 if(intervals.size() == 2){ // we've found three note chord, let's try to classify it as a triad  
 Interval bottom\_mid = intervals.get(0);  
 Interval bottom\_top = intervals.get(1);  
 try {  
 TriadClassifier triadClassifier = new TriadClassifier();  
 classified = triadClassifier.classifyTriad(chordBuilder, bottom\_mid, bottom\_top);  
 } catch (InvalidTriadException e){  
 System.*out*.println(e);  
 /\*  
 *TODO define another classifier threeNoteNonTriad and threeNoteCluster and call*  
 \*/  
 }  
 }  
 return classified;  
}

## 2.7 YOUR CODE

Unless your facilitator requests or allows another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request an exception. Specify nontrivial methods with pre- and postconditions (and examples if this clarifies). For excellent work, specify the class invariants.

<Your response here>

## 2.8 EVALUATION OF ASSIGNMENT 2



## Appendix 1 (will be read as-needed only—add more as necessary)