# Class Diagram

From my previous experience with the CIT 262 class, as well as from different readings which I will be citing in this portfolio entry, class diagrams are a necessary part of each well-developed application. It provides you with the opportunity to diagram the model, view, and control layers of your application and its associated methods. It is another diagram of many which allows you to discover your application, flesh it out, and understand exactly what you are wanting to accomplish and how.

From tutorialspoint.com, they listed the purposes of the diagram is the following (<http://www.tutorialspoint.com/uml/uml_class_diagram.htm>):

* Analysis and design of the static view of an application
* Describe responsibilities of a system
* Base for component and deployment diagrams
* Forward and reverse engineering

A class diagram begins with a class. You list your classes in separate class entities with their titles at the top, their attributes in the following section, and finally their methods in the bottom section. The +/- on each attribute/method describes if the item is a public or private item respectively. The method section allows you to name what inputs you expect and what output type it will provide. You can also place specific models into a package, which allows you to group them by function.

Once you have developed your classes you can begin the development of their associations, aggregations, dependencies, composition, and inheritance. Each of these is documented through different types of arrowed lines. A dashed line with an arrow at the end shows a dependency from the object the arrow begins at to the object it is pointing to. Inheritance and association are a solid arrowed line in the direction from the child to the parent. Association is a line between two objects showing are basic relationship, but no dependency between the objects. Composition is shown by a line with a black diamond where it shows that the child class is part of the parent class, and will only exist if there is a parent class. (<https://nirajrules.wordpress.com/2011/07/15/association-vs-dependency-vs-aggregation-vs-composition/>)

I created a class diagram for my application, Youth OnCall, and we also created a class diagram for our group application. Our group diagram can be found at (<https://drive.google.com/open?id=0B5_UhQZPuf-5QUVLNTY0TmFXalE>). The diagram for my personal project can be found at (<https://github.com/hodges-olan/YouthOnCall/blob/master/YouthOnCall/src/Documentation/Class%20Diagram.vsdx>).

# State Diagram

Our classes quite often will go through different phases in their lifecycle. In an effort to help developers comprehend the state a class can go through, a state document can be created. According to tutorialpoint.com, the main purposes for a state diagram are the following (<http://www.tutorialspoint.com/uml/uml_statechart_diagram.htm>);

* To model dynamic aspects of a system
* To model lifetime of a reactive system.
* To describe different states of an object during its lifetime.
* Define a state machine to model states of an object.

In my personal project, I created a state diagram for the two separate model classes that exist and their states throughout the use of the application. The link to this file is here:

<https://github.com/hodges-olan/YouthOnCall/blob/master/YouthOnCall/src/Documentation/State%20Diagram.vsdx>

In my application, Youth OnCall, jobs can be created by members for youth to pick up and complete for either service or pay. The job starts of as an unsubmitted job into the system. Once the job has been submitted, it enters its first state as an Open Job. Once a youth accepts the job, it enters the accepted job state. The youth can then either complete the job, or they can decline the job and have it re-enter the open job queue. Members can also cancel their jobs or complete them manually as needed. The final state for a job is the completed job state where they will stay for reporting purposes.

The same goes for the member’s class. They start out as unregistered members and then transition to either new members or new youth based upon their age. The youth can then transition to members once they turn 18. If they decide to unregister, their account becomes disabled, but is still available for reporting purposes for their associated jobs that they submitted and completed.

# Threads, Executors, Runnables, & Handlers

This was my favorite topic by far out of all of the topics we’ve reviewed this semester. I enjoyed the client and server sockets as well as HTTP URLConnection topics, but this is a subject I’ve always wanted to know for languages that do not natively perform concurrency for you. I’ve mostly dealt with web based applications so the web server helped with the concurrency for me on a user basis, but I’ve never known how to do concurrency outside of that.

Well, long story short, our CPUs are thread based. Whether you have a really old single core and single threaded CPU, or you have a more modern multicore/multithread CPU like today, you are utilizing threads whether you like it or not. Now, your application isn’t tied to just how many threads your CPU has, but that would be a good idea for a limitation on your application. These threads are all running their own tasks and all at the same time. This allows us to send the application off on a task while the user can still perform other tasks, thus keeping from slowing them down and frustrating them.

Each thread will be given a runnable, which is basically a class of your own creation that has code for it to do. Managing each thread and which runnable goes to which would be too tedious for us to manage ourselves, so that is where the executors come in to play. You tell them how many threads to create and it will manage that many for you. It also will create a queue in the event that all of the threads are currently working on other runnables.

Handlers and loopers are Androids equivalent of an Executor. It allows you to send messages to threads and it will manage a queue for each thread, but the handler is only tied to a single thread. This means you have to make the decision on which thread to send your tasks to.

Below is a link to a recording I made on this topic and that I shared with the class just before sending this to you.

<http://youtu.be/xClwSZ-StfU>

Below is a link to the code that I made as part of this to test out the idea.

<https://github.com/hodges-olan/CIT360-Portfolio/tree/master/CIT360-Portfolio/src/ThreadsExecutorsRunnablesHandlers>

And finally here are the links I utilized to learn about this topic. The first is the entire tutorial on concurrency from Oracle which is also in our topics reference list. The second is a YouTube video where the example I utilize was built and explained. I modified his example to include an executor as he does not utilize this method. And the final two are a long, two part series on java concurrency and gives a great explanation on how to utilize them, how they inherit from each other, etc.

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

<https://www.youtube.com/watch?v=br_TEuE8TbY>

<https://www.youtube.com/watch?v=2NAlvSkK9-k>

<https://www.youtube.com/watch?v=nD8kTrnKH2Y>

# Use Case Diagram

Use case diagrams allow you to see how the system in its entirety functions. Users (actors) interact with the system often for different purposes. This diagram allows you to show which type of users will interact with which portions of the application. It also allows you to show the basic associations between other use cases. (<https://en.wikipedia.org/wiki/Use_Case_Diagram>)

Tutorialspoint.com lists the following purposes for the Use Case diagram (<http://www.tutorialspoint.com/uml/uml_use_case_diagram.htm>):

* Gather requirements of a system
* Get an outside view of a system
* Identify external and internal factors influencing the system
* Show the interaction among the use cases and actors.

We created a use case diagram for the group application (<https://drive.google.com/open?id=0B5_UhQZPuf-5NlZfLW9neGpfczQ>) It depicts the two separate types of actors that can interact with the system and what their role would be in relation to the application. There is also some extensions between use cases for specific actions depending on if the user requires it or not.

I also created a use case diagram for my own personal application (<https://github.com/hodges-olan/YouthOnCall/blob/master/YouthOnCall/src/Documentation/Use%20Case%20Diagram.vsdx>) It depicts the three separate actors and their associated use cases.

# Use Case Document

A use case document is an appendage to the use case diagram. Each use case from the use case diagram will have a page in the use case document. Inside this document, it will describe what the use case title, actors, pre and post conditions, a description of what the use case diagram is all about, the main success scenario, contingencies to each scenario when a failure can occur, and a priority amongst all of the other use case documents.

I watched a recording from a business analyst on Youtube that described in detail the purpose of a use case diagram and document and how to build them. His explanation was very straight forward and helped me more fully understand it’s purpose. (<https://www.youtube.com/watch?v=nN7lTDWKP6g>)

I documented the use case document for my own application at the following location (<https://github.com/hodges-olan/YouthOnCall/blob/master/YouthOnCall/src/Documentation/Use%20Case%20Document.docx>) and we placed multiple use case documents in our project repository as we split the work up to accomplish it quicker (https://drive.google.com/open?id=0B5\_UhQZPuf-5U0I3SU9CMUQ4V1k) They are all labeled with UC-# on them. I was responsible for [UC-3 Add-Remove Members Use Case Document](https://drive.google.com/open?id=0B5_UhQZPuf-5b2Npa2QwTjdTN2s)