Name: Michael Gohde
Project Name: Process Manager
Project Description: Process manager is a tool that distributes tasks as defined in scripts to a set of executor nodes.

List the features that were implemented:
Please note that the original features table had a total of 10 IDs:

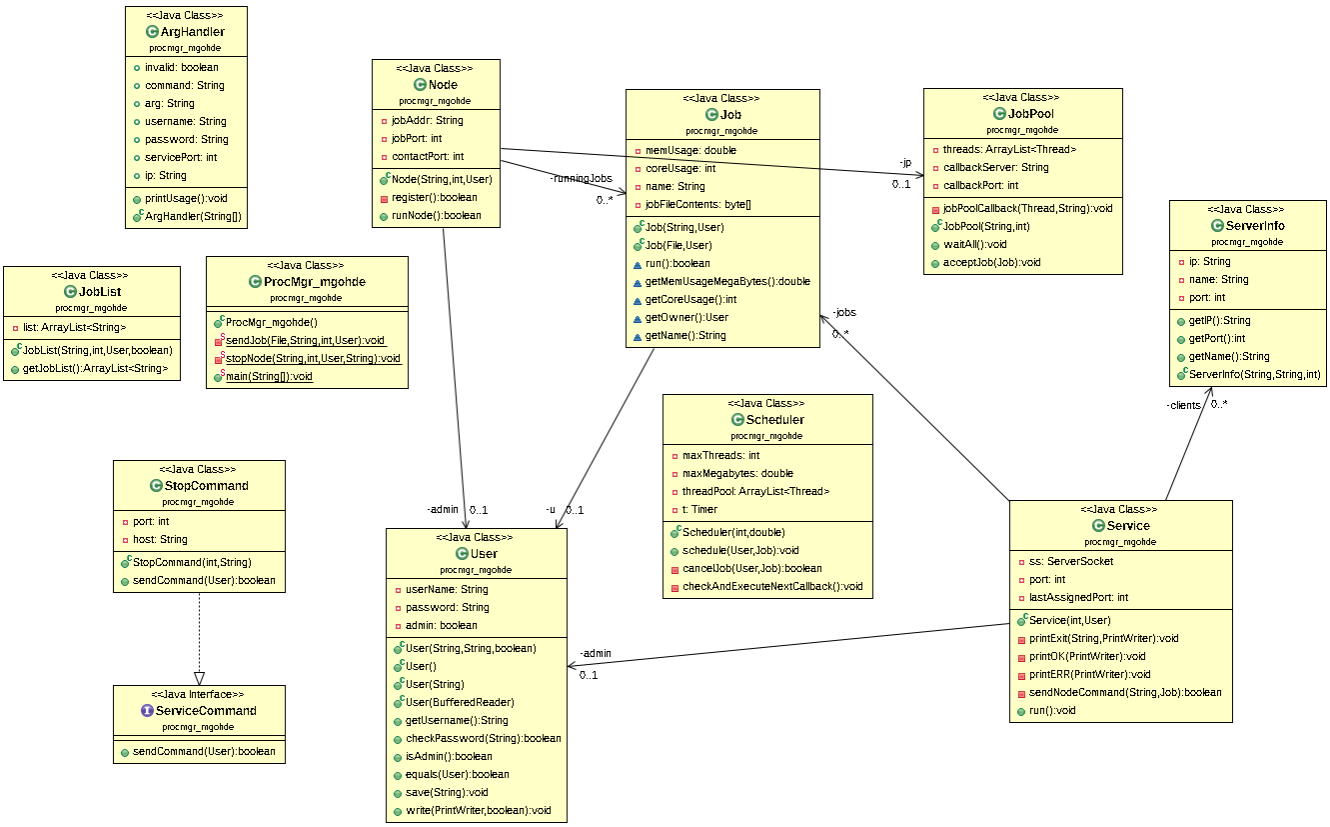
ID	Feature
1	The system is able to execute jobs as represented by Java classes • Each class should inherit and extend attributes from a common interface. • Using a Java Runnable would be a good choice. REVISED TO: The system is able to execute jobs encapsulated within a job class. • The Job class is serializable for network communication. • Jobs are read from shell scripts with additional formatting much like existing projects such as Slurm.
4	Jobs specify their approximate resource requirements. NOTE: It is possible for the Job class to read and transmit a job lacking resource requirements, as such specifications are up to the user.
7	All users are able to list the job queue.
8	All users areable to log in and authenticate
9	Users are able to cancel their own jobs once submitted
11	The scheduler is able to schedule jobs for execution.
12	Execution nodes spawn worker threads to execute jobs.
13	Administrators may terminate running job services.
14	Administrators may terminate the scheduler service.

List of features that were dropped:

ID	Requirement
2	Administrators must be able to terminate jobs. (Only a user may terminate their own jobs, but administrators now have the ability to terminate job servers).
3	Administrators must be able to specify available resources to the job scheduler. (Resource considerations have been dropped in favor of a simpler round robin scheduling system)
5	The job scheduler must be able to track the resource requirements of running jobs and

	compare them to available resources. (Resource considerations have been dropped in favor of a simpler round robin scheduling system)
6	The job scheduler must be able to queue jobs that cannot be run immediately due to a lack of resources. (The job scheduler merely keeps track of jobs. At present, it submits jobs to a separate execution service that spawns worker threads to handle request. As such, this feature no longer makes sense given the present state of the implementation.)
9.1	Administrators must be able to cancel any job currently running or in the queue. (Unfortunately, users may only cancel their own jobs. Administrators are, however, able to terminate entire job servers).
10	Administrators must be able to create and manipulate user accounts (The concept of user accounts has been constrained substantially. Usernames and passwords are instead specified at run time, with the user specified to the server and node services as they start becoming the de facto administrator of those services).

Final class diagram (please note that every class is used by the methods of at least one other class).



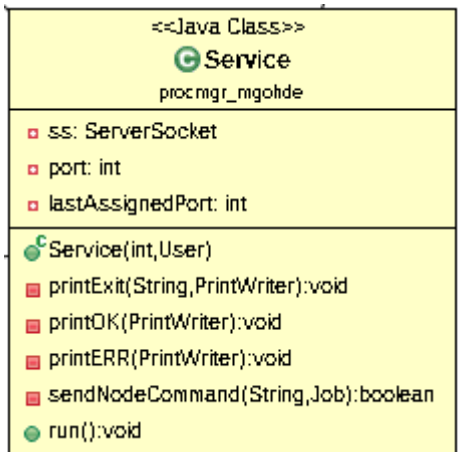
It should be noted that a fair amount changed between the initial class diagram and the final implementation. The number of features that were cut or otherwise altered dramatically due to both redesigns and time constraints played a large role in this change.

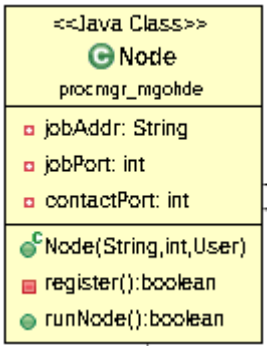
Most of the changes were practical: as implementation started, better means of going about implementing the project (particularly from a speed of implementation standpoint) became apparent. This explains the fact that much functionality, particularly related to the User class, went unused.

Design Patterns:

The first set of design patterns I would like to highlight are particularly interesting because they would not reasonably appear on a UML diagram. Instead, they are representative of the overall behavior of the project. It should be noted that there are fundamentally three major components to the project:

1. The client, which is a means of sending commands and job scheduling requests
2. The "Service" (queue or job management service), which maintains a list of clients and running jobs in addition to forwarding jobs on to executor nodes.
3. The "Node" (executor node), which maintains a thread pool and accepts jobs for execution. This component automatically registers itself with a "Service" on startup and is assigned an internet protocol port over which to communicate with the "Service."





(Client not shown as it represents the composite behavior of numerous classes and methods).

This architecture alone should yield the following design patterns:

1. Interpreter: each network component of this project sends and/or receives commands from another component over a socket. This means that they may only communicate in bitstreams, which require parsing and interpretation.

2. Strategy (only a weak case can be made): at run time, the “Node” receives an algorithm over the network, which it then executes. The “Node” in this case is functionally acting as an interface over the network for a set of algorithms that may be executed.

3. Observer: the “Service” may inform “Nodes” of its intent to shut them down as well as of various changes to the job queue. It maintains a list of such nodes as well as the jobs presently running with various callback functionality.

What have you learned about analysis and design now that you’ve stepped through the process?
First and foremost, I’ve actually applied the overall design process here (UML and careful consideration of all design features) to a project at work, which I believe to be substantially better designed than this project. I found it particularly interesting to see these patterns influence how I made network design related decisions, since the idea of having Nodes call back, register themselves, and communicate their progress back to the main scheduler service was new to me (particularly from an implementation perspective.) I will admit that, when designing this project, I continued with the slapdash “iterate until it works” mentality that had usually pervaded all of my prior projects. This explains why so many features were radically changed, dropped, added, and reworked over the past few months. I had a clear vision in mind and simply worked and reworked until it was somewhat achieved. As a part of this, I regret not being more deliberate in my design choices to start with. Given the opportunity to start from scratch, I would absolutely formally use more design patterns, especially in how I handle sending commands to each running component. The wasted potential of the StopService class and its interface is truly disappointing, since I could’ve easily replaced much of the deeply nested branch heavy logic with serialized objects or equivalent. Upon further reflection, such commands could have easily been serialized using the normal Java serialization interface and then cast to an interface before being executed. This would have left me with a clear cut example of a design pattern in code that doesn’t require networking tricks or substantial explanation to justify.