Sprint Two:

Requirements Engineering:

Part A) User and System Requirements:

User Requirements:

The system will recommend a bar that is cool (hosts a set of amenities, has high traffic during peak hours, is open often, and has good reviews) and safe (features of a set of safety regulations, does not host too many people at a given time).

System Requirements:

1. Given some location and search radius, the system will store a collection of bars in-line with the given parameters.
2. The system will gather and store data (amenities and safety regulations) with respect to each bar.
3. The stored list of bars will be sorted by an inputted metric (either by safety or coolness).
4. The system will output, to the user, at least one bar that matches the constraints.
5. If few constraints are provided, the system will provide the user with a list of bars following whatever inputs are provided.

Part B) Functional and Non-Functional Requirements:

Functional Requirements:

The bar will follow reasonable restrictions with respect to occupancy or density and implement certain safety measures such as social distancing, and these will apply to all customers and staff.

Non-Functional Requirements:

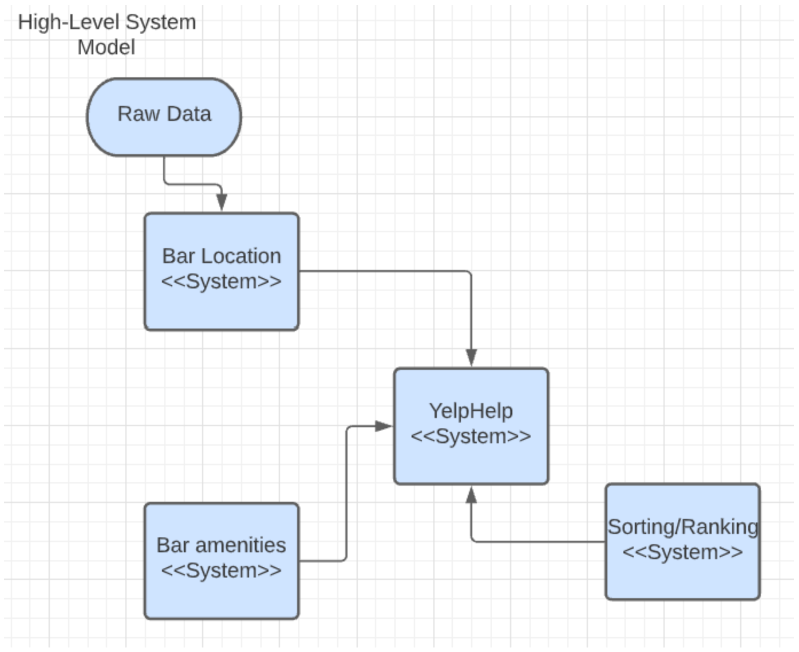
1. The system will provide a bar that is related to the user's desired coolness, amenities, safety regulations, location, and search radius.
2. Each set of inputs will correspond to a unique bar recommendation (i.e changing the inputs will change the output).
3. The safety of a bar will be proportional to the inverse of the coolness of that bar ($B \sim 1/C$).

Non-Functional Requirements:

1. Product Requirements:
    a. Performance: The system shall return a bar recommendation within some specified time constraint. The number of accesses to bars (calls to Yelp API and to web scraping system) or specific dates must be minimized so performance is maximized.
    b. Space: The system should only store bars that match the inputs. It should actively identify and store bars that match as many of the inputs as possible. In this manner, the system avoids storing bars that are not possible recommendations.
    c. Dependability/Reliability: The system must be designed to be dependable specifically in the event that relevant bar data is absent. If no bars match the location and radius or no data is provided by the web scraping sub-system then the system must be able to operate in conjunction with the defined requirements.
    d. Security: The system must be protected against security threats and bugs when accessing web pages and calling external APIs.
2. Organizational requirements:
    a. Environmental: The system shall be hosted in an environment that is able to fetch and interact with real-time data. The environment must be able to adapt and support changes to existing data (if a bar closes or is no longer as cool as it previously was).
3. External requirements:
    a. Ethical considerations: The system shall not access personal data from the user without permission.

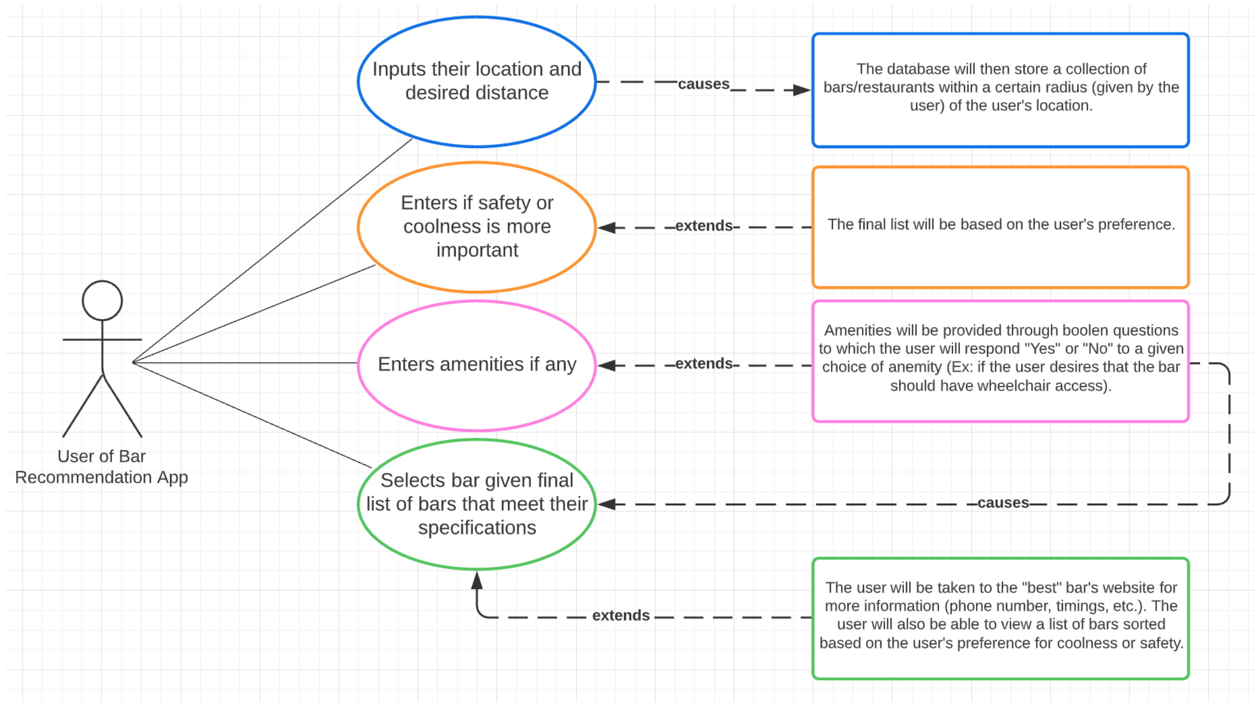
System Models:

High-level System Model:



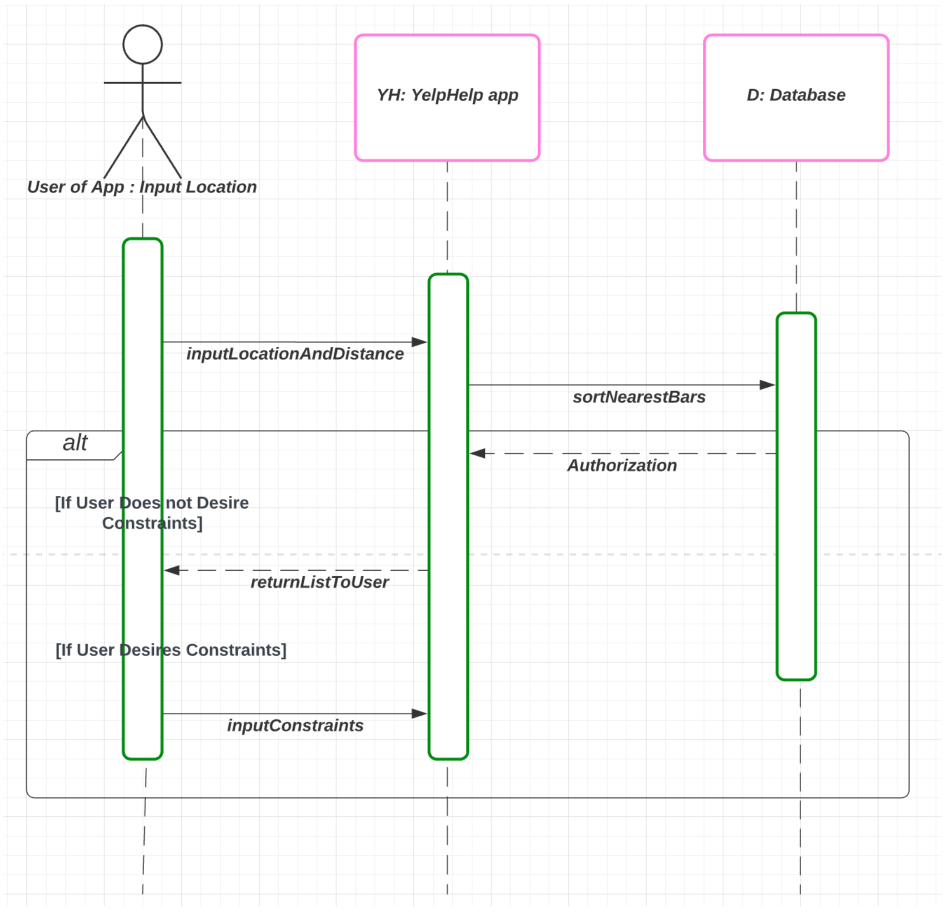
Bar Amenities are now taken into account for Sprint 2.

Use Case Diagram:

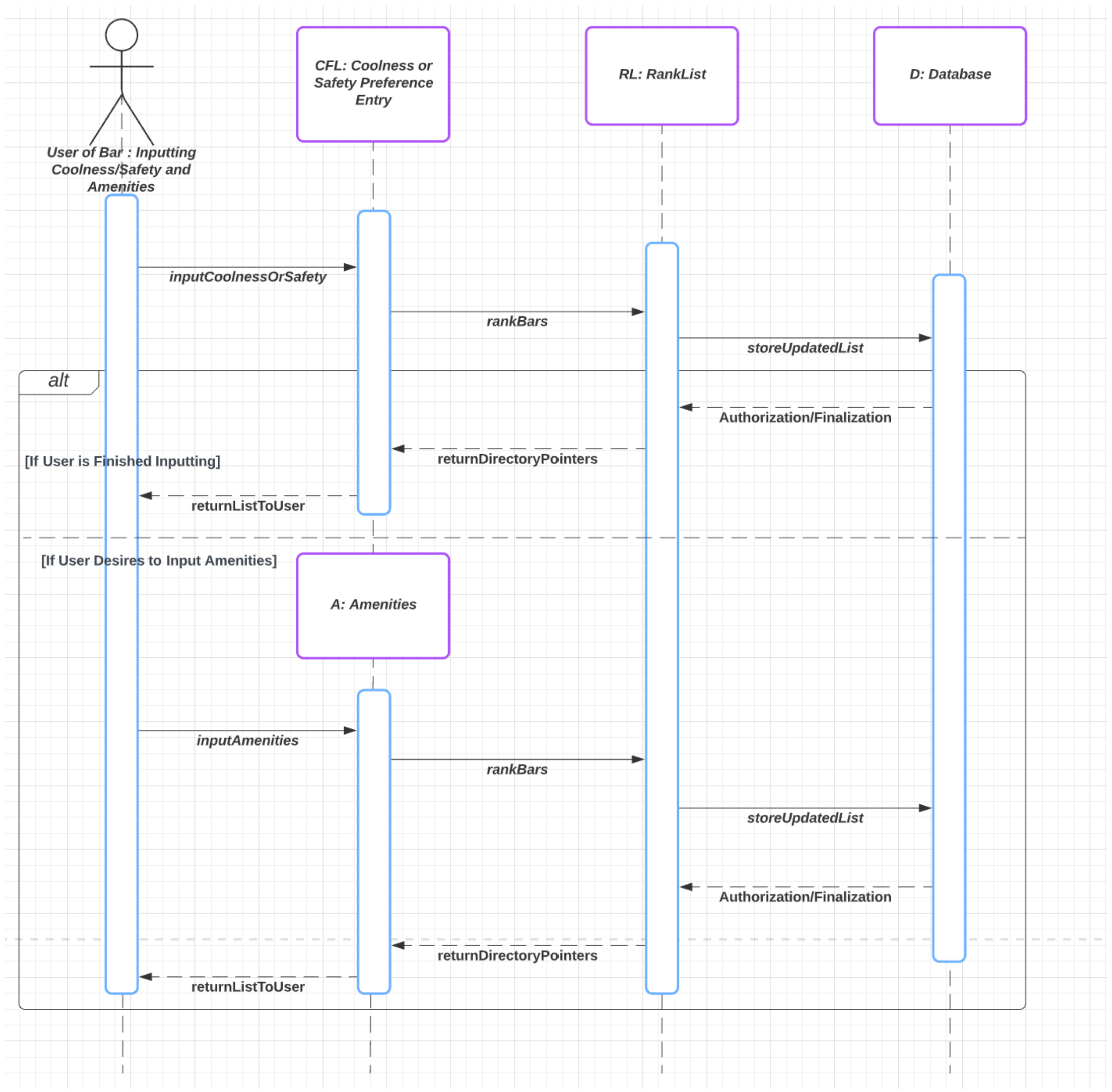

Inputs their location and desired distance
causes
The database will then store a collection of bars/restaurants within a certain radius (given by the user) of the user's location.
Enters if safety or coolness is more important
extends
The final list will be based on the user's preference.
Enters amenities if any
extends
Amenities will be provided through boolen questions to which the user will respond "Yes" or "No" to a given choice of anemity (Ex: if the user desires that the bar should have wheelchair access).
User of Bar Recommendation App
Selects bar given final list of bars that meet their specifications
causes
extends
The user will be taken to the "best" bar's website for more information (phone number, timings, etc.). The user will also be able to view a list of bars sorted based on the user's preference for coolness or safety.

The use case diagram is updated with a use case for inputting desired amenities. This way, the user will be able to be given a recommendation of a bar that is best suited to their specific preferences and has the amenities that the user wishes the bar to have.

Sequence Diagrams:

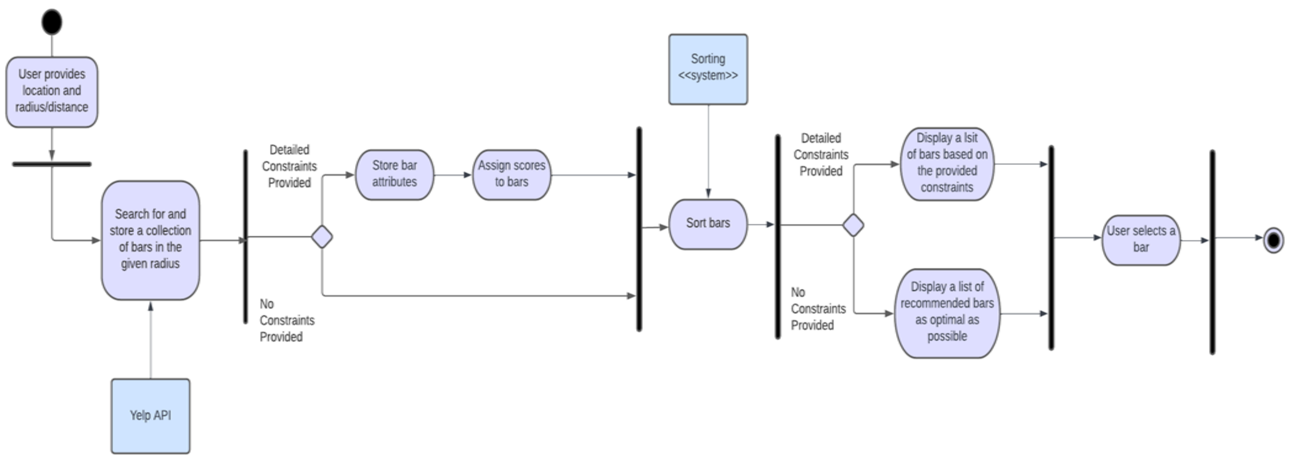


This sequence diagram gives an overview of the first use case: inputting location and radius. Once these two inputs are given, the bars that are located within the given radius of the user's current location are gathered and sorted. Once this is done, the user is asked if they wish to input any constraints related to safety or coolness (includes desired amenities). If so, the user then moves on to the next use case: inputting coolness or safety preference and amenities. If not, the sorted list is returned to the user.


CFL: Coolness or Safety Preference Entry
RL: RankList
D: Database
User of Bar : Inputting Coolness/Safety and Amenities
inputCoolnessOrSafety
rankBars
storeUpdatedList
alt
Authorization/Finalization
returnDirectoryPointers
[If User is Finished Inputting]
returnListToUser
[If User Desires to Input Amenities]
A: Amenities
inputAmenities
rankBars
storeUpdatedList
Authorization/Finalization
returnDirectoryPointers
returnListToUser

The sequence diagram (from Sprint 1) is updated with the user inputting specific coolness and safety factors that the YelpHelp app takes into account to rank the bars. The ranked list will now consist of bars that are better suited to the customer's tastes and interests. The alt box is included to check if the user wishes to input any safety considerations; otherwise, the ranked list (ranked based on coolness) is sent back to the user.

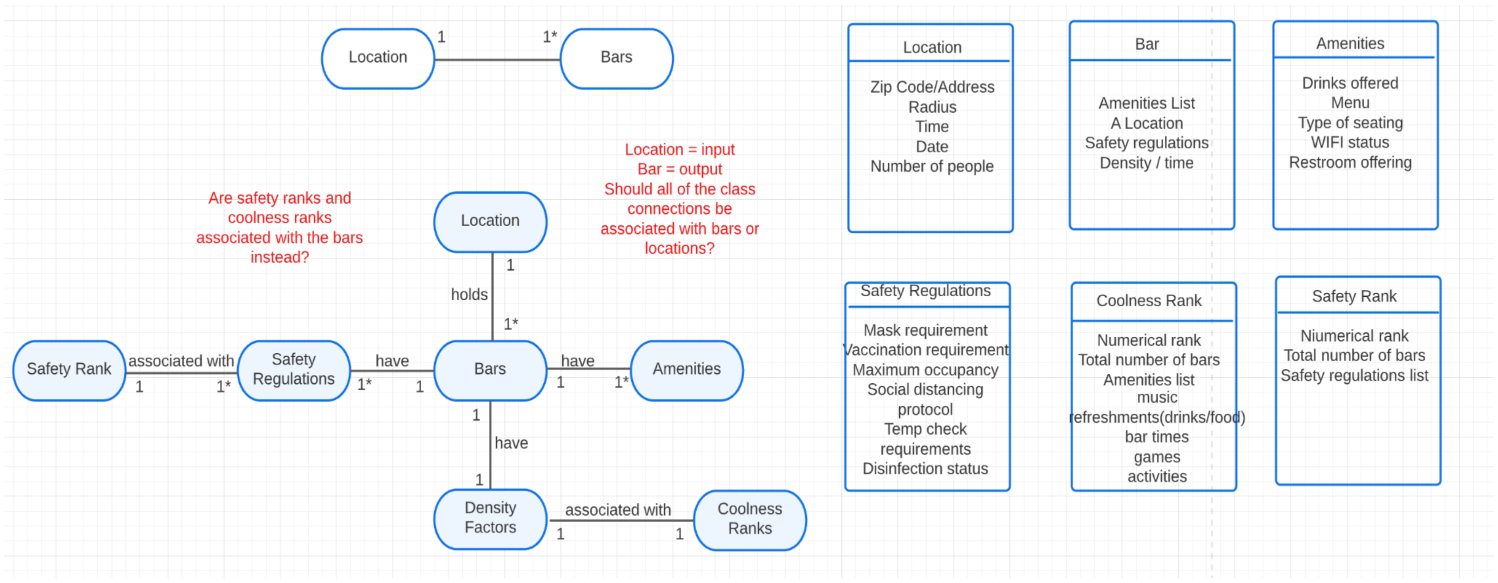
Activity Diagram/Process Model:


User provides location and radius/distance
Search for and store a collection of bars in the given radius
Yelp API
Detailed Constraints Provided
No Constraints Provided
Store bar attributes
Assign scores to bars
Sorting <<system>>
Sort bars
Detailed Constraints Provided
No Constraints Provided
Display a lsit of bars based on the provided constraints
Display a list of recommended bars as optimal as possible
User selects a bar

The bar customer provides location and radius. Then a collection of bars will be stored within the given radius of the user’s location. If detailed constraints are provided, then the system will store bar attributes and assign scores to bars. Then bars are sorted. If no constraints are provided, then bars will be sorted immediately. The system will output at least one bar that matches the constraints. If no constraints are provided, the system will output a list of recommended bars. In the end, the user will choose a bar from the list.

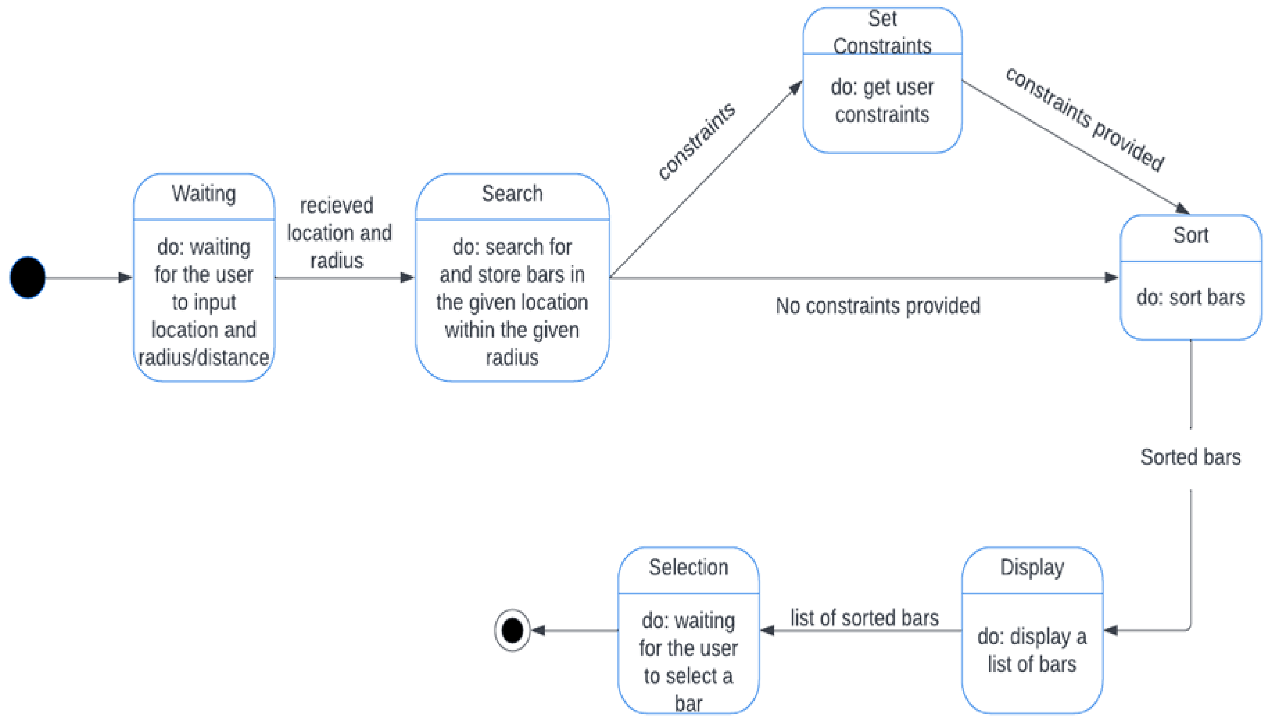
Class Diagrams:

Class Diagram:



The class diagram given above details the main interaction between the location class and bars class, where one location can have multiple bars (one input generates numerous bar recommendations). The specific relationships are given in the second diagram, where one location holds multiple bars, and each bar has multiple amenities, safety regulations, and one density factor (how populated each bar is). Safety regulations are also associated with safety ranks where a few regulations are stored in one bar which has one safety rank, and each density factor is associated with a coolness rank. The specific attributes and methods of each class are given in the diagrams on the right where many attributes are stored directly and some will be inferred (for example time, date, and zip code will be inferred from the user's machine and the supplied city/address). Not all classes will be implemented.

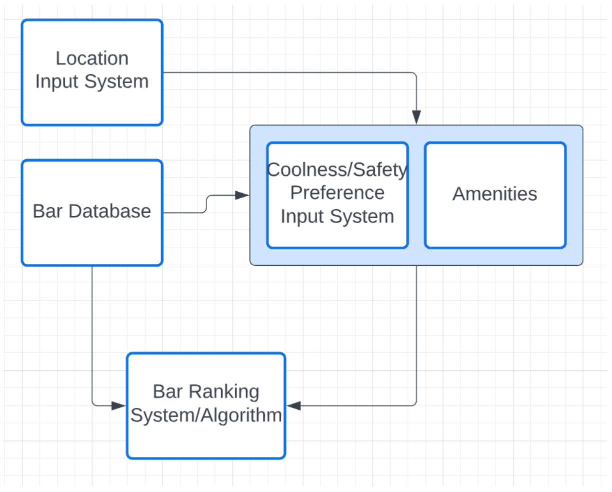
State Diagram:


Waiting
do: waiting for the user to input location and radius/distance
recieved location and radius
Search
do: search for and store bars in the given location within the given radius
constraints
Set Constraints
do: get user constraints
constraints provided
No constraints provided
Sort
do: sort bars
Sorted bars
Display
do: display a list of bars
list of sorted bars
Selection
do: waiting for the user to select a bar

The diagram shows the system states and the events that transition one state to another. The system starts with a waiting state. The system will wait for the user to input location and radius. Entering the location and radius will transition the system to a search state where the system will search for and store bars in the given location within the specified radius. Then if the user wants to provide constraints, the system will get the user's constraints and sort bars. If no constraints are provided, the system will be caused to transition to the sort state. After the bars have been sorted, the system will display a list of bars. Then the user will select a bar from the given list.

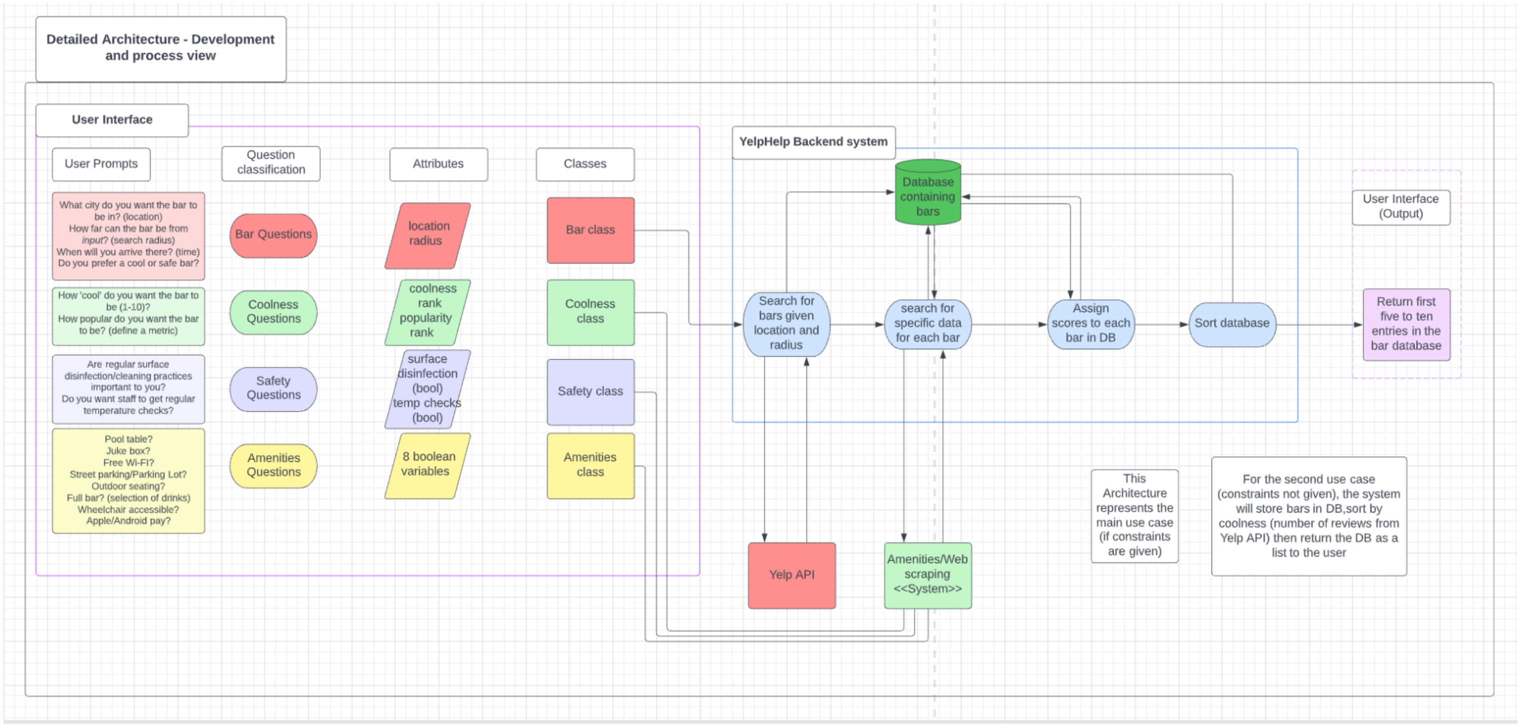
Architectural Designs:

Conceptual View:


Location Input System
Bar Database
Coolness/Safety Preference Input System
Amenities
Bar Ranking System/Algorithm

The conceptual view diagram is updated with a variety of amenities which are based on considerations to both safety (Ex: Masks Required) and coolness (Ex: Pool Table).

Development View:


Detailed Architecture - Development and process view
User Interface
User Prompts
Question classification
Attributes
Classes
What city do you want the bar to be in? (location)
How far can the bar be from *input*? (search radius)
When will you arrive there? (time)
Do you prefer a cool or safe bar?
How 'cool' do you want the bar to be (1-10)?
How popular do you want the bar to be? (define a metric)
Are regular surface disinfection/cleaning practices important to you?
Do you want staff to get regular temperature checks?
Pool table?
Juke box?
Free Wi-FI?
Street parking/Parking Lot?
Outdoor seating?
Full bar? (selection of drinks)
Wheelchair accessible?
Apple/Android pay?
Bar Questions
Coolness Questions
Safety Questions
Amenities Questions
location radius
coolness rank popularity rank
surface disinfection (bool) temp checks (bool)
8 boolean variables
Bar class
Coolness class
Safety class
Amenities class
YelpHelp Backend system
Database containing bars
Search for bars given location and radius
search for specific data for each bar
Assign scores to each bar in DB
Sort database
Yelp API
Amenities/Web scraping <<System>>
User Interface (Output)
Return first five to ten entries in the bar database

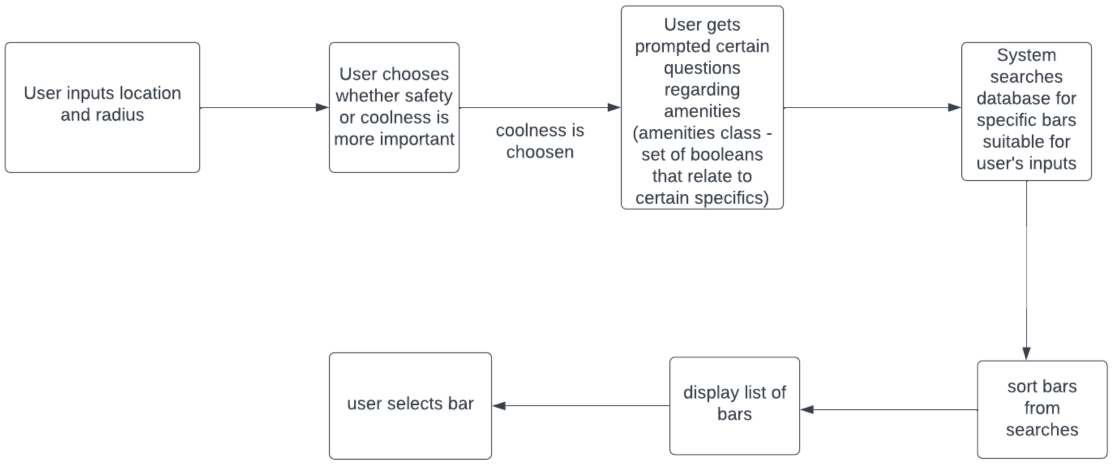
This Architecture represents the main use case (if constraints are given)

For the second use case (constraints not given), the system will store bars in DB,sort by coolness (number of reviews from Yelp API) then return the DB as a list to the user

This architecture is based on a development view that expands upon the user interface, class hierarchy/attributes in each, and the process for the rest of the system in the context of all other systems that will be interacted with. The system begins by prompting the user with a series of questions including desired location, search radius, coolness/safety questions, and a series of amenities questions. The classification of these questions as well as some of the attributes that will be listed within those question groups (classes) is represented in the architecture. The interface then stores all of these attributes into one of the four classes depicted above.

Next, the main functionality of the system is implemented, where the certain classes move onto certain stages of the remainder of the system. First, the bar class (driver) will request to search the Yelp API for all bars matching the inputs, this is then stored in the database, Next, the system will search for specific data (amenities offered, time open/closed) for each bar in the database by interacting with the web scraping system, storing into the child classes and then storing objects into the database. Finally, the system begins to assign scores to each bar in the database, following some internal algorithm to determine how scores will be assigned. The architecture ends by then sorting the database and returning recommendations to the user via the UI.

Behavioral View:


User inputs location and radius
User chooses whether safety or coolness is more important
coolness is choosen
User gets prompted certain questions regarding amenities (amenities class - set of booleans that relate to certain specifics)
System searches database for specific bars suitable for user's inputs
sort bars from searches
display list of bars
user selects bar

This is the behavioral diagram for the code's amenities section and how it will be implemented in the system. The user must first specify the desired location and radius. The driver will next inquire as to whether the issue of coolness or safety is more important. If coolness is selected, the user will be asked a series of questions like free wifi, mask requirements, outside sitting, and wheelchair accessibility. After those questions are answered, the system will search the database for bars that have all of those options, score them, and then present the user with a list of bars from which to choose.

Implementation can be found in the 'source code' and 'tests' subfolders under the sprint two folder.

Software Testing:

Software testing is comprised of multiple phases and sub-parts, including development, release, and user testing. As our 'customer' can be considered to be internal (to our organization), we provide minimal specifications and details about user testing, instead, we allow relevant team members to act as the user and provide relevant feedback.

Testing shall be broken down into three main parts, development, release, and user testing. Many of the unit tests/test cases are automated tests that can be found in the file "Test_driver.py". This document will discuss the breakdown of tests, the outputs, and the relevant validation of tests. The outputs of each test will not be provided in this document, as they can be generated by running the test file.

Development Testing:
Composed of three sub-parts: Unit tests, component tests, and system tests

A) Unit Tests

We create a series of unit tests where each of the inputs is modified. Our main test revolves around ensuring that we are recommended the correct bar (safest/coolest), the bar contains correct data, the list is outputted in order, and the program provides all queried data.

We define the () unit tests below (inputs) and validate the observed behavior (against expected). All tests can be found under the unit_Tests method in the testing file

Test One:

Inputs: loc = New Brunswick NJ, rad = 1610, Coolness, more inputs, [yes for all amenities questions]

For this unit test we find no issues, the recommended bar is Destination Dogs with a coolness value of 62.5, it is the highest in the list and the subsequent list is also sorted properly. All supplied data is also correct and no data is missing

Test Two:

Inputs: loc = New Brunswick NJ, rad = 1610, Safety, more inputs, [yes for all amenities questions]

For test two we recreate test one but instead focus on safety. Again the outputted bar is the safest in the collection “Blackthorn Restaurant…” with a safety value above 95. All printed bars in the list are sorted, and the recommended bar contains valid data.

Test Three:

Inputs: loc = Detroit Michigan, rad = 5000, Coolness, more inputs, [yes for all amenities questions]

For test three we vary the inputs greatly, focusing on the location Detroit Michigan but also increasing the search radius to over three miles to increase the load on the system. We are given a bar named Bronx bar which has a coolness value over 78, and is the highest on the list, all data is valid and the list is ordered correctly.

Test Four:

Inputs: loc = Detroit Michigan, rad = 5000, Safety, more inputs, [yes for all amenities questions, except two]

In this test we use the same location but vary the priority to safety. The outputted bar is TommysDetroit Bar and Grill with a safety value over 95, all data is valid and the list is ordered correctly.

Test Five:

Inputs: loc = Los Angeles CA, rad = 1610, Coolness, more inputs, [yes for all amenities questions]

For test five we use the final location input, and are provided with a bar Frank N hanks with a coolness value of 89, it is the coolest in the collection and contains all valid data. The list is also provided and contains the correct formatting/ordering

Test Six:

Inputs: loc = Los Angeles CA, rad = 1610, Safety, no more inputs

For test six we are provided with a bar named Paper Tiger Bar with a safety value of 108.5, which is the highest in the collection. Although it may seem invalid for the value to be greater than 100, it is allowed since additional points are assigned to a bar if it meets all safety criteria and requires masks to be worn. The outputted list is in the correct order and all data is valid

All unit tests provide valid outputs. The observed behavior of the system matches the expected behaviour for every unit test.

B) Component Tests

The component tests will focus on testing the API and pushing it beyond defined limits. We also focus on testing the Amenities components by varying the inputs. The tests can be found under the component_tests method.

<u>Test One:</u>

Inputs: loc = Jersey City NJ, rad = 40000, Coolness, more inputs, [True for all amenities questions]

For this test, the goal was to overload the system with a large dataset. Using Jersey City as a location with a search radius equal to the defined maximum will end up potentially overloading the system and causing it to crash. That doe snot occurs, and the outputted bar is the coolest, provides valid data and a valid list of cool bars. The bar is pier 13

<u>Test Two:</u>

Inputs: loc = London UK, rad = 2000, Coolness, no more inputs

This test case attempts to focus on the API and API interface by providing an international location to the system. The system is still able to handle the inputted data and outputs a bar named Connaught Bar located in Mayfair with a coolness value of 68. The supplied data is also valid and the list of bars is in the correct order.

<u>Test Three:</u>

Inputs: loc = San Fransico, rad = 15000, Safety, more inputs, [False for all amenities questions, but three]

Inputs: loc = Detroit Michigan, rad = 5000, Safety, more inputs, [One False, four True]

This test case focuses on overloading the API interface and creating an edge case for the amenities system since all amenities inputs vary greatly. The system however is able to respond well to this test. The recommended bar is Tommys Detroit Bar and Grill with a safety score of 95.75. All supplied data is valid and the list is in the correct order.

<u>Test Four:</u>

Inputs: loc = Los Angeles CA, rad = 1610, Coolness, no more inputs]

This test case is almost identical to one that was used in the unit tests but changes the amenities parameters. The recommended bar is named Frank N Hanks with a coolness value of 89 (as expected) and a safety value over 100. The bar offers more amenities than the user desires, validating th high coolness value. All provided data (including the list) is also valid.

Test Five:

Inputs: loc = Moscow Russia, rad = 100000, Safety, no more inputs

This test case violates the previous pattern as it returns an error and a status code of 400. We expected this as the search radius is defined as 100,000 (greater than the limit), so the API rejects the request causing the status code to print as 400.

C) System Tests

We end the development testing phase with system testing where we focus on testing individual systems that are composed of smaller components within our larger system. We focus on calling specific methods to test their function, on API interfacing systems, as well as sorting and searching-systems.

Test One:

Inputs: loc = Jersey City NJ, rad = 40000, Coolness, more inputs, [True for all amenities questions]
Function: sort_bars_and_attributes

This test focuses on the sorting method with a range of inputs. The System provides pier 13 as the recommended bar which is correct. All data and bars are valid and in line with previous results

Test Two:

Inputs: loc = London UK, rad = 2000, Coolness, no more inputs
Function: sort_bars_and_attributes

The system tested is the sorting and storing system responsible for taking raw data from the Yelp API and storing it into local objects/classes. This test is passed as the system returns "The Social" as the recommended bar, and all relevant data is valid.

Test Three:

Inputs: loc = San Fransico, rad = 15000, Safety, more inputs, [False for all amenities questions, but three]
Function: access_API

This test ensures that the API interface is working to specification. The system does not output anything as this interface is meant to pass data to the next system, but no errors have occurred which means the interface passes this test

Test Four:

Inputs: loc = Detroit Michigan, rad = 5000, Safety, more inputs, [One False, four True]
Function: access_API

This final test checks to see if the API interface is still able to work for a variety of amenities inputs and for a different location and radius input set. The system does not print anything (Which means it's operating properly), and no errors are raised ensuring the interface has been able to parse data from the API correctly.

Release Testing:

A) Requirements based Testing:

We design tests that ensure all requirements are met by the system and should serve as requirements checks. The details are given below:

Initially, we defined a set of requirements (user/system and functional/non-functional), we can create a set of tests corresponding to all of these requirements in order to ensure all requirements have been satisfied by the system

The system will recommend a bar that is cool (hosts a set of amenities, has high traffic during peak hours, is open often, and has good reviews) and safe (features of a set of safety regulations, does not host too many people at a given time).

User & System Requirements:

The system implemented focuses on recommending a bar that is cool (Defined as having a set of amenities and good reviews) and safe (having some regulations and not having too many people in the bar at a given time). The system requirements expand upon the design and specifics of the system and how it will be implemented.

We neglect to design a specific test for the user requirements as it can be easily explained through any of the unit tests. The unit tests provide a bar that maximizes either coolness or safety but still ensures that the other metrics are as high as possible. The details can be found in the unit testing section and code.

System Requirements:

1. Given some location and search radius, the system will store a collection of bars in-line with the given parameters.

Does the system store a collection of bars in-line with the location and radius inputs?

We design a generic test that will provide a location and radius and check to see if the output follows a list format and if one specific bar is recommended/

2. The system will gather and store data (amenities and safety regulations) with respect to each bar.

    Does the system store data for each bar found?
    We design a test to ensure that given some bar the system is able to populate the relevant objects with information about safety regulations and amenities. The component tests provided in the test program showcase this, the output of all of the tests includes information about safety regulations and amenities.

3. The stored list of bars will be sorted by an inputted metric (either by safety or coolness).

    Are bars sorted by the inputted metric?
    All tests that have been given so far, serve as proof to this test. Each list outputted to the user is sorted by either coolness or safety.

4. The system will output, to the user, at least one bar that matches the constraints.

    Does the system recommend a bar that matches the inputs?
    Unit tests one and two provide more detail, but the system is able to recommend a bar that follows the location, radius, priority, and amenities list detailed by the user.

5. If few constraints are provided, the system will provide the user with a list of bars following whatever inputs are provided.

    Does the system provide a list of bars if no constraints are provided?
    The system is able to provide a user with the option of not providing more constraints. The question ‘Do you want to provide more constraints is outputted to the user, where their choice affects the termination of the program.

B) <u>Scenario Testing</u>

Different outputs are clearly defined depending on if the user prefers coolness or safety and if the user responds in the affirmative or not to each of the questions relating to the various amenities available. This shows that the code has great detail and allows for many different scenarios or circumstances.

C) Performance Testing

To ensure the system is performing well enough we generate a test to determine if it can operate at peak conditions. These conditions are defined as a radius of 40,000m and a location that is a large metropolitan area (London,NYC,ect…). Since these tests have been implemented in the component tests, we do not need to specify a specific test but we can measure the performance of the system in these edge cases through runtime.

For this test case the system took between 70 and 100 seconds to provide all data to the user, the average runtime being under 60 seconds. This means that the system is able to perform within a reasonable timeframe but it is not ideal. The web scraper in particular does not perform very well, causing the system to take longer to process and output data to the user. But the performance requirements, as detailed in the non-functional requirements are still met.

User Testing:

User Evaluation:

Several amenities - which take typical customer requirements into consideration - are included in the implementation of the YelpHelp design. The user's responses to the boolean questions for each amenity are factored in to create the final sorted list. The system is arranged and implemented in a very thorough way that places much importance on the customer of the bar. The system allows the user to determine whether coolness or safety factors are more important. It also offers various questions regarding free wifi, masks requirements, pool table, outdoor seating, etc to fulfill different customer needs. The intended requirements are fulfilled by the system. Therefore, the system is ready to deploy.

Evaluation:

Requirements Engineering

- Are all requirements listed clearly?

The Requirements are listed clearly and broken down into relevant parts. The main breakdown of the requirements is user and system as well as functional and non-functional. All details are provided clearly in the requirements specification

- Easy to understand and written thoroughly?

Requirements are also written thoroughly but also easy to understand. The structure and breakdown of the requirements allow the reader to gather enough detail to truly understand the system, but also allow him to easily understand what is going on.

- Are the requirements and the basic functionality of the system similar?

The requirements are given in a straightforward and understandable manner. Each criterion expresses a particular concept. The system's basic job is to recommend a cool and safe bar. All of the sprint's stated requirements contribute to the goal of proposing a cool and secure bar. All of the requirements are described in a consistent manner.

System Modeling

- Are there enough diagrams for each part of the system?
- Each diagram shows a clear understanding of the different parts of the system and how it will operate

A use case diagram, sequence diagram, process model and a state diagram are all included in this sprint. Each model comes with a detailed explanation to help better comprehend the thought process behind it. Each diagram demonstrates a thorough understanding of the system's many components and how they will work together. The use case diagram is revised to include use cases for entering specific coolness and safety aspects that the user considers significant. The sequence diagram is revised to show how the yelphelp app considers the users inputs and ranks the bars. The diagrams display the inputs of both the coolness and safety factors that the user will take into consideration. The process model is revised to show how the constraints work in the system. It shows how the bars will be outputted if there is or isn't any constraints indicated by the user. The state diagram is revised to depict the system as well as the factors that create them to change.

Design and Implementation

- Code is clear and easy to understand?

The source code has been organized and formatted so that it is clear and easy to understand. All methods, attributes, and variables are named within the context of their function. All methods are organized based on the hierarchy of the system and what is intended to do.

- Code works with no errors.

The source code is able to work with no errors and exceptions raised. Since python interprets the code and does not compile it, we can not determine if errors are present at runtime, instead, we know that no exceptions or errors occur for the vast majority of test cases.

- Variables and functions are used correctly.

All variables and functions are used correctly, as a result, the system is able to run to specification and provide all relevant data to the user.

Architectural Design

- Designs are well organized and portray a clear understanding of how the system will work

Each architectural model depicts how the system should be organized. The conceptual, developmental, and behavioral views are all shown and explained thoroughly. Different perspectives are shown within the diagrams to have a better understanding of the system altogether. The conceptual view depicts how the user interacts with the system by inputting the location and then considers the list of coolness and safety factors. Then, the algorithm outputs a list of allotted bars from highest to lowest. The developmental view depicts the user interface and the yelphelp backend system. The system asks the user a series of questions, such as their preferred location, search radius, coolness/safety questions, and a series of amenities questions. These are then stored in the various classes where the driver will ask the user certain questions pertaining to their specific interests. After that, the system will assign scores to each bar and show the output in rank from highest to lowest. Lastly, the behavioral view depicts how the amenities class will be inputted into the system. Once the coolness factor is chosen as more important, the driver will ask a list of questions pertaining to the amenities such as availability of free wifi, if there are pool tables, requirement of masks, outdoor seating availability, etc. After that, the system will rank the list of bars ranking from highest to lowest.

Software Testing

- Outputs as expected.

The system outputs as expected. Every test case used results in a satisfactory output, with the exception of one test case which generates a status code error. That test case, however, was intended to raise that error as some of the parameters exceed the limits provided by the Interface being tested.

- Multiple test cases are used.

The software tests are broken up into multiple sections, development, release, and user. And each subsequent section is broken down into smaller sub-sections. The development testing section employs over fifteen test cases across a large input set, and all remaining parts detail the implications of those tests in the context of the type being discussed (requirements, etc.)