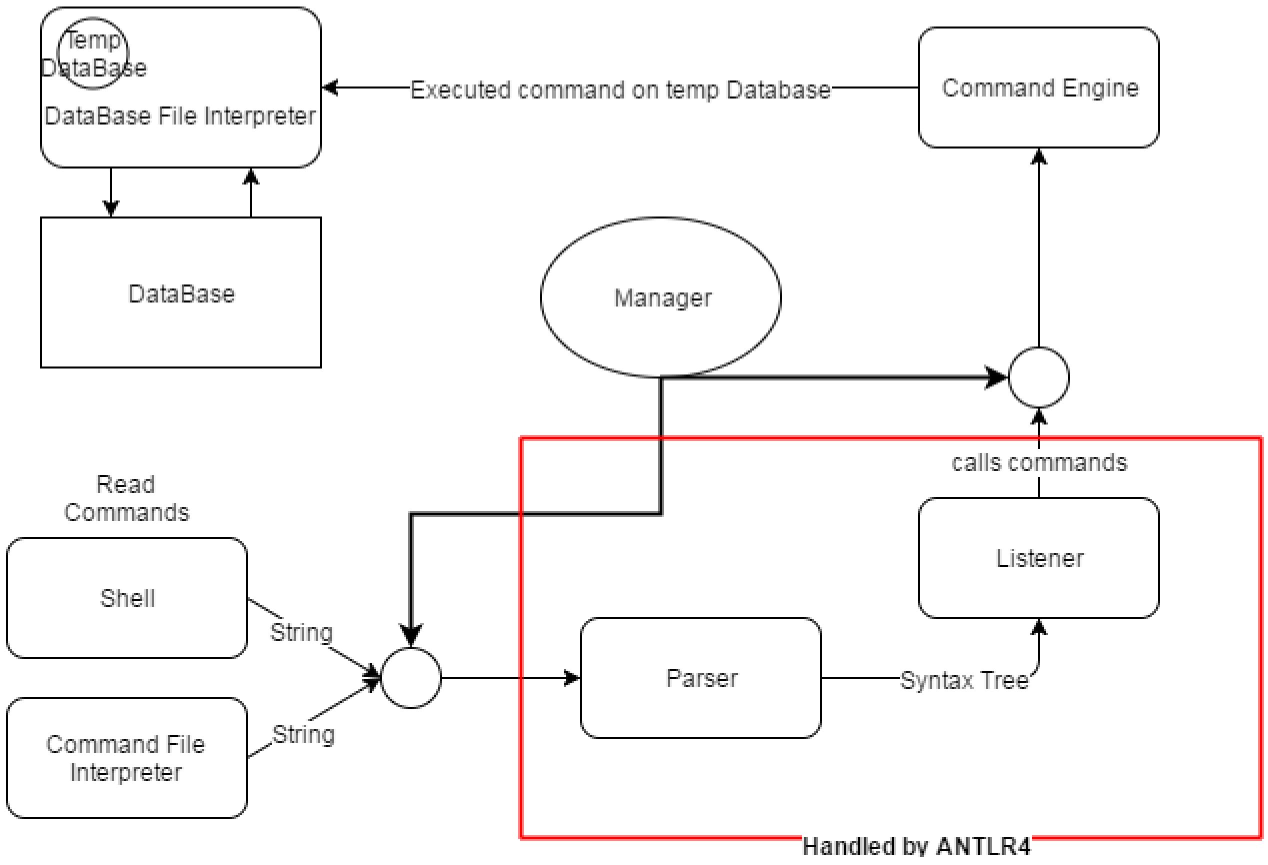
Section 1: Purpose of the Project and Sub-Systems

The overall purpose of this project is to implement the core functions of a simple database management system (DBMS) and test its functionality with real data. The DBMS we will be using is SQL and it will be based on relational algebra. The grammar for the data manipulation language is given to us in the project specification. The data we will use is the TAMU class schedule. The project is split up into two main phases. Phase One implements the DBMS and parses commands and queries into an abstract syntax tree using the parser generator, ANTLR 4. It also designs the interplay between the DBMS and C++ and implements this functionality using Test-Driven Design. Phase Two implements the database engine to actually carry out the commands and queries by traversing the abstract syntax tree sent to it from the parser and evaluating the nodes.

Section 2: High Level Design Entities



The project is divided into 4 parts. First the shell and command file reader which will read in commands either from the user or from a file and feed them into the parser. The parser will then interpret the string into a syntax tree. The syntax tree will be read by the listener which will call the command engine based on what it interprets the command to be. The command engine will perform modifications on a local copy of the database which will be handled by the database file interpreter. The command engine will ask to see the database, store a new version, or write the temporary database to a file. The database file interpreter will read from the file or write when appropriate. A stretch goal is to implement a manager that will allow for higher modularity and a multi-threaded design by handling buffers in between each module.

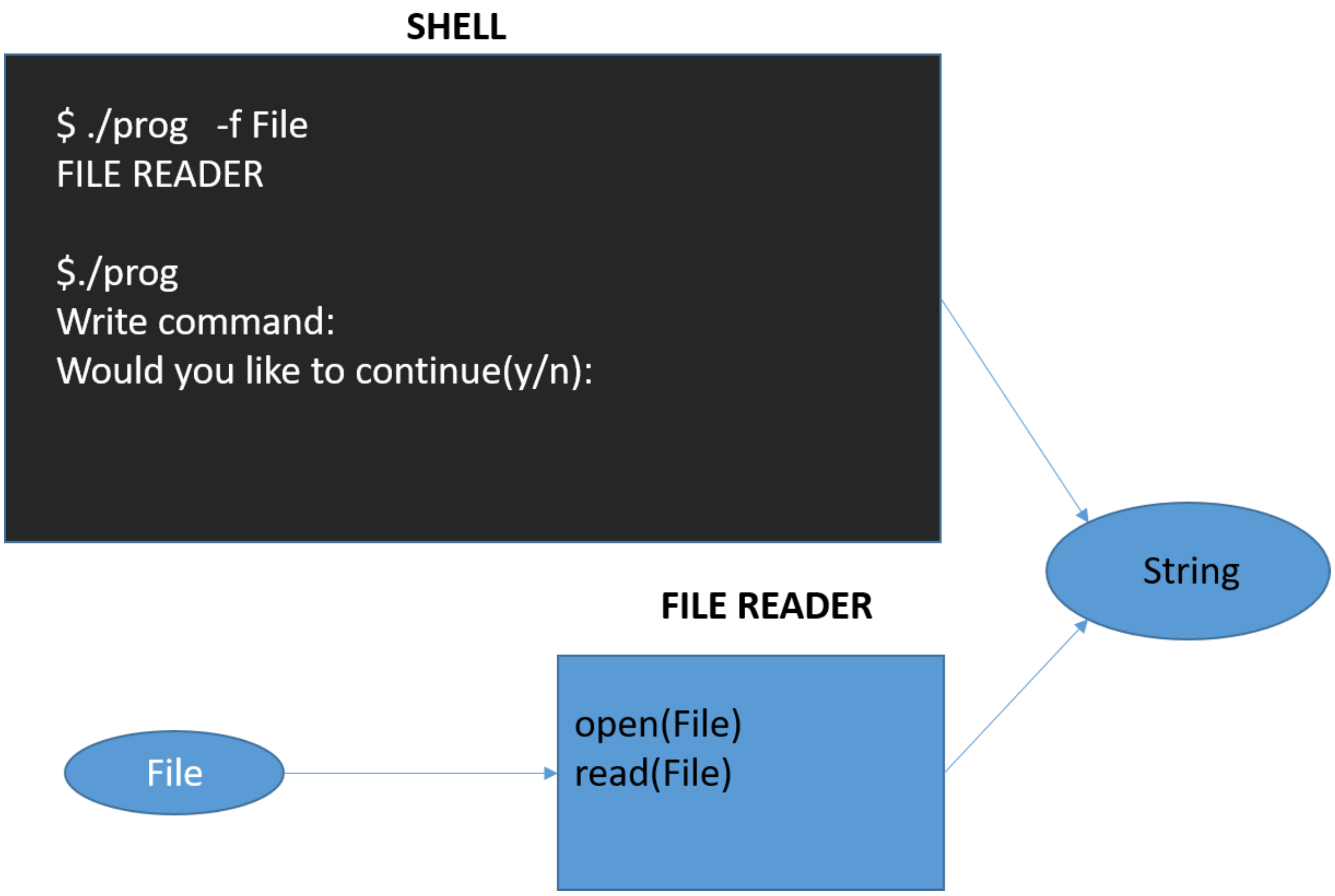
Section 3: Low Level Design Entities

Shell-File-Reader

Usage

The program should be able to read lines from the command line or from a file. If the program is run with a flag "-f File", the list of commands will be read from a file called "File" and sent to the parser. If there is no flag, the program will allow the user to write commands in the shell until the user would no longer like to continue.

Model



```
$ ./prog   -f File
FILE READER

$./prog
Write command:
Would you like to continue(y/n):
```

The shell is showing two different ways the program will compile. If the file flag is included we will implement the file reader and if it is not, we write a list of commands in the shell until we want to stop.

Interaction

As you can see from the high level entities design, the string outputted will be sent to the parser and the manager will overlook this.

Parser and Listener

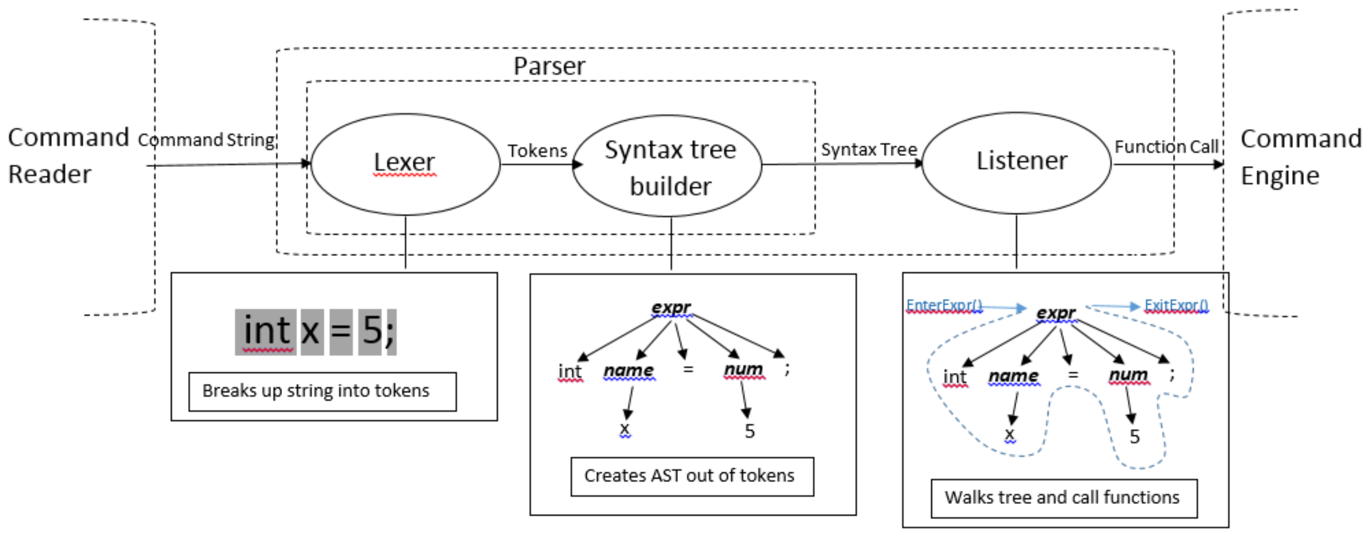
Usage

The parser and listener are components created in ANTLR4. They will receive commands as strings and call functions in the command engine. ANTLR4 will generate two separate files, the parser and the listener. The parser consists of two parts the lexer and the parser. The lexer will first tokenize the input into parts that can be used to build a syntax tree. The parser will then construct the syntax tree out of the tokens given by the lexer. After the syntax tree is created the listener will walk the tree and call the command engine as necessary.

These two parts will be interface as one component. Command strings will be given into the Parser and the Command engine will make calls to the command engine. If the manager system described above is implemented to use buffers, only then, will the function calls change to instead push calls and arguments onto a stack rather than directly calling them.

Defining these components as two separate parts in a module separate from the other components makes sense since both will rely heavily on ANTLR. These parts being separate also allows development to be done on all other components without knowledge of the inner workings. There are two major risks associated with this design. First, the parts may prove too hard to be worked on as one module and splitting them up may be more efficient. On the surface analysis it seems the benefits largely outweighs the risk. Secondly the listener requires knowledge on what each command in the command engine does and when it should call it. This risk can easily be avoided by having a well-defined interface for the command engine.

Model / Interaction



The Parser and Listener will receive a command as a string from the command reader. If the manager is fully implemented it will receive strings from it. Then it will tokenize the string through the lexer. Once it is broken up into the parts we need it will be built into a syntax tree. From there the syntax tree will be walked by the listener. The listener uses a depth-first path or pre-order traversal. As it enters and exits nodes of the tree functions are called. These functions will be overridden to perform the necessary commands in the engine. The listener will directly call these functions unless the manager is fully implemented which will instead have a queue to place function pointers in.

Command Engine and Database Interpreter

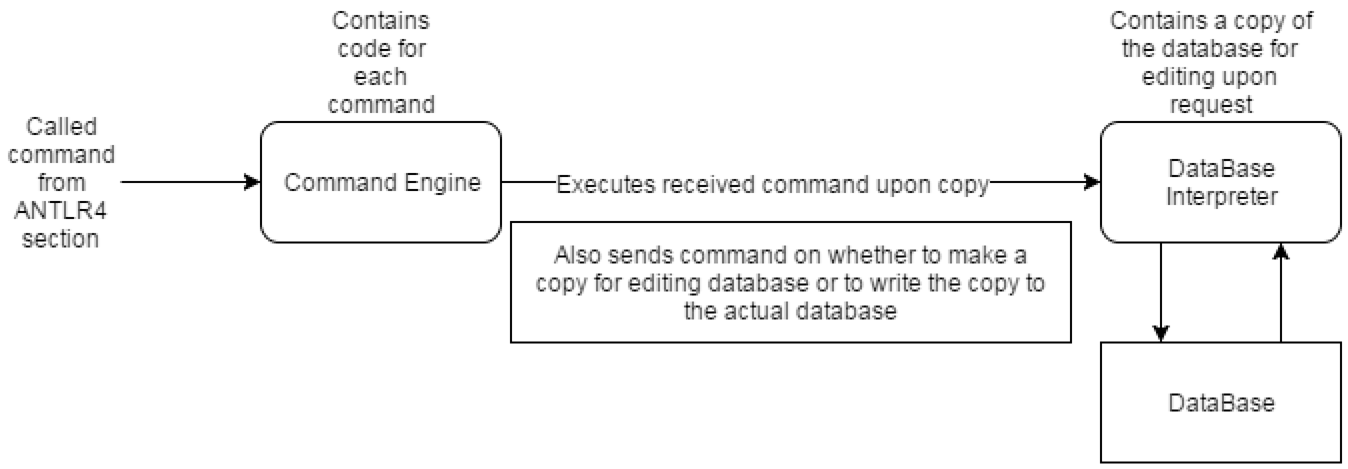
Usage

The command engine and database interpreter will together compose our database engine. This section of our code will manage interactions with the different database files. It will manage how info is stored and edited.

The command engine part contains the code for each command and query for the database. The database also processes each received request for a command and executes them upon database. The command engine will get valid commands from the parser, and either execute them upon the read database that is stored in the DB reader/writer or have the DB reader/writer execute commands such as write/read. The command engine should also be able to send the output of an executed command to the shell. The reason we have decided upon this way is that processing commands and queries can be implemented the same way, so it makes sense to do them all in the same place. Although the way we have designed the command engine may make it easier for each section of our code to interact with the database in a clean way, there's also the risk in integrating it in each section so that it will work.

The database interpreter part will request parts from the database for editing through the command engine. The requested info can then be modified and if a write command is received these changes will be written back to the correct database file.

Model


Called command from ANTLR4 section
Contains code for each command
Command Engine
Executes received command upon copy
Also sends command on whether to make a copy for editing database or to write the copy to the actual database
Contains a copy of the database for editing upon request
DataBase Interpreter
DataBase

The above model shows a general outline of how the command engine will work. It shows that the command engine will carry out the commands that the ANTLR4 section will call and will either send or execute them upon the database interpreter section.

Interaction

- With the ANTLR4 section:
    - The command engine will interact with the ANTLR4 section by executing each of the called commands upon the database interpreter. The database interpreter is just our way that we have designed to interact with the database stored on the disk.
- With the Shell/Output section:
    - The command section will interact with the shell by outputting the results of different commands to it if needed.

Section 4: Benefits, Assumptions, Risks and Issues

Benefits

The molecularity of our design provides two major benefits, firstly development can happen individually with a high level of autonomy. This will make the dev cycle much shorter and productivity higher. Secondly the potentially for a multi-process or multi-threaded design becomes much simpler (this is discussed further later). Another benefit according to the specification document is that using relation algebra allows the implementation effort of the DBMS to stay manageable.

Assumptions

1. We will be able to enter commands and the ANTLR4 section will be able to parse them according the required grammar the correctly.

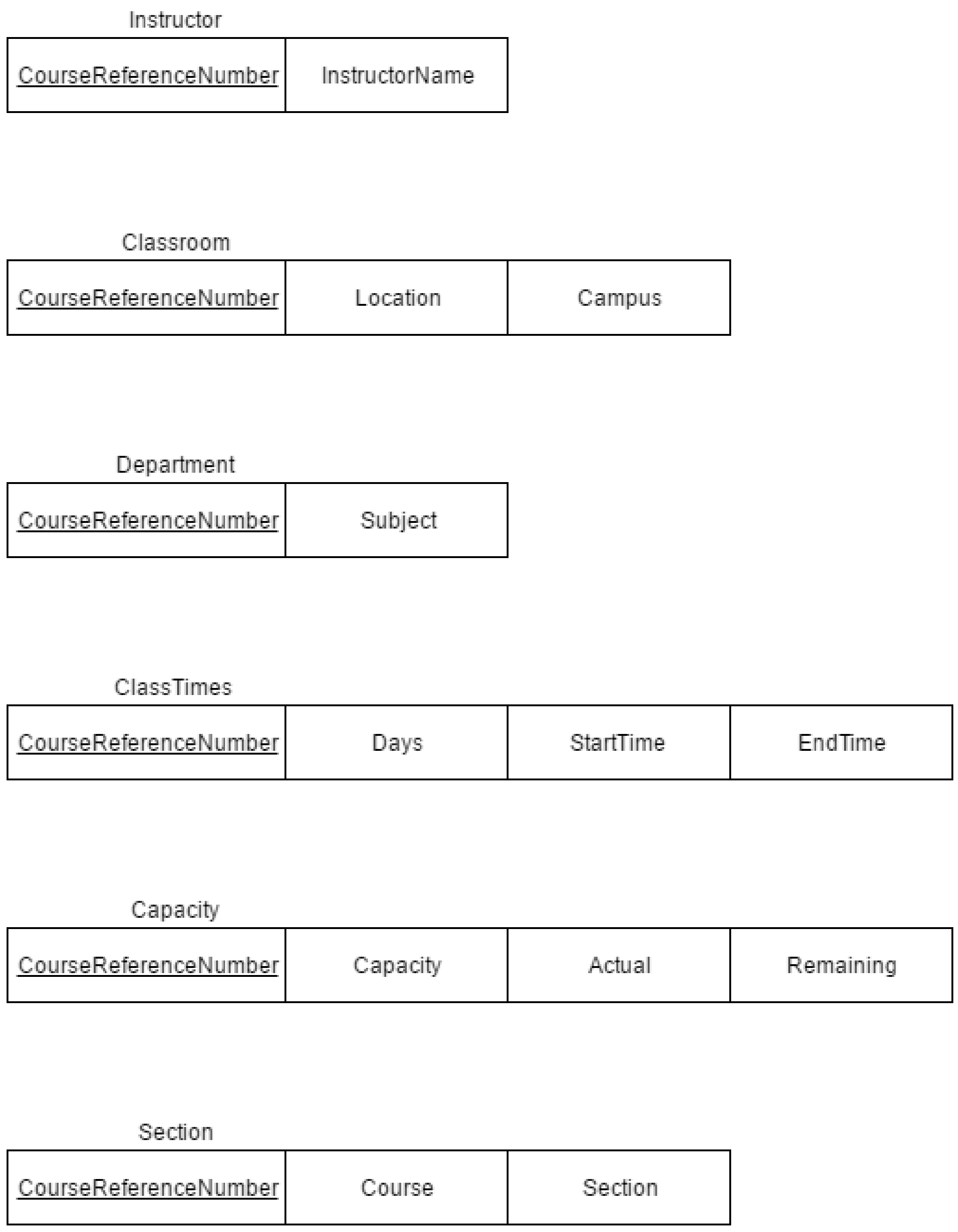
2. The command engine will be able to carry out each command the ANTLR4 section calls correctly.

3. We will store and modify the database in the required ways correctly.

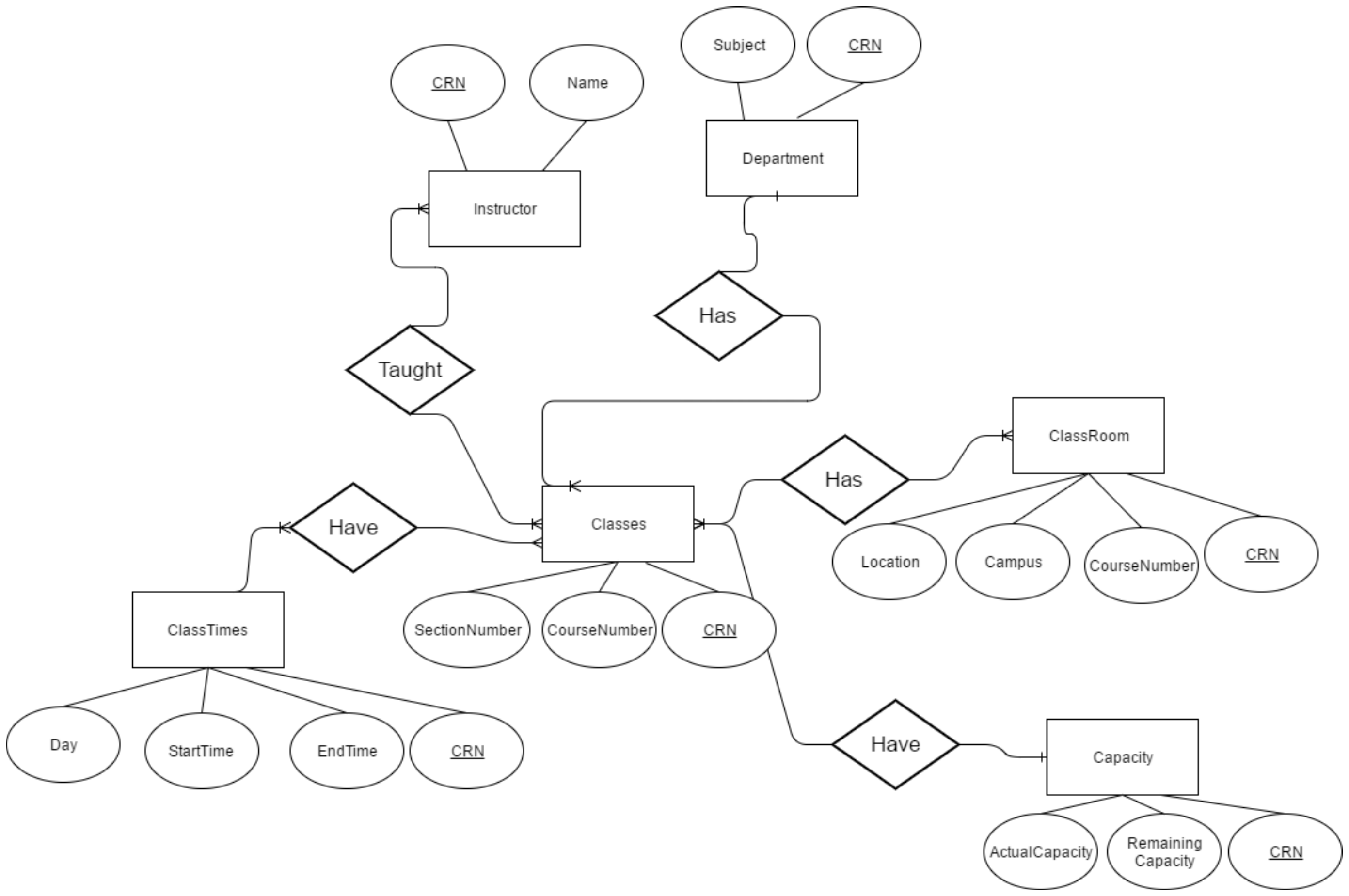
Issues/Risks

Two potential risks that are hard to tackle can be potentially managed given enough time. First if the database must manage multiple user or a high level of requests a multi-threaded design will become necessary. Having each module directly feed into each other requires each one to wait for the next to finish which can lead to speed and throughput issues. The manager stretch goal will handle this but our first iterations of the project will not. Secondly, the current design of storing the database in memory does provide a speed benefit but if a file exceeds the memory for our program this will be problematic. The project can be easily modified to chunk reading and writing from storage but as before our first iterations will not tackle this. Another issue and risk according to the specification document is that queries tend to be more verbose and harder to implement than actual SQL.

Relation Schema



ER Diagram


CRN
Name
Instructor
Subject
CRN
Department
Has
Taught
Has
ClassRoom
Have
Classes
Location
Campus
CourseNumber
CRN
ClassTimes
SectionNumber
CourseNumber
CRN
Day
StartTime
EndTime
CRN
Have
Capacity
ActualCapacity
Remaining Capacity
CRN