

# Social Network Analysis and Visualization Support for the Analysis of Music Collaborations

CSC 2310 – Fall 2022

## Abstract

*In 1927, a group of country music artists were called by Ralph Peer to record old-time music. The Bristol Sessions have been touted as the “Big Bang” of Country Music and had a big impact on the future of the genre. While country music scholars have studied the sessions extensively, the sessions have become popularized with the release of the Ken Burns Country Music documentary in 2019. In this project, we will use the Discogs API to gather and use data about artists to perform a graph-based social network analysis to visualize and synthesize new insights about the artists that recorded and collaborated on the recordings of the original Bristol Sessions.*

Data is critical for making both short-term and long-term decisions. Data scientists are often called upon to create tools for providing insight into the knowledge embedded in data by fusing information from various sources into a single cogent collection of information. For instance, consider the following example taken from the Discogs ([api.discogs.com](https://api.discogs.com)) API:

```
{
  "name": "Jimmie Rodgers",
  "id": 269365,
  "resource_url": "https://api.discogs.com/artists/269365",
  "uri": "https://www.discogs.com/artist/269365-Jimmie-Rodgers",
  "releases_url": "https://api.discogs.com/artists/269365/releases",
  "realname": "James Charles Rodgers",
  "profile":
    "American country singer, guitarist and songwriter (born September 08, 1897 in Meridian, Mississippi - died May 26, 1933 in New York City, New York).\r\n\r\nIn the early 20th century known most widely for his rhythmic yodeling. Among the first country music superstars and pioneers, Rodgers was also known as The Singing Brakeman, The Blue Yodeler, and The Father of Country Music.\r\n\r\nInducted into Songwriters Hall of Fame in 1970.\r\n\r\nInducted into Rock And Roll Hall of Fame in 1986 (Early Influence). \r\n\r\nInducted into Country Music Hall of Fame in 1961.\r\n\r\n\r\n[b]Note! Please be careful when assigning credits to this artist.[/b] Other artists with a same or similar name exist (list not complete):\r\n\r\n- [a=Jimmy Rogers] - Chicago blues singer and guitarist\r\n\r\n- [a=Jimmie Rogers] - multi-instrumentalist (flute, didgeridoo, jew's harp ...)\r\n\r\n- [a=Jimmie Rogers (2)] - rock n' roll songwriter\r\n\r\n- [a=Jimmie Rodgers (2)] - folk/pop singer (born 1933)",
  "data_quality": "Correct"
}
```

From this and other associated information, enough data can be acquired that can be used to identify all of the artists that collaborated with an artist (a 1<sup>st</sup>-order degree of separation) and the artists that collaborated with the 1<sup>st</sup>-order artists (i.e., the 2<sup>nd</sup>-order degree of separation from the original artist. By doing some elementary data mining, the data can be assembled to form a graph depicting the extent of the collaborations between different artists, as shown in Figure 1.

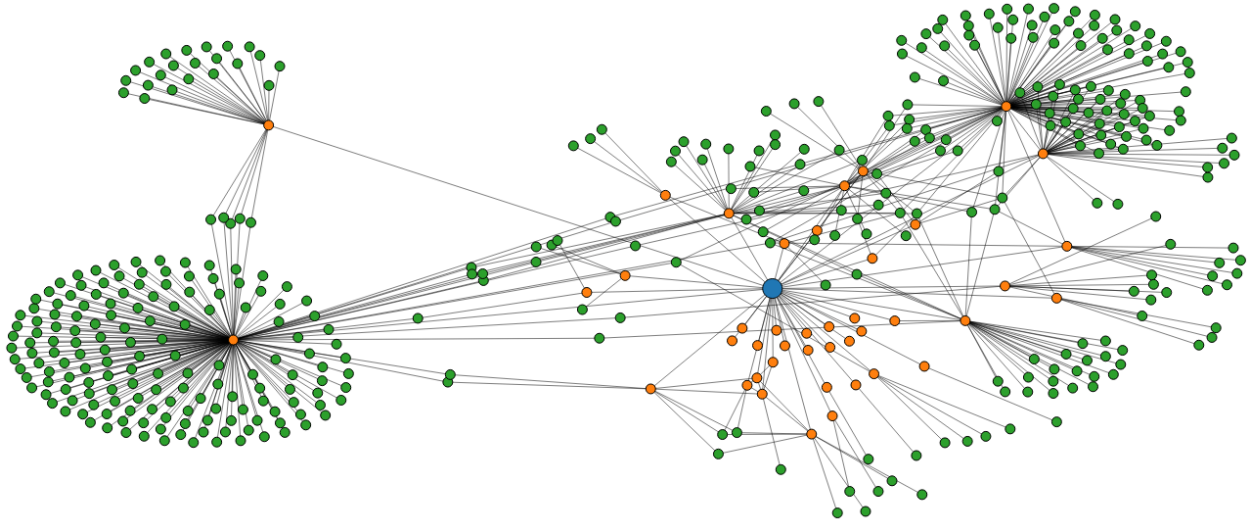


Figure 1 Collaboration Graph of Jimmie Rodgers

As you begin the brainstorming process, recall that as a “developer” you have biases, and that you need to try adopt the mindset of different kinds of users. You may want to ask yourself questions such as:

- Who are the users?
- Should a user be able to add or modify data? Which type of users can do this?
- How will they use this system? What should the screen designs be to make using the system easy?
- What should a user be able to learn from the display of the data?
- Are there specific analyses that can be performed to identify interesting relationships?
- Are there other data sets that may be of interest?
- How should we incorporate color, sizes, and other visual features to convey information about artists?
- What other information should we display? What does each type of user need to see and do?

In this iteration, you will be doing the following:

- Brainstorm the potential features of the system – for instance, some users might want to add information expand the graph by clicking on a node.
- Specify user stories for each of the user types based on your problem analysis using the user story format of “As a <user>, I want <feature>, so that <reason>”.

As you go through this activity, you should attempt, as much as possible, to avoid thinking about how any of this is going to be implemented. ***The most important aspect of this activity is identifying all of the possible features for this system (noting that identification of all of the possible features does not mean you will have to implement all of the features).***

## Project Phases

The project will be conducted in two major phases: *concept-initiate* and *iteration-construction*. In the *concept-initiate* phase you will identify project requirements, create models, and prep your development environment to support your implementation activities. In the *iteration-construction* phase of the project you will iteratively develop software for the project by adding support according to the requirements developed during the *concept-initiate* phase. The schedule of the phases of the project and the individual iterations contained therein are defined below:

Date (Midnight)	Phase	Description
September 9	Concept Initiate 0	User stories
September 23	Concept Initiate 1	Use case diagrams
October 7	Concept Initiate 2	Initial Design
October 21	Iteration 0	Project environment setup and initial execution demonstration
November 4	Iteration 1	Graph Visualization
November 18	Iteration 2	Graph Algorithms
December 2	Iteration 3	Delivery

### Concept Initiate 0

#### *User Stories*

In the initial phase of the project, you will create user stories based on the description provided while also generating ideas of other potential features for such a system. The user stories that you create should be specified in the following form:

**As a <user type>, I want <desired feature>, so that <outcome>**

You must also include, with each user story, a description of how you would expect to be able to observe this behavior in the system. Your source of information for the user stories you will identify are given in the description provided above as well as any discussion sessions we conduct in class. You may also reach out to faculty or other students to generate ideas on potential stories. **However, the work you turn in must be your own. You may not copy other students' user stories, descriptions, or other similar work products.**

A template for your user stories is provided below.

#### *User Story Template*

Use the following user story template for *concept-initiate 0*. Replicate this as many times as is necessary.

<b>User Story Name:</b> <Short phrase for the user story>	
<b>As a:</b>	<user type>
<b>I want:</b>	<feature>
<b>So that:</b>	<outcome or reason>
<b>Description</b>	<description of how the user story / feature would be observed and tested in the system>

### *Submission*

Submit your user stories as a PDF document only. *Microsoft Word, LibreOffice, or any other native word processing format will not be accepted.*

### *Rubric*

*Completeness:* You will be graded on the completeness of your turn-in. In particular, you must analyze the description and attempt to identify as many features as you can that are relevant for the project. Note, there is no upper bound on the number of user stories you can define, but there is a lower bound **that will not be specified**.

*Correctness:* You will be graded based on your adherence to the format of the user story template and on the accuracy of the user stories with respect to relevance to the described project, including your description of how you might expect to observe the user story / feature in the system.

*Points:* The assignment is worth 20 project points.

## Concept Initiate 1

### *Use Case Diagrams*

In our initial step on this project, we did some work to try to identify user stories for the *Dashboard* system. A baseline set of 23 user stories has been uploaded to the course iLearn site and should be used as a starting point for completing the next activity in the project. You must add additional user stories from your own set produced from the set you created for Concept Initiate 0. For Concept Initiate 1, you will be creating use case diagrams for the *Bristol Social Network* system. The baseline user stories fall into a few categories:

- *Analysis* – Stories that are related to interpretation of graph information
- *Data* – Stories that are related to managing data
- *Interaction* – Stories that are related to the use of the application

As you did in Laboratory 2, you will use GenMyModel to create your use case diagram(s) for the Dashboard system. The user stories that you will use for creating the use cases in the diagram are included as an attachment to the assignment in iLearn. Some things to note about these user stories:

- There are 23 base user stories specified in the reference set across 3 different user types in the baseline set. You must add your own additional user stories (at least 5-10)
- User stories may differ by users with the same feature. These should be represented by different users in the diagram linked to a common use case.
- The use case diagram should be partitioned according to the feature categories specified above. Additional stories may result in additional categories that you should name, if appropriate.

### *Submission*

For your turn-in, you should export the model to an image and copy and paste the image to word or some other word processing suite and generate a PDF. The due date is found in the table found in the Project Phases section found earlier in the document.

### *Rubric*

*Completeness:* You will be graded on the completeness of your turn-in. In particular, your submission must at the very least include the user stories provided in the baseline set. However, it is also expected that you will include other stories in your diagram.

*Correctness:* You will be graded on the correctness of your use case diagrams. In particular, your submission must correctly represent use cases, actors (both interactive and external actors), and partitioning of the stories into appropriate subsystems by category.

*Points:* The assignment is worth 25 project points.

<b>User Story Name:</b> Annotate	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	annotate graphs with notes
<b>So that:</b>	I can add information about an artist
<b>Description</b>	Allows a user to take notes about a given artist

<b>User Story Name:</b> Degree	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	generate vertex degree information
<b>So that:</b>	I can find important nodes in the graph
<b>Description</b>	<a href="https://networkx.org/documentation/stable/reference/algorithms/centrality.html">https://networkx.org/documentation/stable/reference/algorithms/centrality.html</a>

<b>User Story Name:</b> In-Degree	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	generate vertex in-degree
<b>So that:</b>	I can find important nodes in the graph
<b>Description</b>	<a href="https://networkx.org/documentation/stable/reference/algorithms/centrality.html">https://networkx.org/documentation/stable/reference/algorithms/centrality.html</a>

<b>User Story Name:</b> Out-Degree	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	generate vertex out-degree
<b>So that:</b>	I can find important nodes in the graph
<b>Description</b>	<a href="https://networkx.org/documentation/stable/reference/algorithms/centrality.html">https://networkx.org/documentation/stable/reference/algorithms/centrality.html</a>

<b>User Story Name:</b> Centrality measures	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	generate centrality measures
<b>So that:</b>	I can find important nodes in the graph
<b>Description</b>	<a href="https://networkx.org/documentation/stable/reference/algorithms/centrality.html">https://networkx.org/documentation/stable/reference/algorithms/centrality.html</a>

<b>User Story Name:</b> Select nodes	
<b>Category:</b> Analysis	
<b>As a:</b>	Analyst
<b>I want:</b>	select multiple nodes in the graph
<b>So that:</b>	I can analyze whether groups of nodes are special
<b>Description</b>	Need to be able to select multiple nodes (drag a box?)

<b>User Story Name:</b> Upload new data	
<b>Category:</b> Data	
<b>As a:</b>	Data archivist
<b>I want:</b>	upload new data on the fly
<b>So that:</b>	I can add new information to the site on-the-fly
<b>Description</b>	By right clicking a leaf node

<b>User Story Name:</b> Batch upload new data	
<b>Category:</b> Data	
<b>As a:</b>	Data archivist
<b>I want:</b>	batch upload data
<b>So that:</b>	I can add a large amount of data using a file upload
<b>Description</b>	Upload multiple artists at a time

<b>User Story Name:</b> Export	
<b>Category:</b> Data	
<b>As a:</b>	Data archivist
<b>I want:</b>	export to csv
<b>So that:</b>	I can use the data in other tools
<b>Description</b>	Need to define what the export format will be

<b>User Story Name:</b> Select artist	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	select a single artist
<b>So that:</b>	I can build a graph based on that artist's collaborations
<b>Description</b>	Build a graph using just one selected artist

<b>User Story Name:</b> Select multiple artists	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	select multiple artists
<b>So that:</b>	I can build a graph based on multiple artist's collaborators
<b>Description</b>	Build a graph using multiple artists; Need to differentiate between different people

<b>User Story Name:</b> Color nodes	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	view nodes using different colors
<b>So that:</b>	I can differentiate between the role someone had in a recording
<b>Description</b>	Colors determine distance from the originally selected artist

<b>User Story Name:</b> Info on hover	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	see artist info when I hover over a node
<b>So that:</b>	I can see relevant information about the artist on demand
<b>Description</b>	Hover to see artist info

<b>User Story Name:</b> Save	
<b>Category:</b> Save	
<b>As a:</b>	User
<b>I want:</b>	save graphs
<b>So that:</b>	I can share a graph with another user
<b>Description</b>	Save graphs so I we don't have to generate them multiple times

<b>User Story Name:</b> Download	
<b>Category:</b> Save	
<b>As a:</b>	User
<b>I want:</b>	download graphs
<b>So that:</b>	I can put them into other documents
<b>Description</b>	pdf, png, etc.; should be split into different stories by type

<b>User Story Name:</b> Clear graph	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	clear the graph
<b>So that:</b>	I can start a new analysis
<b>Description</b>	Start work over

<b>User Story Name:</b> Move nodes	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	move nodes on the graph
<b>So that:</b>	I can make the graph more understandable
<b>Description</b>	Requires an ability to manipulate the graph

<b>User Story Name:</b> Name graph	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	name a graph
<b>So that:</b>	I can provide context to the graph that was generated
<b>Description</b>	Add a name of the artist or group of artists to the interface



<b>User Story Name:</b> Set defaults	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	configure features
<b>So that:</b>	I can determine default features in a graph
<b>Description</b>	color, size, shape of nodes

<b>User Story Name:</b> Artist info in list	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	see artist info in the selection list
<b>So that:</b>	I can differentiate between artists in the main list
<b>Description</b>	Add extra info into the selection list?

<b>User Story Name:</b> Hide nodes	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	hide nodes in the graph
<b>So that:</b>	I can clear some clutter
<b>Description</b>	Collapse leaf nodes in the graph

<b>User Story Name:</b> Generate graph in tab	
<b>Category:</b> Interaction	
<b>As a:</b>	User
<b>I want:</b>	generate graphs in new tabs
<b>So that:</b>	I can generate multiple graphs at once
<b>Description</b>	Allows a user to create multiple views at once

<b>User Story Name:</b> List of saved graphs	
<b>Category:</b> Data	
<b>As a:</b>	User
<b>I want:</b>	see a list of graphs generated thus far
<b>So that:</b>	I can go back to previous work
<b>Description</b>	Would require saving graphs in a database

## Concept Initiate 2

### Class Diagrams

In the last iteration, we worked with a number of use-cases for the Bristol Project. As implied by the project description, the primary goal of the application is to manage the collaborations between different artists (regardless of the genre to which they are associated). A *possible* use case diagram solution from the core set of user stories identified for last iteration is shown in Figure 2.

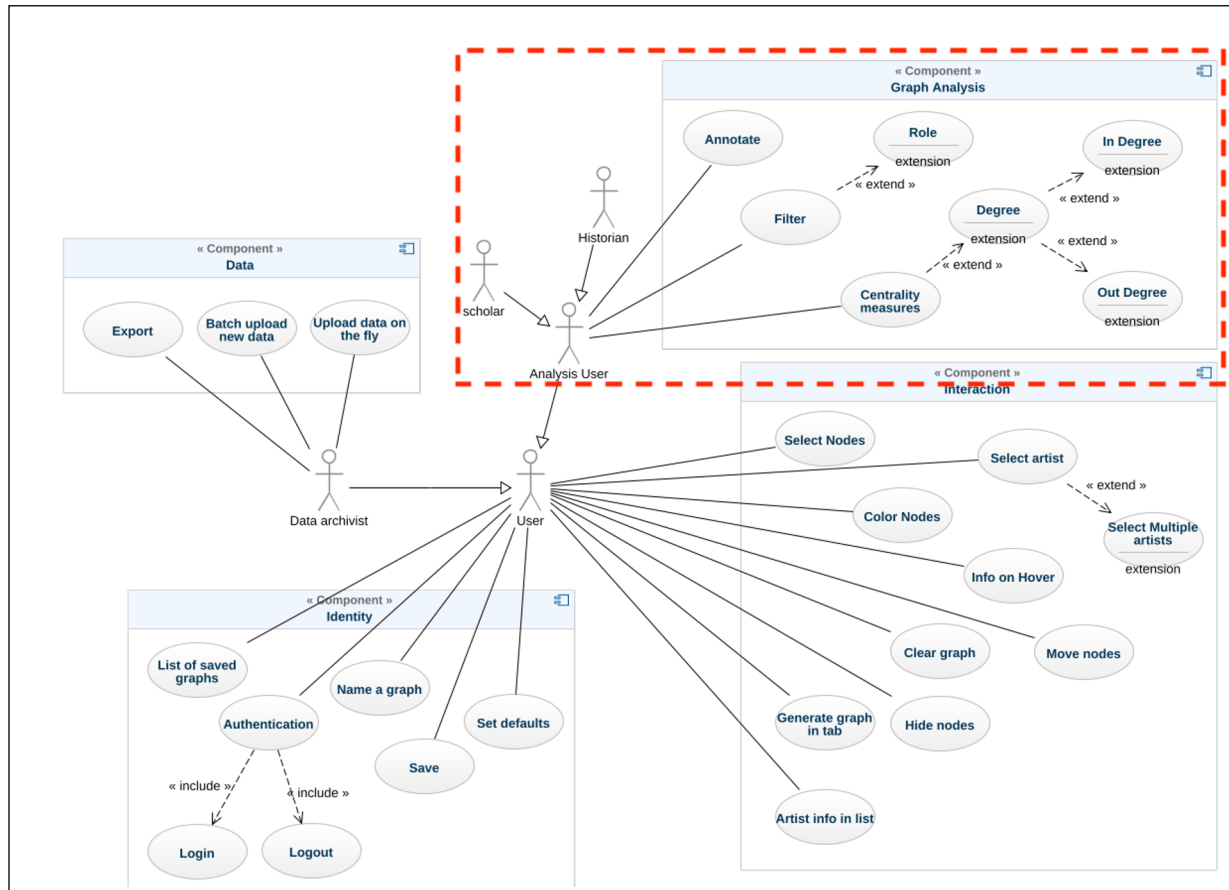
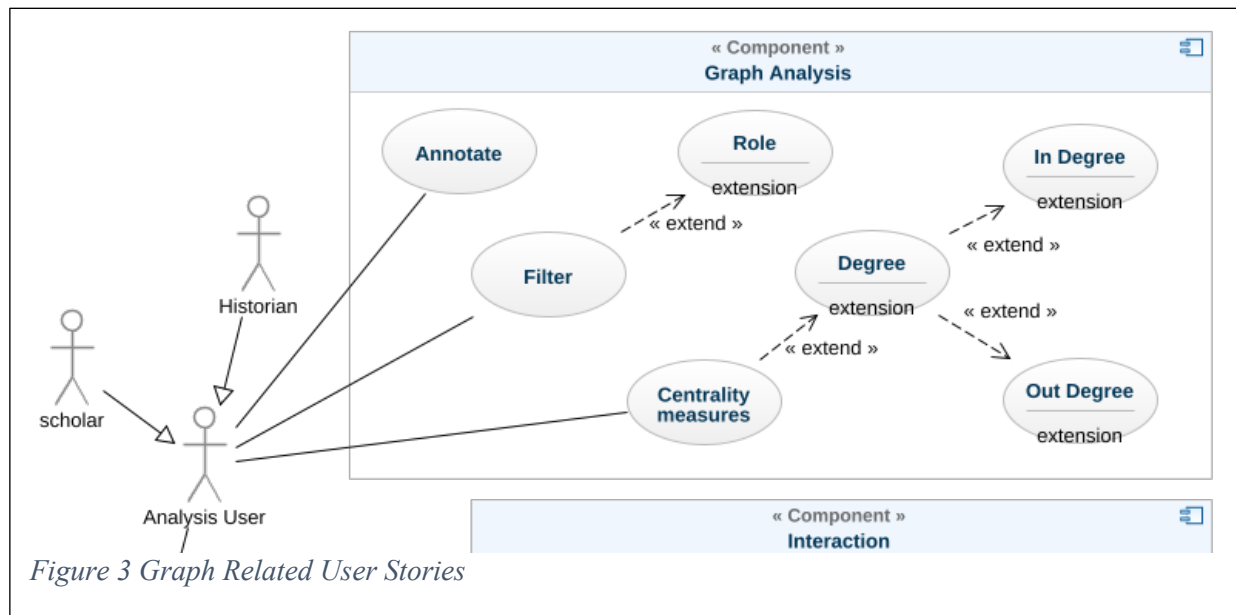


Figure 2 Use Case Diagram for Bristol Project

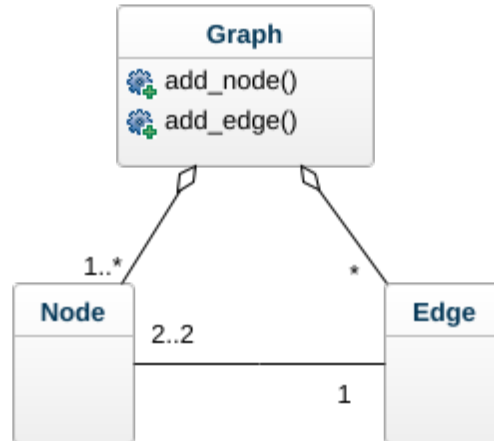
One of the primary aspects of this system is the analysis of graphs in order to better understand the collaborations between different recording artists. The user stories in the upper right-hand corner of the diagram (in the inset in Figure 2 and shown in Figure 3) provide details on these particular features. In particular, these features point to the need for an *ArtistGraph* class, which can be used to represent *Artists* as *nodes*, and the relationships (i.e., *collaborations*) between them as *edges* in a *Graph*. That is, an *ArtistGraph* is a *Graph*.

According to the diagram, other desired features include the ability to indicate what *role* an artist had on a recording, the number of collaborations an artist had with others (either as a primary artist via an “out-degree”, or as a supporting artist via an “in-degree”), and different metrics for a graph such as degree centrality, betweenness centrality, and closeness centrality.



### Data Analysis

The following class diagram shows a *Graph* class as a collection of *Node* and *Edge* objects. The diagram depicts an edge as a relationship between two nodes. Through the use of inheritance, aggregation, composition, and association, you will create a model depicting the major concepts needed to analyze artists and their collaborations.



In this iteration you will be creating a class diagram to model the major classes for the Bristol Project. The core classes that you should include are:

- Graph
- Node
- Edge
- ArtistGraph

Other classes, however, also exist that you will have to identify by analyzing some of the data available to populate the graphs. In particular, you should analyze the following sample data related to Jimmie Rodgers available from the Discogs API:

- Artist Data: <https://api.discogs.com/artists/269365>
  - The data for an artist provides a listing of associated attributes including the name, artist id, profile, and other information
  - The artist data also includes a link to a list of releases related to the artist: for instance, for Jimmie Rodgers, the list can be found here:  
<https://api.discogs.com/artists/269365/releases>;
- Release Data: <https://api.discogs.com/releases/11530368>
  - The data for an individual release provides a release id, the year of the release, and information about *Tracks* (i.e., individual songs) including the artists that performed on the track.

In this iteration, you are to create a class diagram that identifies all of the classes, their relationships (e.g., **inheritance**, **composition**, **aggregation**, **association**), the **multiplicities** of the relationships (e.g., 1-1, 1..\*, etc.) and **potential attributes** and (non-constructor, non-getter, and non-setter) **methods**.

To help simplify the modeling, it is suggested that you create modeling diagrams as follows:

- Relationships: Create a model that shows the classes and their relationships. Omit the use of attributes and methods on this diagram and focus only on the relationships
- Class information: Create one or more models that omit the relationships but show the attributes and methods of each class. This should include any parameters used for methods.

### *Submission*

Use the <https://app.genmymodel.com> system to create your models. Export your model to a PNG and submit as a PDF to iLearn. Videos are available online on the different modeling notations for UML.

### *Rubric*

*Completeness:* You will be graded on the completeness of your submission, most notably the inclusion of classes described above as well as on the identification of implicit and derived classes that arise from a thorough analysis of the problem.

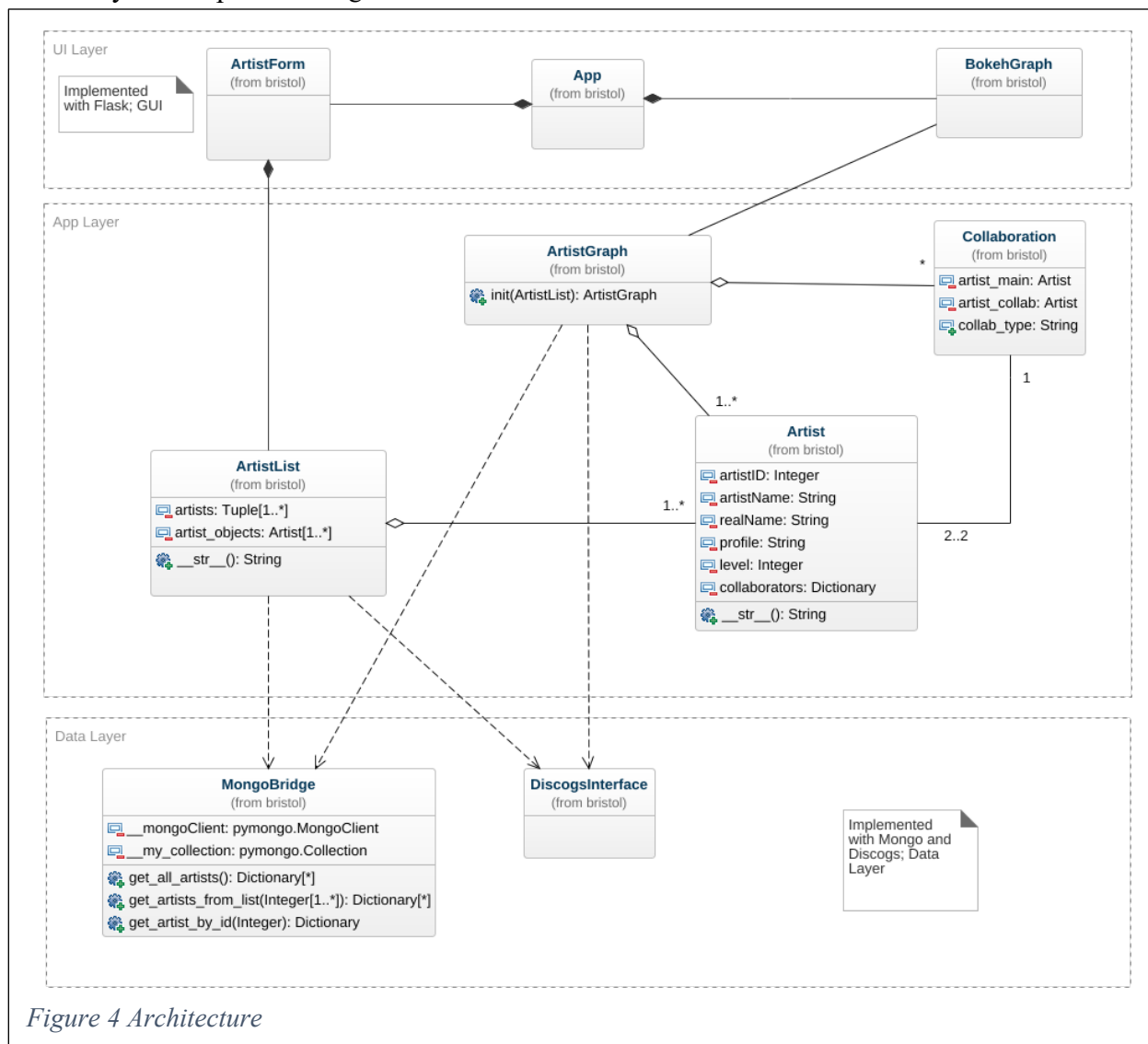
*Correctness:* You will be graded on the correctness of the class specifications you have created including correct capture of the relationships between the classes and the specification of attributes and methods that arise from a thorough analysis of the problem.

*Points:* The assignment is worth 30 project points.

## Iteration 0

### Architecture

In the last iteration, you analyzed the data related to artists and their collaborators. The model you created is an important part of understanding the relationship between artists and the collaborations they had with other artists. For the initial iteration of the implementation you will be creating a number of classes as part of a standard 3-tier architecture involving a *UI* layer, an *App* layer, and a *Data* layer as depicted in Figure 4.



As shown in the architecture, each layer is responsible for different aspects of the implementation, including interactions with the user in the UI layer, the modeling of the data as a graph in the App layer, and access to the data in the Data layer. The architecture depicts a handful of classes that you will either implement or use as part of creating the solution, or are planned as part of later iterations. In this first iteration the focus is on getting the basic application up and running, implementing some support classes, and creating a bridge between the application and its data source (i.e., a mongo database).

The application itself runs using the Flask web development framework for python. The application works as follows:

1. The App object instantiates an ArtistForm object, which populates the UI with a list of the names of Bristol artists using an ArtistList object.
2. The ArtistList object receives a list of integers corresponding to Artist IDs, which it uses to retrieve a dictionary of data for an artist using the MongoBridge.
3. The MongoBridge object is an interface to a MongoDB instance; it makes the call to MongoDB using the pymongo library.
4. When a user selects one or more artists using the ArtistForm object, the App object will display:
  - A list of the selected artists as “artist name (artist id)” (as shown in Figure 5)
  - Eventually, the app will display a graph showing artists and their related collaborators

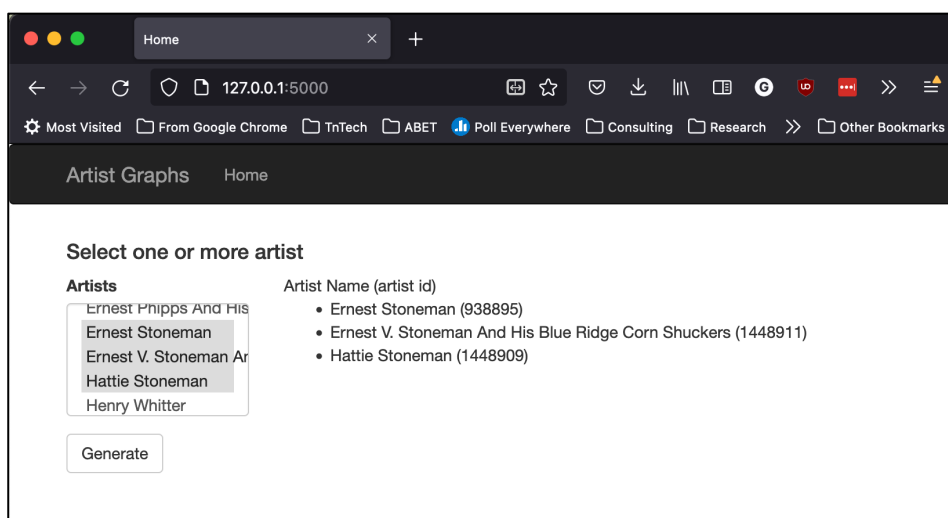


Figure 5 Interface

## Implementation

This section describes the setup and coding for the iteration.

### Setup

Begin by cloning the project repository:

[https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph\\_0.git](https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph_0.git)

replacing *yourid* with your userid. **It is recommended that you create a new directory, cd into that directory, and then clone the repo at that location.** The repository has the structure shown in Figure 6 with the applayer, datalayer, and guilayer corresponding to the architecture layers. **When doing your work, edit the files in these directories directly.** Specifically, do not copy these files into other directories or to other locations other than where they were downloaded using the clone command.

Your ability to clone, modify, stage, commit, and push your solution back to gitlab will be assessed in this iteration of the project.

Once you have cloned the project, you must setup the venv environment by issuing the following command:

```
pip3 install -r requirements.txt
```

This will ensure that all of the libraries needed for the project are available in your environment.

### Coding

In this iteration you will be implementing three classes shown in the architecture diagram found in Figure 4. All of the source files that you will need have been pre-populated in the project repository and contain stubs for all of the methods that you must implement. You may add other methods but you must not change the signature of any of the methods already stubbed in the provided source files. Each of the methods has

been documented. *It is highly recommended that you implement the classes in the order listed below. In particular, ArtistList depends on a functioning MongoBridge class.*

Name	Last commit	Last update
applayer	Initial commit	1 day ago
datalayer	Initial commit	1 day ago
docs	Initial commit	1 day ago
guilayer	Initial commit	1 day ago
templates	Initial commit	1 day ago
test	Initial commit	1 day ago
.gitignore	Initial commit	1 day ago
README.md	Initial commit	1 day ago
app.py	Initial commit	1 day ago
requirements.txt	Initial commit	1 day ago

Figure 6 Repo Structure

### Artist

The Artist class contains basic information regarding an artist as well as a dictionary of their related collaborators. The class implements a getter for all of the attributes except level, for which it also implements a setter. The class also implements a `__str()` operation, which is used to print the object in the form “artistName (artistID)”.

### MongoBridge

The MongoBridge class is an interface to the MongoDB instance containing the data for the application. The class reads data from the database and returns a list of dictionaries containing the raw data found in the database collections. To begin, you should import the data found in the datalayer/Artists.json file into a Mongo database called “**BristolData**” into a collection called “**Artists**” using the same approach used in Lab 01:

```
mongoimport --db BristolData --collection Artists --type json --jsonArray --uri mongodb://localhost:27017 --file /path/to/Artists.json
```

Note that you must startup docker as you did in Lab 01. To access the data from a mongo server, you must first instantiate a mongo client and establish a connection to a collection in a database, as follows:

```
mongoClient = pymongo.MongoClient("mongodb://localhost:27017/")
myCollection = mongoClient["BristolData"]["Artists"]
```

To retrieve an individual record from the database you must specify a filter and issue a find command, as follows, which will search the database for the record with artistID = 938895:

```
afilter = {"artistID": 938895}
a = myCollection.find_one(afilter)
```

Variable a will either contain a single record or None.

A variant of the find method will return all records found in a collection but must be explicitly iterated through in order to “download” the records:

```
result: List[dict] = []
artists = myCollection.find()
for a in artists:
    result.append(a)
```

As such, the result variable will contain all records.

### *ArtistList*

The ArtistList class is used to support displaying a list of artists. The artists attribute is a *sorted* list of tuples in the form:

(artistID, artistName)

For example:

(269365, "Jimmie Rodgers")

The artist\_objects attribute is simply a list of Artist objects. The class implements getters for the two attributes as well as a \_\_str\_\_() operation, which is used to print the individual Artist objects in a comma delimited list.

This class is responsible for connecting to the mongo server using the MongoBridge class. It will also instantiate individual Artist objects, saving them in the artist\_objects attribute list.

### *Test Files*

The test directory contains three test files:

- test\_artist.py
- test\_mongobridge.py
- test\_artistlist.py



These files contain the core tests using unittest for the Artist, ArtistList, and MongoBridge classes, respectively. It is suggested that you fully employ the TDD approach by using these test files as your guide for implementing the project. Also, make note that you should implement the classes in the order of Artist, MongoBridge, and then ArtistList.

You should not make modifications to the test files except to add your own tests. Changing the nature of how tests are executed for each of the individual methods may affect the fidelity of your implementation.

***You are required to create your own tests.***

### *Execution*

The application has been configured to allow for unittest, debugging, and system tests. While the system can be executed by PyCharm, your system should also be executable via the terminal using the following command: **python3 -m flask run**. The application will not run correctly until all of the code has been written.

### *Submission*

The schedule for the project has shifted slightly as shown below:

Date (Midnight)	Phase	Description
September 9	Concept Initiate 0	User stories
September 23	Concept Initiate 1	Use case diagrams
October 7	Concept Initiate 2	Initial Design
<del>October 21</del> Mon Oct 31	Iteration 0	Project environment setup and initial execution demonstration
<del>November 4</del> November 11	Iteration 1	Graph Visualization
<del>November 18</del> December 2	Iteration 2	Graph Algorithms and Delivery

Upon completion of your code, you should commit and push to gitlab using the following commands:

```
git commit -m "Completed Solution"
git push -u origin master
```

### *Rubric*

The primary concerns for evaluation are the following:

- Implementation for the Artist, MongoBridge, and ArtistList classes must address the features described in the comments of the provided source files and associated tests. You will be evaluated on the quality of the code and the execution.
- You are required to create your own tests in addition to the provided set. The tests you create for the application must pass as you have specified.

- Supplied tests for Artist, MongoBridge, and ArtistList must pass as specified in the associated test files.
- Proper use of git: You will be assessed on your ability to clone, modify, stage, commit, and push your solution back to gitlab. DO NOT submit your files to iLearn. No files submitted to iLearn will be accepted.
- The application must run when issuing the `python3 -m flask run` command in the terminal.

*Points:* This assignment is worth 40 points

### *Tips*

Historically, this iteration has tended to be overwhelming. Here are some tips:

- Start early; there is a lot of information to consume. Starting early will help you with comprehension
- Focus on one task at a time: implement the classes in the order suggested and focus on the individual methods as specified in the source file comments
- Use the unittest tests as your guide for implementing the methods; create your own tests as appropriate in order to break down the implementation task into smaller chunks

## Class Diagrams

```

classDiagram
    class Node {
        +id: Integer
        +name: String
        +realname: String
        +profile: String
        +releases: Release[1..*]
        +collaborators: ArtistDict[1..*]
        +filter_role(): Artist
    }
    class GraphBase {
        +add_node(Artist): None
        +add_edge(Artist, Artist): None
        +has_node(Artist): Boolean
        +has_edge(Artist, Artist): Boolean
        +incr_edge(Artist, Artist): None
    }
    class ArtistGraph {
        +artists: Artist[1..*]
        +collaborations: Collaboration[1..*]
        +__init__(Artist[1..*])
        +add_artist(Artist): None
        +add_collaboration(Collaboration): None
    }
    class Edge {
        +artist0: Artist
        +artist1: Artist
        +roles: String[1..*]
        +__init__(Artist, Artist, String[1..*])
    }
    class Collaboration {
        +artist0: Artist
        +artist1: Artist
        +roles: String[1..*]
        +__init__(Artist, Artist, String[1..*])
    }
    class Artist {
        +artistID: Integer
        +name: String
        +realname: String
        +profile: String
        +releases: Release[1..*]
        +collaborators: ArtistDict[1..*]
        +filter_role(): Artist
    }
    class Release {
        +id: Integer
        +title: String
        +year: Integer
        +tracks: Track[1..*]
    }
    class Track {
        +title: String
        +artists: Artist[1..*]
    }
    class Annotation {
        +text: String
    }
    class ReleaseList {
        +releases: Release[1..*]
    }

    Node <|-- Artist
    Node <|-- Release
    GraphBase <|-- ArtistGraph
    GraphBase <|-- Collaboration
    Edge <|-- Collaboration
    ArtistGraph "1" o-- "*" Collaboration
    Artist "1" o-- "*" Release
    Artist "1" o-- "*" ReleaseList
    Artist "1" o-- "*" Track
    Release "1" o-- "*" Track
    Collaboration "1" o-- "*" Artist
    Collaboration "1" o-- "*" Release
    Annotation "*" o-- "1" Artist
    ReleaseList "1" o-- "*" Release
  
```

Figure 7 shows an overview of the Graph classes used to represent Artists and Collaborations between Artists in an ArtistGraph. A detailed view of the diagram is shown in Figure 8. The model indicates that ArtistGraph is a subclass of GraphBase, and that Artist and Collaboration are each subclasses of Node and Edge. Figure 9 shows a sample graph, including Artists and nodes, and their Collaborations as edges.

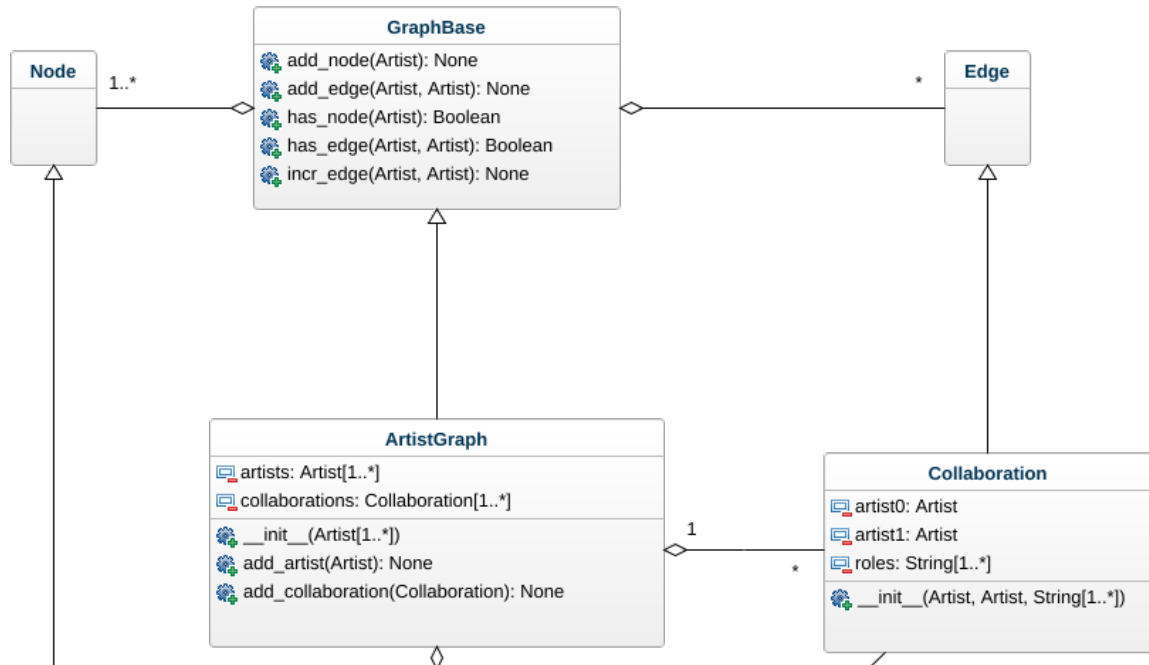


Figure 8 Detailed View of ArtistGraph and Collaboration

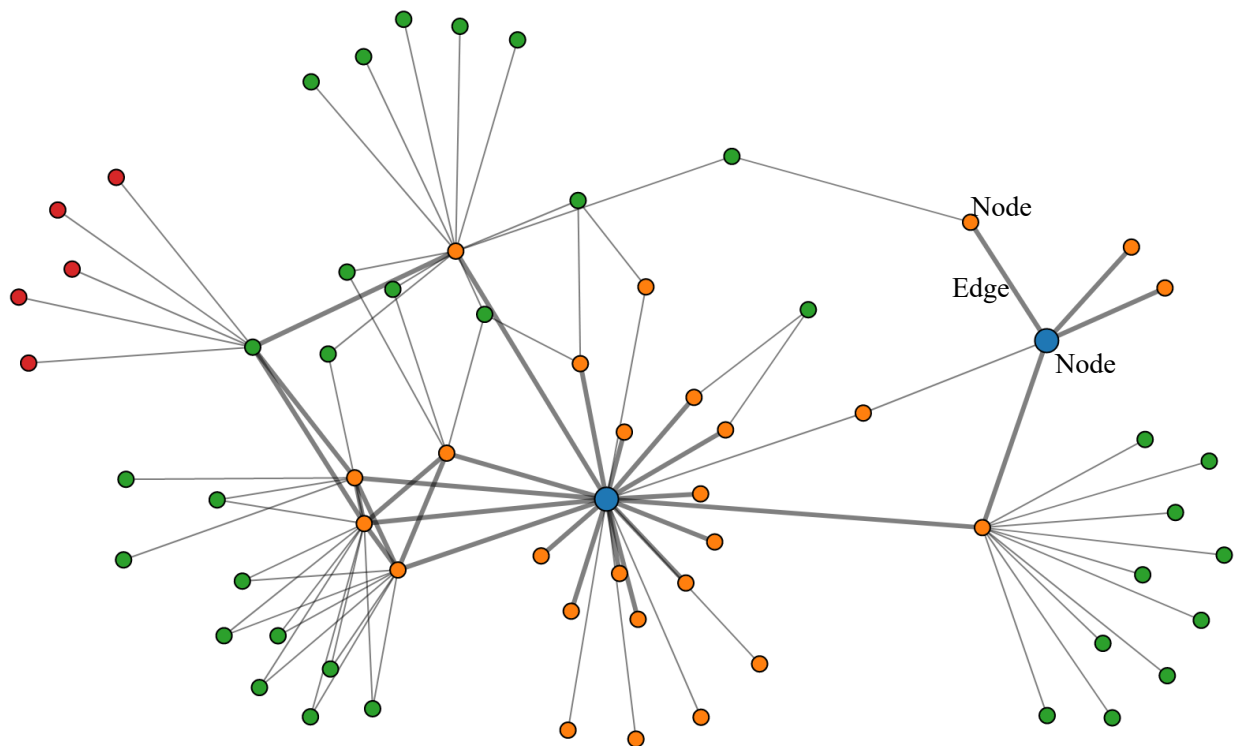


Figure 9 Sample Graph

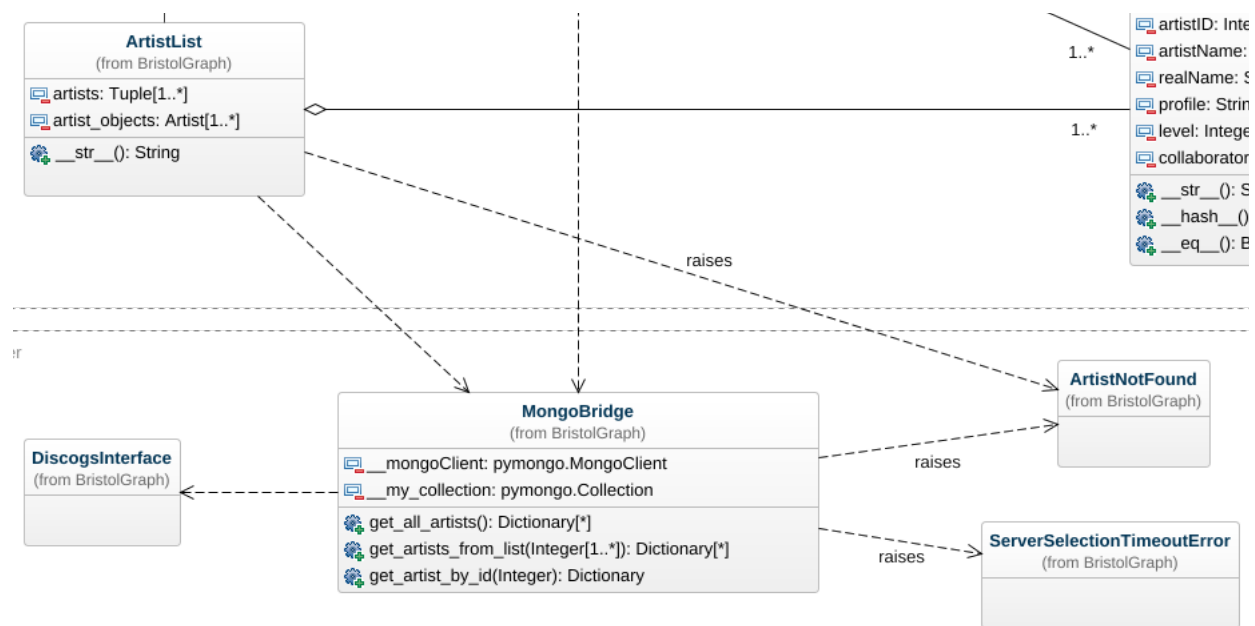


Figure 10 MongoBridge and Exceptions

Figure 10 depicts the updated architectural model with MongoBridge and its dependency on the ArtistNotFound exception.

### Implementation

You are responsible for implementing or modifying the following classes. The descriptions below provide details of what is to be developed. The models above provide some syntactic information.

### Setup

Begin by cloning the project repository:

[https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph\\_1.git](https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph_1.git)

replacing *yourid* with your userid. **It is recommended that you create a new directory, cd into that directory, and then clone the repo at that location.** The repository has the structure shown in Figure 6 with the *applayer*, *datalayer*, and *guilayer* corresponding to the architecture layers. **When doing your work, edit the files in these directories directly.** Specifically, do not copy these files into other directories or to other locations other than where they were downloaded using the clone command.

Your ability to clone, modify, stage, commit, and push your solution back to gitlab will be assessed in this iteration of the project.

Once you have cloned the project, you must setup the *venv* environment by issuing the following command:

```
pip3 install -r requirements.txt
```

This will ensure that all of the libraries needed for the project are available in your environment.

## Coding

It is suggested that you implement the classes in the order listed below. You are required to write tests for your solution prior to developing the classes.

**You must commit and push your tests to gitlab before you write your code as follows:**

`git commit -m "Tests written"`

When writing your tests, it is suggested you do the following:

- Consider all of the conditions that must be true when using the classes
- Consider what happens when “bad” data is passed to the classes

## Collaboration

The Collaboration class represents an edge drawn in a graph. In this domain, it represents a collaboration between two artists. The following details provide some details to assist in the implementation.

- Implement class for Collaboration as a subclass of object
- Attributes:
  - `self.__artist0: Artist`
    - First node of an edge
  - `self.__artist1: Artist`
    - Second node of an edge
  - `self.__roles: List[str]`
    - A list of strings of roles taken by an artist in a collaboration
- Methods:
  - `def __init__(self, art0: Artist, art1: Artist, role: List[str] = None):`
  - Getters for `artist0`, `artist1`, and `roles`

## ArtistGraph

The ArtistGraph class is the primary component of the system needed to display graphs visually. The class relies on the GraphBase attributes and methods. The class has been provided and includes operations for adding nodes and edges, and for checking whether nodes and edges have already been added.

- Implement this class as a subclass of GraphBase; GraphBase is included. All interactions with GraphBase should occur as calls to the super-class methods.
- Attributes
  - `self.__artists: List[Artist]`
    - Used to maintain a list of all Artist objects added to the graph
  - `self.__collaborations: List[Collaboration]`
    - Used to maintain a list of all of the Collaboration objects added to the graph
- Methods:
  - `def __init__(self, artist_list: ArtistList, depth: int) -> None:`
    - Iterate through the list of Artist objects, including the collaborators found in the Artist objects
    - Build the graph by adding artists and collaborations to the graph using appropriate ArtistGraph methods (see below).

- Use MongoBridge to retrieve individual artists from the database as needed when iterating through the collaborators
- Handle exceptions when ArtistNotFound encountered
- def add\_collaboration(self, collab: Collaboration) -> None:
  - Add a collaboration to the graph using the super().add\_edge method if the edge does not already exist
  - If an edge exists, use super().incr\_edge() instead
- def add\_artist(self, artist: Artist) -> None:
  - If an artist node doesn't already exist in the graph, add it
- Getters for artists, collaborations

### *MongoBridge*

- Modify get\_all\_artists(), get\_artists\_from\_list(), and get\_artist\_by\_id() to handle cases of when artists are not found
  - get\_artists\_from\_list should generate ArtistNotFound exception only if none of the artists in the list are found
  - get\_all\_artists should generate ArtistNotFound exception only if no artists are found
  - get\_artist\_by\_id should generate ArtistNotFound exception if the artist is not found

### *ArtistList*

- Modify to handle exceptions generated by MongoBridge.

### *Test Files*

The same test files from Iteration 0 have been included in the base distribution. In addition, a new class has been added: test\_artistgraph.py. Other tests will be released mid-iteration. You are required to submit your initial set of tests prior to beginning your solutions.

### *Execution*

When you are ready to test the program, find the following line of code in app.py and uncomment the lines after it:

# Uncomment the next two lines to do system tests

The program is executable using the same command used in Iteration 0.

### *Submission*

The schedule for the project has shifted slightly as shown below:

Date (Midnight)	Phase	Description
<b>September 9</b>	Concept Initiate 0	User stories
<b>September 23</b>	Concept Initiate 1	Use case diagrams
<b>October 7</b>	Concept Initiate 2	Initial Design
<del>October 21</del> <b>Mon Oct 31</b>	Iteration 0	Project environment setup and initial execution demonstration
<del>November 4</del> <b>November 15</b>	Iteration 1	Graph Visualization

Upon completion of your code, you should commit and push to gitlab using the following commands:

```
git commit -m "Completed Solution"
git push -u origin master
```

### *Rubric*

The primary concerns for evaluation are the following:

- Implementation for ArtistGraph is the primary focus of this iteration. Spend time working out how the graphs are supposed to be built by examining the data available in the mongo database. Consider what should happen when iterating through a list of collaborators from the mongo dictionary, and what should happen when an artist ID from a collaborator list does not exist.
- You are required to create your own tests in addition to the provided set. The tests you create for the application must pass as you have specified.
- Supplied tests for Artist, MongoBridge, ArtistList, and ArtistGraph must pass as specified in the associated test files.
- Proper use of git: You will be assessed on your ability to clone, modify, stage, commit, and push your solution back to gitlab. DO NOT submit your files to iLearn. No files submitted to iLearn will be accepted.
- The application must run when issuing the `python3 -m flask run` command in the terminal.

*Points:* This assignment is worth 50 points

### *Tips*

This iteration is challenging and the level of complexity has increased. Here are some tips:

- Start early; there is a lot of information to consume. Starting early will help you with comprehension.
- It is suggested you analyze the GraphBase class and how to use inheritance. Your implementation of the ArtistGraph depends on your understanding of how to use it.
- Focus on one task at a time: implement the classes in the order suggested and focus on the individual methods as specified in the source file comments
- Use the unittest tests as your guide for implementing the methods; create your own tests as appropriate in order to break down the implementation task into smaller chunks



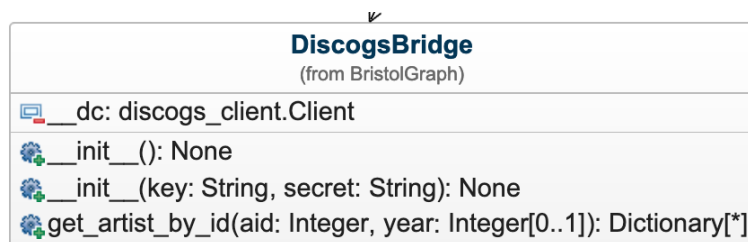
## Iteration 2

### Class Diagram

In the final iteration you will be implementing a class that will be used to extract data using the Discogs API. The class you will implement is called DiscogsBridge. This class will deliver data in the exact same format used to implement the MongoBridge class from Iteration 0. The assignment is split into two parts:

- A minimal, mandatory component worth 15 points
- An additional extra credit component worth an additional 16 points

The diagram snippet below shows the attributes and methods that you should implement. Refer to Laboratory 9 for details.



### Implementation

Laboratory 9 provides details on creating an account in Discogs in order to access the API. In addition, the lab describes how retrieve the python3-discogs-client library.

### Setup

Begin by cloning the project repository:

[https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph\\_2.git](https://gitlab.csc.tntech.edu/csc2310-fa22-students/yourid/yourid-artistgraph_2.git)

replacing *yourid* with your userid. **It is recommended that you create a new directory, cd into that directory, and then clone the repo at that location.** The repository has the structure shown in Figure 6 with the *applayer*, *datalayer*, and *guilayer* corresponding to the architecture layers. **When doing your work, edit the files in these directories directly.** Specifically, do not copy these files into other directories or to other locations other than where they were downloaded using the clone command.

Your ability to clone, modify, stage, commit, and push your solution back to gitlab will be assessed in this iteration of the project.

Once you have cloned the project, you must setup the venv environment by issuing the following command:

```
pip3 install -r requirements.txt
```

This will ensure that all of the libraries needed for the project are available in your environment.

## *Coding*

You are only expected to implement a single method (in addition to the constructors):

- `def get_artist_by_id(self, aid: int, year: int = 1935) -> dict:`

The `get_artist_by_id` method accesses the Discogs API in order to create a dictionary with the following keys:

- `artistID`
- `artistName`
- `realname`
- `profile`
- `collaborators`
- `level`

The data retrieved should be for the artist with the given artist id and limited to dates that precede the input `year`. Note that `year` is optional and has a default of 1935. As shown in the `discogsdemo.py` file from Lab 09, artist data is accessed by initializing a discogs client, and using the `artist` method:

```
dc = discogs_client.Client(
    'CSC2310_Lecture/1.0',
    consumer_key=key,
    consumer_secret=secret,
)

artist_dictionary = dc.artist(aid)
```

In the example, `artist_dictionary` will contain a number of attributes, all of which similarly correspond to the elements of the JSON data found here:

<https://api.discogs.com/artists/1141491>

## *Minimal Implementation*

At a minimum, you must implement the basic retrieval of the `artistID`, `artistName`, `realname`, `profile`, and `level` attributes. These are depicted below as `id`, `name`, `realname`, and `profile`. As before, `level` is “0” by default.

```

name: "Alfred G. Karnes"
id: 1141491
resource_url: "https://api.discogs.com/artists/1141491"
▶ uri: "https://www.discogs.com/.../1141491-Alfred-G-Karnes"
releases_url: "https://api.discogs.com/artists/1141491/releases"
▶ images: [...]
realname: "Alfred G. Karnes"
▶ profile: "Alfred G. Karnes (1891-1...orbin, Kentucky, area. "
▶ urls: [...]
▶ namevariations: [...]
data_quality: "Correct"

```

### Collaborators

The collaborators component of the return value is a collection of collaboration dictionaries formatted as shown in the following example:

```

{
  "collaboratorID": 1448909,
  "collaboratorName": "Hattie Stoneman",
  "releaseID": 10594844,
  "roles": ["Fiddle [Uncredited]"]
}

```

The artist dictionary includes an attribute for all of the releases associated with the artist. This attribute is a collection of *release* information. For the above example, release information is first collected in the `releases_url`, and in separate release objects as depicted in the following url:

<https://api.discogs.com/releases/8401049>

Each release object *may have* the following attributes:

- `id`: a release id
- `year`: a release year
- `artists`: a list of dictionary objects with artist information
- `extraartists`: a list of dictionary objects with artist information
- `tracklist`: a list of tracks, which in turn *may have* an `extraartists` attribute.

We emphasize that these objects and dictionaries *may have* attributes as the data may exclude them. As such, for a release object called “`r`”, for *year*, for instance, the data must be retrieved by first checking to see if the attribute exists:

```

artist = dc.artist(aid)
artistName = artist.name
releases = artist.releases
for r in releases:
    if hasattr(r, "year"):
        print(r.year)

```

For tracks pulled from the tracklist attribute of a release, the extraartists attribute must be “fetched”. For a track “t”, this is achieved with:

```
ea = t.fetch("extraartists")
```

### *Submission*

Upon completion of your code, you should commit and push to gitlab using the following commands:

```
git add .  
git commit -m "Completed Solution"  
git push -u origin master
```

### *Rubric*

This iteration is worth 15 points for the *Minimal Implementation*. The implementation of the *Collaborators* component worth 15 extra credit points. The rubric is based on the following components:

- Git submission: 5
- Minimal attributes implemented: 5
- Minimal attributes pass tests: 5
- Collaborators coded: 8
- Collaborators passes tests: 8