Software Design Document

For

Majorizor

Version 1.0 Approved

Prepared by Emily Campbell, Jackson DeMeyers, Joseph Mortefolio, and Lingling Yao

April 14, 2017

Table of Contents

1. Introduction

The purpose of this software design document is to provide a low-level description of the Majorizor system, providing insight into the structure and design of each component. Topics covered include the following:

- Class hierarchies and interactions
- Data flow and design
- Processing narratives
- Design constraints and restrictions
- User interface design
- Test cases and expected results

This document is meant to equip the reader with a complete understanding of the inner working of the Majorizor system.

1.1 Goals and Objectives

Majorizor is a new, independent system that Clarkson University students and advisors can use to schedule undergraduate courses and monitor academic progress.

Currently, Clarkson University uses the PeopleSoft system for class enrollment and changes to majors and minors. The disadvantage of PeopleSoft is that it does not provide students with a complete list of the course requirements they need to graduate; the task of selecting courses and fulfilling academic requirements is left to students and advisors.

Majorizor offers a means for students and advisors to automate the process of progress tracking, and provide tools to make course selection quicker and easier for students and advisors.

1.2 Statement of Scope

The Majorizor system is composed of two primary components. A server-side Microsoft ASP.NET application which allows for user input and interaction, and a server-side MySQL database which stores and synchronizes the required data across all devices accessing the system.

The system features can also be broken up into two sub-groups; core features, which are essential to the operation of Majorizor, and additional features, which are meant to add extra functionality and usability to the application, but are not essential to the release of the system.

1.2.1 Core Features

The following are core features of the Majorizor application:

- Academic major selection

- Academic minor selection
- Automatically generated undergraduate curriculums and anticipated graduation dates

1.2.2 Additional Features

The following are additional features provided by the Majorizor application:

- Curriculum progress tracking based on course completion
- Advisor overview of students' schedules and progress

1.3 Major Constraints

The biggest constraint on the Majorizor project is time. The project is to be planned, designed, implemented, and tested, and fully documented; including both the server-side ASP.NET application, and the MySQL database, over the course of a single semester. As a group, there is limited experience with Bootstrap and C#, so a portion of the time will be spent training the group to learn the basics of the framework and language so development can go smoothly. Additionally, a project of this scale would realistically require more than one semester to fully design and implement all intended features. As such, this time constraint will lead to core features being implemented, and additional, non-essential features, can be added in future releases of the software.

1.4 Intended Audience and Reading Suggestions

This document is intended for individuals involved in the design and development process of the Majorizor system. In our setting, at Clarkson University, these individuals may include developers, project consultants, and University professors. The document is not required to be read sequentially, but instead is intended to allow readers to skip to any section to find information they may find relevant.

Below is a brief overview of each section of the document.

2. Data Design

2.1 Internal Software Data Structure

Majorizor's internal structure is divided between two server-side systems; an ASP.NET application, and a MySQL database.

On the ASP.NET application, data will reside on the application server. The data will be organized and stored in different classes, which will be clearly documented in this document. The ASP.NET WebForms framework is event driven, where actions by the users trigger processes to be run on the server with the provided input. Data generated by the application, and required to be stored long term, will be interfaced and stored in the database.

The database will store the necessary information to feed the ASP.NET application with data. This includes users, user types, course and scheduling information, data required for progress tracking, and many other records. The records are designed to match the structure of the classes in the .NET application, and are interfaced with the application using a set of data layer classes. Both the records in the database, and classes used to interface with the database will be covered in this document.

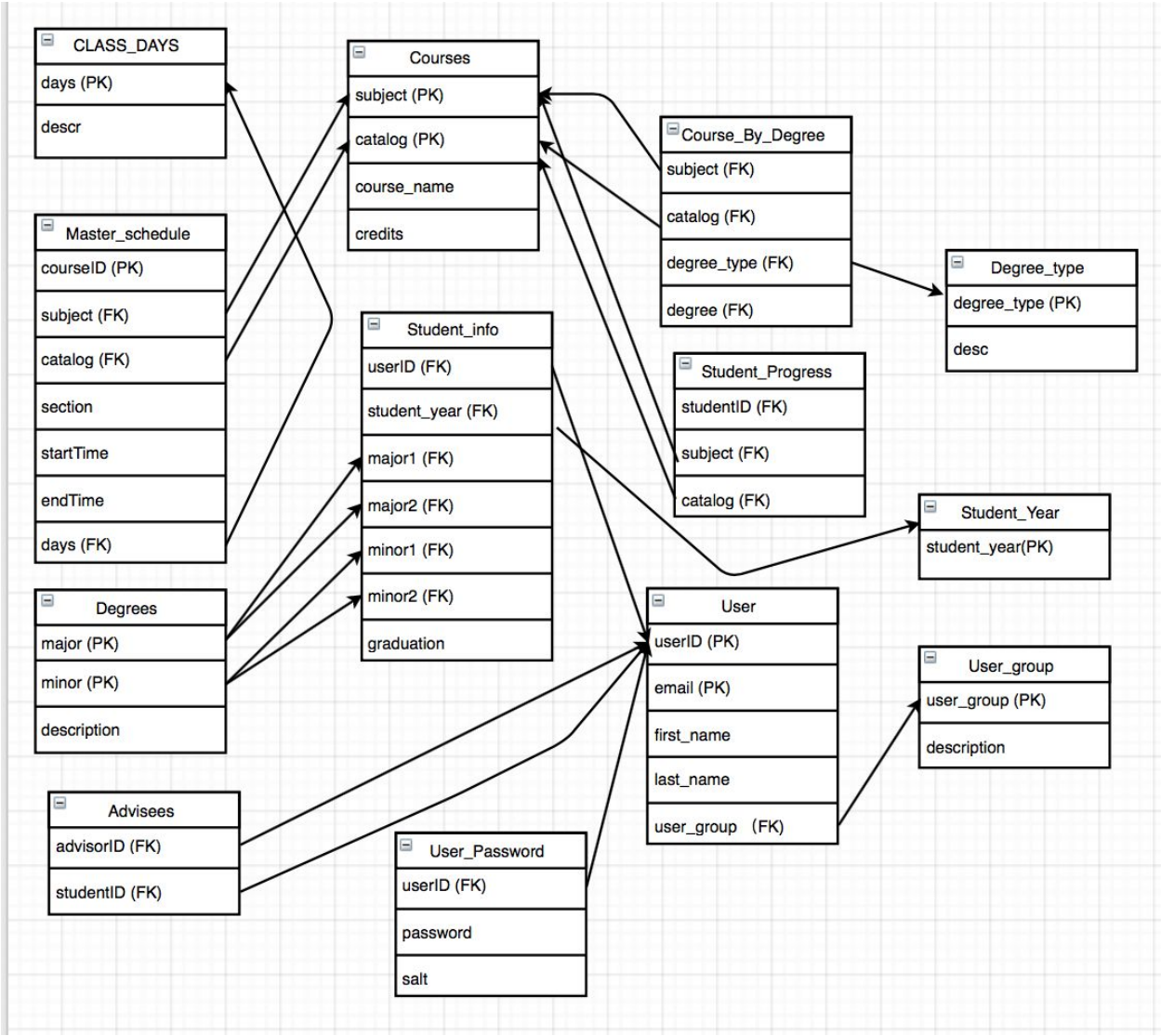
2.2 Global Data Structure

The global data for Majorizor is best described by the database design, as detailed in Section 2.4. The database stores all required information for the Majorizor system to run, and is organized between a variety of different records, using primary and foreign key constraints to link the data together. The .NET application code will talk to the database using data layer classes in the application code to call stored procedures on the database.

2.3 Temporary Data Structure

Temporary data structures, as pertained to Majorizor, refer to data objects which are created and stored on the application server using information acquired from either user input or the data layer code receiving information from the MySQL database. These objects only exist for, at most, the duration of a user's session, while many only exist while processing information either to or from the database. Once data is displayed on the screen and the application server is done processing an event, most objects will be deleted from memory.

2.4 Database Description



Here is the data diagram where all the arrows point from foreign key to primary key. PK refers to primary key and FK refer to foreign key. Using primary key in database allow users to unique the data for the column. Foreign key allows user to put constraint on the data where user cannot allow update data in the foreign key table without update the data in the primary key table.

3. Architectural and Component-Level Design

3.1 System Structure

Majorizor's internal structure is divided between two server-side systems; an ASP.NET application, and a MySQL database.

The ASP.NET application code uses WebForms, written in HTML and Bootstrap using ASP.NET server side controls and an event driven architecture written in C#. Each WebForm has a corresponding code-behind page, where the C# code for the events reside. Users trigger events by pushing buttons, changing selected values in dropdowns, or through other ASP.NET compatible events. The events use data the user input into the server controls rendered on the screen to process, send, and receive information to the backend database.

All code in the ASP.NET application can be categorized into two groups; Resources, and Data Access. The Resources group contains all classes which will be directly accessed by the WebForms, including data objects and methods required to accept user input to objects of different types. The Data Access group contains all classes and code which will communicate to the MySQL database. This design scheme allows us to separate our classes used directly within the WebForms from the classes used to communicate with the database. Each class in the Data Access group provides means to send and receive data from the MySQl database for its corresponding Resources class.

MySQL is an open source relational database management system which allow user to stored information as form of related tables. We use MySQL to store all our schedule, user and security data. We used store procedure to set up the connection with MySQL database with Microsoft Visual Studio. Then when we running our C# language code in Visual Studio, it will pull the information which is stored in the MySQL database and use it when needed. Here are some main tables we developed in MySQL:

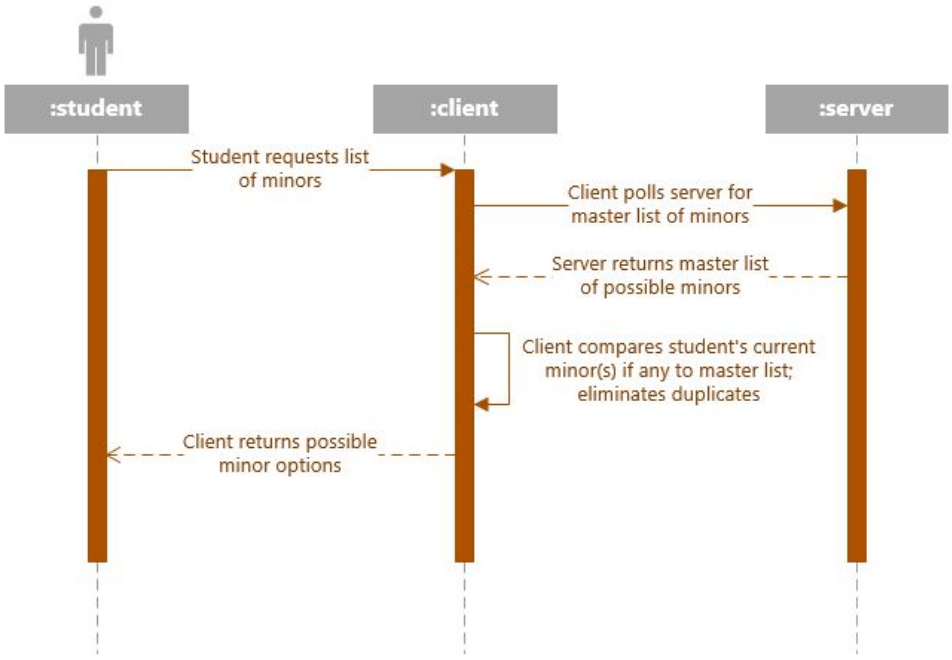
```
CREATE TABLE class_days (
 days char(4) NOT NULL,
 descr char(50) NOT NULL,
 PRIMARY KEY (days)
);
```

```
CREATE TABLE course_by_degree (
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
  degreeType char(3) NOT NULL,
  degree char(2) NOT NULL,
```

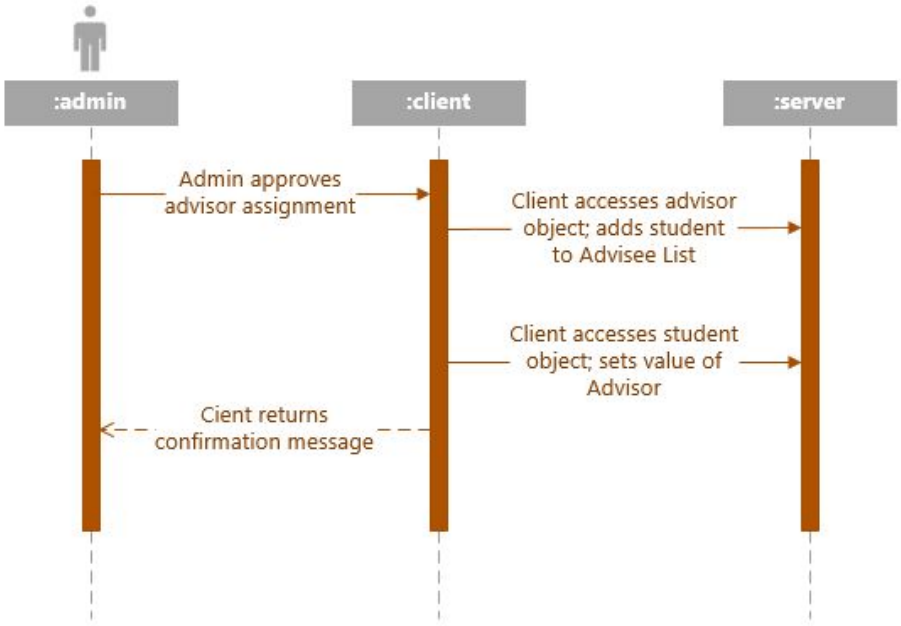
```
  KEY fk_cbd_courses (subject,catalog),
  KEY fk_cbd_degreeType (degreeType),
  KEY fk_cbd_degree (degree),
  CONSTRAINT fk_cbd_courses FOREIGN KEY (subject, catalog)
  REFERENCES courses (subject, catalog),
  CONSTRAINT fk_cbd_degree FOREIGN KEY (degree)
  REFERENCES degrees (major),
  CONSTRAINT fk_cbd_degreeType FOREIGN KEY (degreeType)
  REFERENCES degree_type(degreeType)
);

CREATE TABLE courses (
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
  name char(100) NOT NULL,
  credits int(1) DEFAULT NULL,
  PRIMARY KEY (subject,catalog)
);

CREATE TABLE degrees (
  major char(2) NOT NULL,
  description char(50) DEFAULT NULL,
  PRIMARY KEY (major)
);
CREATE TABLE master_schedule (
  courseID int(8) NOT NULL,
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
  section char(2) NOT NULL,
  startTime time DEFAULT NULL,
  endTime time DEFAULT NULL,
  days char(4) DEFAULT NULL,
  PRIMARY KEY (courseID),
  KEY fk_mas_courses (subject,catalog),
  KEY days (days),
  CONSTRAINT fk_mas_courses FOREIGN KEY (subject, catalog)
  REFERENCES courses (subject, catalog),
  CONSTRAINT master_schedule_ibfk_1 FOREIGN KEY (days)
REFERENCES class_days (days)
);

CREATE TABLE user_group (
  user_group char(10) NOT NULL,
  description char(50) DEFAULT NULL,
  PRIMARY KEY (user_group)
);
```

The numbers in the parentheses represent how much memory space we are using and the character in front of the number shows what types of input we are declare. Each table in the database store their unique values. A complete MySQL can be view in Appendix.

3.2 Sequence Diagrams



The figure is a sequence diagram displaying the interactions between the user and the application when the user opts to select a minor. As soon as the user makes the request, the application requests the master list of minors from the database. Once the list is returned, the Majorizor application cross-references the master list of minors with the list of minors in which the student is already enrolled. The application then returns a list of all majors in which the user is not currently enrolled.



This sequence diagram illustrates the series of events that occur during the “Approve Advisor” action. Pending approvals appear as message boxes on the admin landing page; this sequence begins when the admin presses the “Approve” button. The application contacts the database and requests a change be made to the relevant Advisor object. The name of the student for whom the advisor is being approved is added to the Advisor object’s list of advisees. The application then sends a request to update the Advisor field of the Student object. Once these are complete, the application sends a confirmation message to the admin user to inform them that their changes have successfully gone through.

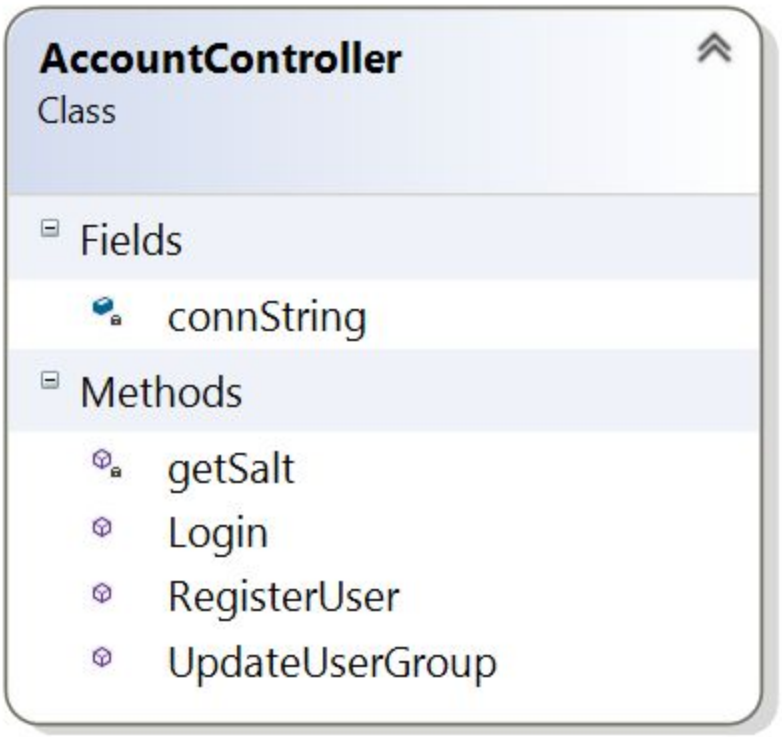
3.3 AccountController Class

The AccountController class is DataAccess group class meant to provide the necessary functions to communicate with the database for any data related to specific user accounts. This includes account registration, login, and updating the user’s user group.

3.3.1 Processing Narrative

When a user uses the Login screen to either login or register, the AccountController class’s static methods are called to communicate with the MySQL database. Additionally, if an admin uses the User Management panel on the Admin Landing screen, an AccountController class static method is called from within the UserGroup class to change the user group in the database.

3.3.2 Class Diagram



3.3.3 Design Class Hierarchy

The AccountController class has no parent or child classes. However, the Login method returns a UserGroup type, an enumeration defined in the UserGroup class.

3.3.4 Restrictions/Limitations

The getSalt method is used only by the Login method to re-hash the user's password with the correct salt from the database. Since it is only used from within the Login method, getSalt should be a private method.

3.3.5 Performance Issues

Since this class is responsible for communicating with the database, there are potential performance issues based upon the status of the database. If the client or application server cannot communicate with the database the methods in this class will not function properly.

3.3.6 Design Constraints

The getSalt method should not be accessed by any other classes. As such, it should be a private method.

3.3.7 Processing Detail for Each Operation

- Private getSalt(email : String) : String
    - When called from within the Login method, the getSalt method will return the salt stored in the user_password table of the database that matches the given email. This is used to re-hash and check the password upon an attempted login.
- Login(email : String, password : String) : UserGroup
    - The Login method takes an email and password string, and attempts to login to the application. It calls getSalt, which returns the salt in the database which matches the provided email, re-hashes the the password, and checks it with the password in the database. If the passwords match, the user's user group is returned. If the passwords did not match, a default value is return to tell the calling code that login was unsuccessful. This method uses the Login stored procedure on the database.
- RegisterUser(firstname : String, lastname : String, email : String, password : String, salt : String) : void
    - The RegisterUser method takes all given parameters, and attempts to add the new user to the user and user_password tables in the database. The password is salted and hashed using the Security class, and then the database attempts to insert the new user. If the insert is successful, the user will be logged in and redirected. If any key constraints in the database fail, the user is not registered. The Register stored procedure is used for this method.
- UpdateUserGroup(userID : int, userGroup: UserGroup) : void
    - The UpdateUserGroup method is called from within the UserGroup class. When called, it will use the UpdateUserGroup stored procedure to update the given user, by userID, to the user group provided.

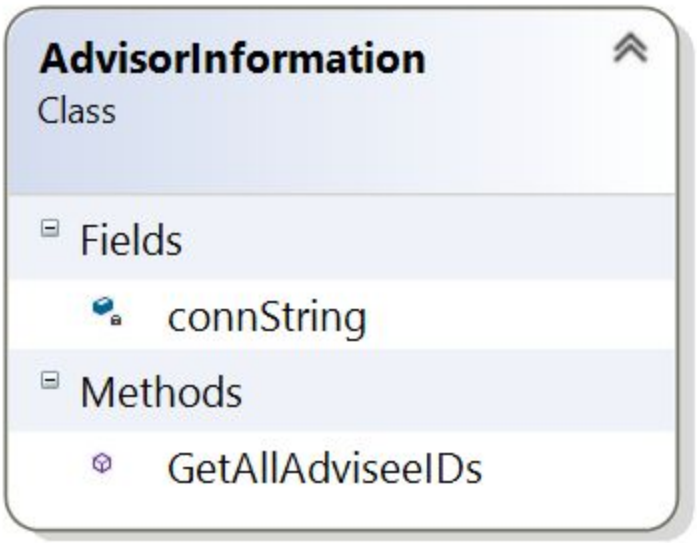
3.4 AdvisorInformation Class

The AdvisorInformation class is a DataAccess group class meant to provide the necessary functions to communicate with the database for any data related to the Advisor class.

3.4.1 Processing Narrative

When the Advisor class initializes a new Advisor object using the Advisor(userName : string) constructor, the AdvisorInformation class's GetAllAdviseeIDs method is called to fill the list of advisees. This is currently the only use of this class.

3.4.2 Class Diagram



3.4.3 Design Class Hierarchy

The AdvisorInformation class has no child or parent classes.

3.4.4 Restrictions/Limitations

There are no restrictions or limitations on the AdvisorInformation class.

3.4.5 Performance Issues

Since this class is responsible for communicating with the database, there are potential performance issues based upon the status of the database. If the client or application server cannot communicate with the database the methods in this class will not function properly.

3.4.6 Design Constraints

There are no major design constraints for this class.

3.4.7 Processing Detail for Each Operation

- GetAllAdviseeIDs(string userName) : List<int>
    - This method takes an advisor username and uses the GetAllAdviseeIDs stored procedure from the database to return a list of all advisee userIDs which correspond to the given advisor. This method is used to fill the member variable AdviseeIDs list in the Advisor class.

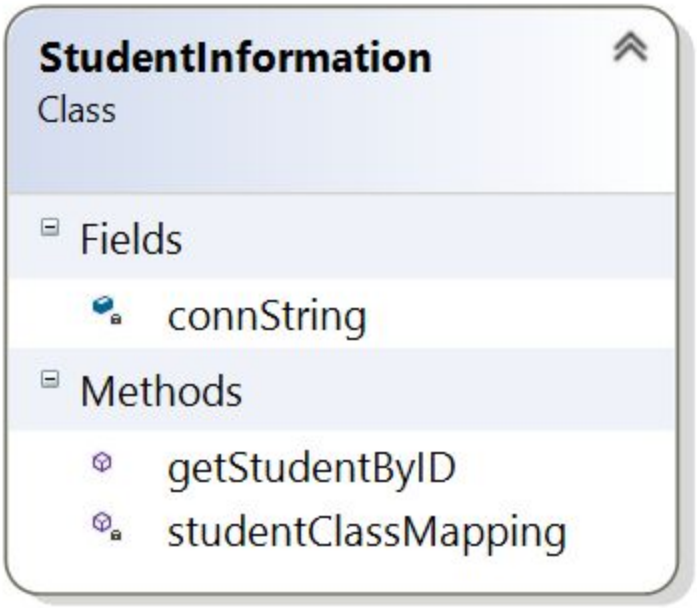
3.5 StudentInformation Class

The StudentInformation class is DataAccess group class meant to provide the necessary functions to communicate with the database for any data related to the Student class.

3.5.1 Processing Narrative

When the Student class initializes a new student using the constructor with an ID as parameter, it class the getStudentByID method of the StudentInformation class to get all information needed to fill a Student object from the database. The getStudentByID method calls the private StudentClassMapping method, which is responsible for setting up the Student object based on the values returned from the database.

3.5.2 Class Diagram



3.5.3 Design Class Hierarchy

The StudentInformation class has no child or parent classes. However, both the class's methods return a Student type from the Student class.

3.5.4 Restrictions/Limitations

The studentClassMapping method is used only by the getStudentByID method to correctly match all returned values from the database into a Student object. Since it is only used from within the getStudentByID method, studentClassMapping should be a private method.

3.5.5 Performance Issues

Since this class is responsible for communicating with the database, there are potential performance issues based upon the status of the database. If the client or application server cannot communicate with the database the methods in this class will not function properly.

3.5.6 Design Constraints

The studentClassMapping method should be used only by the StudentInformation class. As such, it is a private method.

3.5.7 Processing Detail for Each Operation

- getStudentByID (userID : int) : Student
    - This method returns a Student object, filled with all appropriate data from the database for the user, given the userID. This method uses the GetStudentByID stored procedure, and uses the studentClassMapping method to map the data to the object correctly.
- studentClassMapping (_dr : DataRow) : Student
    - This method returns a Student given a DataRow. The method takes the row, parses the correct columns from the row into the correct variables of a Student object, and then returns the completed Student object. This is a private method, since it is called from within GetStudentByID.

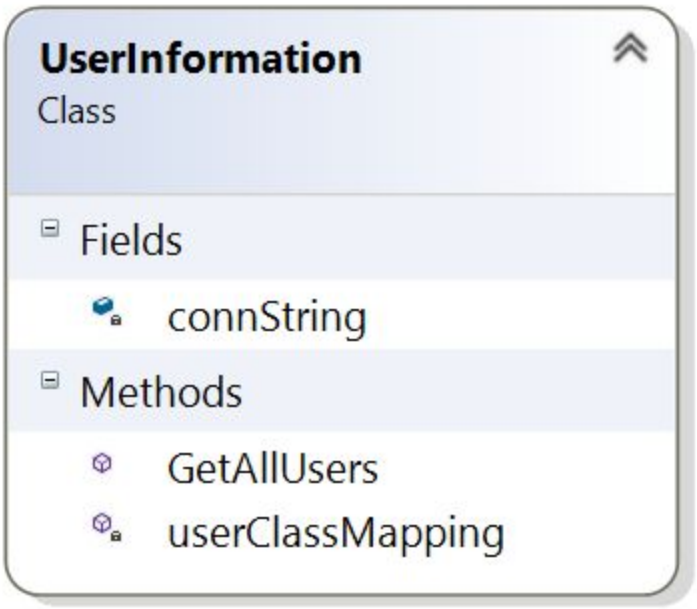
3.6 UserInformation Class

The UserInformation class is DataAccess group class meant to provide the necessary functions to communicate with the database for any data related to the User class.

3.6.1 Processing Narrative

When the User class class the GetAllUsers method to return a list of all the users in the system, the GetAllUsers method in the UserInformation class is called. This method uses the GetAllUsers stored procedure to fill a Dataset with the rows returned from the stored procedure. Then, for each row in the dataset the userClassMapping method is called to map each row into a User object. Then, a list of User objects is returned to the calling code.

3.6.2 Class Diagram



3.6.3 Design Class Hierarchy

The UserInromation class has no child or parent classes. However, the methods return User types from the User class.

3.6.4 Restrictions/Limitations

The userClassMapping method is used only by the getAllUsers method to correctly match all returned values from the database into a User object. Since it is only used from within the GetAllUsers method, useClassMapping should be a private method.

3.6.5 Performance Issues

Since this class is responsible for communicating with the database, there are potential performance issues based upon the status of the database. If the client or application server cannot communicate with the database the methods in this class will not function properly.

3.6.6 Design Constraints

The userClassMapping method should be used only by the UserInformation class. As such, it is a private method.

3.6.7 Processing Detail for Each Operation

- GetAllUsers( ) : List<User>
    - This method returns a list of User objects, all filled with all appropriate data from the database for the each user. This method uses the GetAllUsers stored procedure, and uses the userClassMapping method to map the data to the object correctly.
- userClassMapping (_dr : DataRow) : User
    - This method returns a User given a DataRow. The method takes the row, parses the correct columns from the row into the correct variables of a User object, and then returns the completed User object. This is a private method, since it is called from within GetAllUsers.

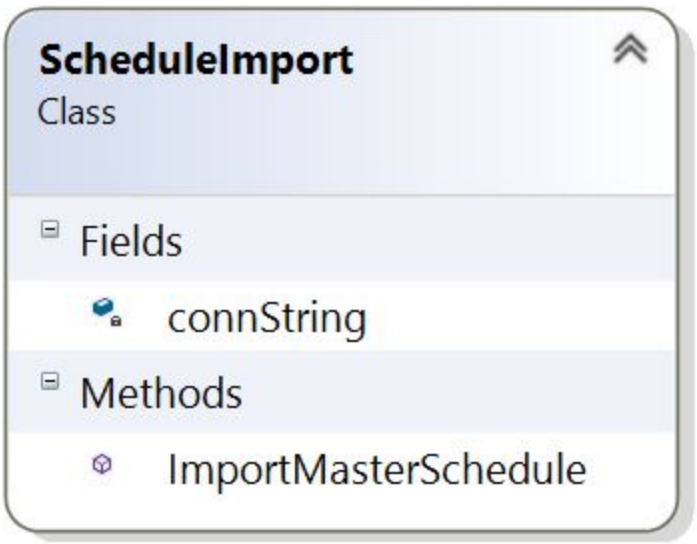
3.7 ScheduleImport Class

The ScheduleImport class is DataAccess group class meant to provide the necessary functions to communicate with the database for any data related to the MasterScheduleLoader class.

3.7.1 Processing Narrative

When the MasterScheduleLoader class calls the LoadSchedule method, it invokes a call to ImportMasterScheudle method of the ScheduleImportClass. It takes a list of Course objects as input, and inserts each course from the schedule into the database using a SQL transaction and stored procedures.

3.7.2 Class Diagram



3.7.3 Design Class Hierarchy

The ScheduleImport class has no child or parent classes. However, it does use Course objects from the Course class as a parameter to the method.

3.7.4 Restrictions/Limitations

There are no restrictions or limitations on the ScheduleImport class.

3.7.5 Performance Issues

Since this class is responsible for communicating with the database, there are potential performance issues based upon the status of the database. If the client or application server cannot communicate with the database the methods in this class will not function properly.

3.7.6 Design Constraints

If, while inserting a course into the database, an error occurs, causing a MySqlException to be throw, then the rows inserted into the database during the same function call should not be inserted to avoid partially uploaded schedules. As such, a SQL transaction is used, and only commits the changes if no exceptions are thrown for each insert. Otherwise, the changes are rolled back.

3.7.7 Processing Detail for Each Operation

- ImportMasterSchedule (schedule: List<Course>) : void
    - This method is used to insert all the courses given in a list of Course objects into the database. The method creates a SQL transaction, and for each course in the list inserts the row using the UpoadMasterScheudle stored procedure. If every insert is successful, then the transaction should commit the changes, otherwise the transaction should rollback the changes and throw an exception.

3.8 Course Class

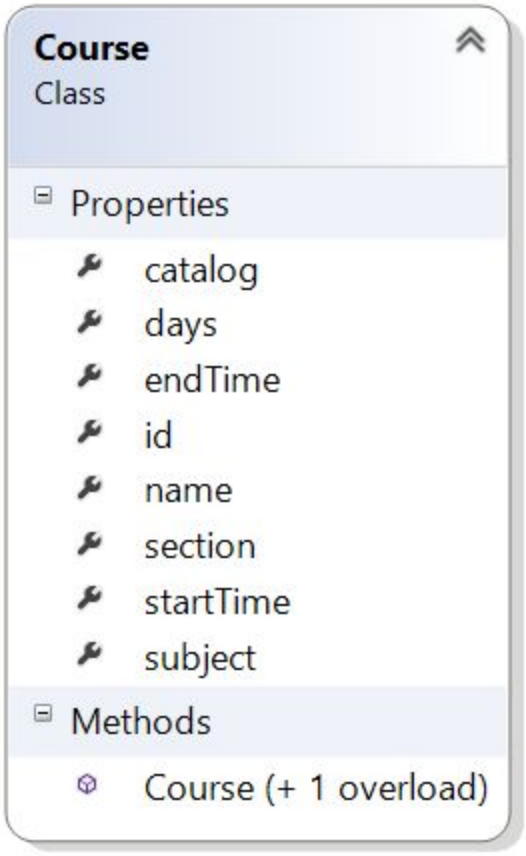
The Course class is a created data type used to represent the information about a specific course. This information includes class number, subject and other relevant information covered below. Course objects are meant to organize the information of a course to assist in schedule generation.

3.8.1 Processing Narrative

Majorizor uses Course objects to store all of the classes used in the generation of schedules. Each Course object includes the following information:

- Course Number
- Subject
- Catalog Number
- Section Number
- Course Name
- Start time
- End time
- Meeting Days

3.8.2 Class Diagram



3.8.3 Design Class Hierarchy

The Course class has no parent or child classes. However this data type is used to assist in the parsing and loading of the master schedule as well as the generation of a schedule.

3.8.4 Restrictions/Limitations

Since the Course class is self-contained, there are no practical restrictions.

3.8.5 Performance Issues

Once again since the Course class is self contained and only used as a means of organizing and storing data, there will be no performance related issues with this class.

3.8.6 Design Constraints

The only major design constraint with this class is the large number of components required in its main constructor. That is, in order to make a new Course variable, one must have valid entry for each of the 8 components.

3.8.7 Processing Detail

The Course class is purely used for data storage. There are no algorithms associated with this class.

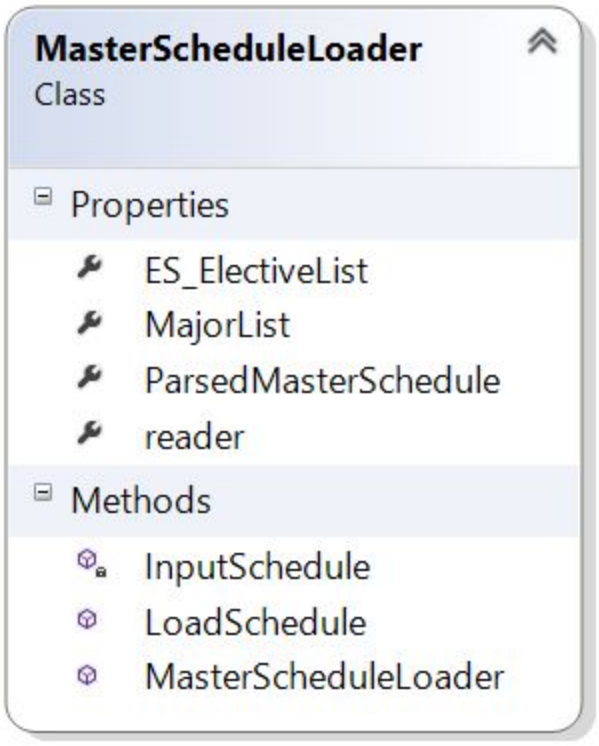
3.9 MasterScheduleLoader Class

The purpose of this class is to take the master schedule from peoplesoft as a csv file, then isolate and store only the courses that contribute to majors and minors relevant to the majorizor.

3.9.1 Processing Narrative

When the MasterScheduleLoader class is initialized it opens up a stream to the csv file. That stream is then parsed one line at a time for relevant courses which are put into a list of course objects. That list of course objects is then sent to the ScheduleImport class to update the database.

3.9.2 Class Diagram



3.9.3 Design Class Hierarchy

The MasterScheduleLoader class has no parent or child classes. However it is heavily reliant on the Course class. It is also associated with the ScheduleImport class.

3.9.4 Restrictions/Limitations

There are currently no practical limitations to this class.

3.9.5 Performance Issues

The MasterScheduleLoader uses minimal resources and will put little strain on a user's hardware.

3.9.6 Design Constraints

The MasterScheduleLoader has one major design constraint, and that is that the csv file must use ‘|’ as a delimiter meaning that in order successfully upload the master schedule the user must change their computer's settings such that ‘|’ is the generated delimiter for csv files rather

than ‘,’. This is because there are a number of courses in the master schedule that include at least one ‘,’ in their course title which leads to a desync if the parsing is done using ‘,’.

3.9.7 Processing Detail For Each Operation

The MasterScheduleLoader(Stream scheduleStream) component has two key functions. The first function is to open a new data stream from the MasterSchdeule csv file. The second function is to declare and initialize a list of Course objects that will be used to store the parsed courses from the MasterSchedule csv.

The InputSchedule() function is the component that parses the data from the csv file and converts it to a list of course objects. The function does this by going through each line of the csv (each line containing the information for one course) and splits the string using ‘|’ as its delimiter. It then stores the split string as an array of smaller strings with each element in the array representing a different piece of information about the course (i.e. Course Number, Subject,...) It then checks to see if the course is one of the required courses for one of the specified majors and minors and if it is, the courses information is then converted from a set of strings to the correct data types required to be defined as a course object. A new course object is then created whose parameters are the information of the course, and this new course is then added to the list of course objects.

The LoadSchedule() function has only one responsibility, and that is to take the list of courses generated by the InputSchedule() function and pass it to the ScheduleImport class which will then alter the database accordingly.

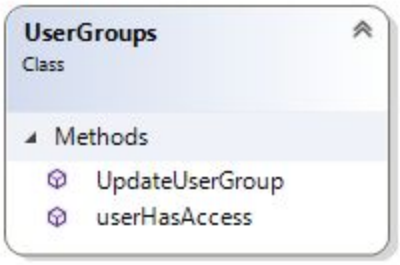
3.10 UserGroup Class

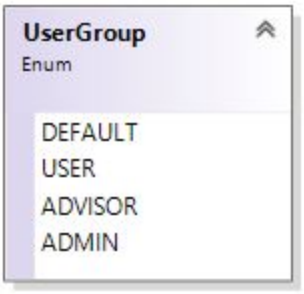
The purpose of the UserGroup class is to determine if the user is allowed to

3.12.1 Processing Narrative

When a UserGroup object is initialized, it can have one of four possible values: DEFAULT, USER, ADVISOR, and ADMIN. These values are derived from the enumerated type UserGroup. When a user requests a page or set of data, a UserGroup object is created and compared the UserGroup object associated with that request.

3.12.2 Class Diagram





3.12.3 Design Class Hierarchy

The UserGroup class has no parent or child classes.

3.12.4 Restrictions/Limitations

The value of UserGroup must be DEFAULT, USER, ADMIN, or ADVISOR.

3.12.5 Performance Issues

UserGroup objects require negligible resources, are are in most cases deleted immediately after use. Therefore, it is not liable to cause any performance issues.

3.12.6 Processing Detail For Each Operation

- UpdateUserGroup(int UserID, UserGroup userGroup) - sets the userGroup of the User whose ID number is provided to the value of the userGroup object
- userHasAccess(UserGroup check, UserGroup userGroup) - checks for a match between the values of check and userGroup

3.11 User Class

The purpose of the User class is to store the relevant information about a generic user for use by other functions.

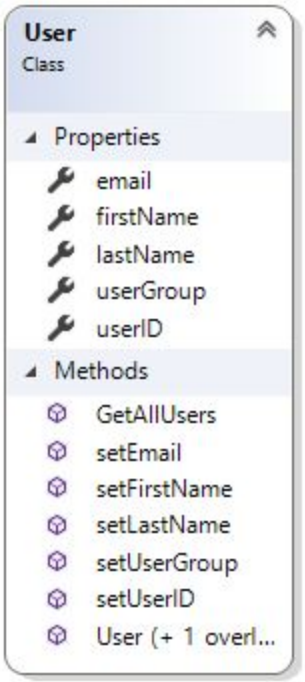
3.11.1 Processing Narrative

The User class stores the following information about the student:

- User ID
- Firstname
- Lastname
- Email
- User type

Additionally, the class has functions for modifying each of the above attributes. It also has a function to get all users stored in the Majorizor database.

3.11.2 Class Diagram



3.11.3 Design Class Hierarchy

The User class has no parent or child classes.

3.11.4 Restrictions/Limitations

The value of the email variable should be a valid email address.

3.11.5 Performance Issues

The User class does not use a significant amount of resources, and therefore is not a performance concern.

3.11.6 Processing Detail For Each Operation

- GetAllUsers() - returns a list of all users in the Majorizor database
- setUserID(int _userID) - changes the value of the userID variable to that of the _userID variable
- setFirstName(string _firstName) - changes the value of the firstName variable to that of the _firstName variable
- setLastName(string _lastName) - changes the value of the lastName variable to that of the _lastName variable
- setEmail(string _email) - changes the value of the email variable to that of the _email variable
- setUserGroup(UserGroup _userGroup) - changes the value of the userGroup variable to that of the _userGroup variable

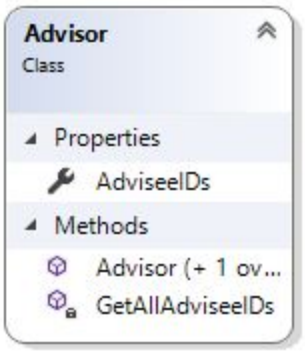
3.12 Advisor Class

The purpose of the Advisor class is to store the relevant information about an advisor for use by other functions.

3.12.1 Processing Narrative

The Advisor class contains a list of advisees. It also contains methods for retrieving all Advisee IDs from an AdvisorInformation object.

3.12.2 Class Diagram



3.12.3 Design Class Hierarchy

The Advisor class has no parent or child classes.

3.12.4 Restrictions/Limitations

The list of Advisee IDs should only contain valid IDs.

3.12.5 Performance Issues

The Advisor class does not use a significant amount of resources, and therefore is not a performance concern.

3.12.6 Processing Detail for Each Operation

- GetAllAdviseeIDs(string advisorEmail) - retrieves all Advisee IDs from the AdvisorInformation object whose email was specified.

3.13 Student Class

The purpose of the Student class is to store the relevant information about a student for use by other functions.

3.13.1 Processing Narrative

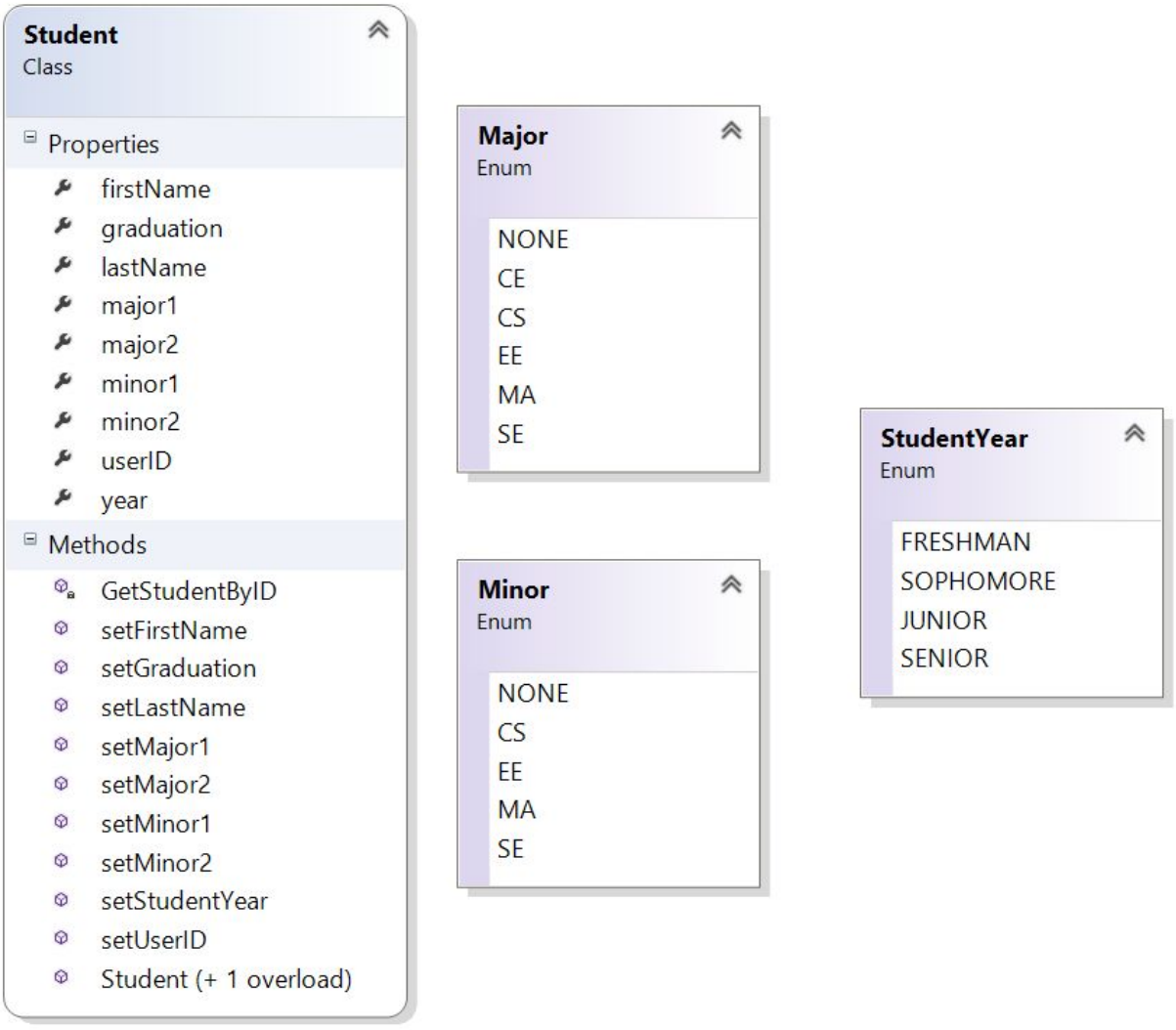
The Student class stores the following information about the student:

- User ID
- Firstname

- Lastname
- Year
- Major 1
- Major 2
- Minor 1
- Minor 2
- Graduation date

Additionally, the class has the ability to get a student's information from their user ID.

3.13.2 Class Diagram



3.13.3 Design Class Hierarchy

The Student class has no parent or child classes. However, the methods of other classes return values of type Student.

3.13.4 Restrictions/Limitations

There are currently no limitations with regard to the uses of this class

3.13.5 Performance Issues

The Student class does not use a significant amount of resources, and therefore is not a performance concern.

3.13.6 Processing Detail For Each Operation

- setUserID(ID : int) - sets the student objects userID to the specified int
- setFirstName(fN : string) - sets the student objects firstname to the specified string
- setLastName(lN : string) - sets the student objects lastname to the specified string
- setStudentYear(sY: StudentYear) - sets the student objects year to the specified StudentYear
- setMajor1(m : Major) - sets the student objects major1 to the specified Major
- setMajor2(m : Major) - sets the student objects major2 to the specified Major
- setMinor1(m : Minor) - sets the student objects minor1 to the specified Minor
- setMinor2(m : Minor) -sets the student objects minor2 to the specified Minor
- setGraduation(g : string) -sets the student objects graduation to the specified string

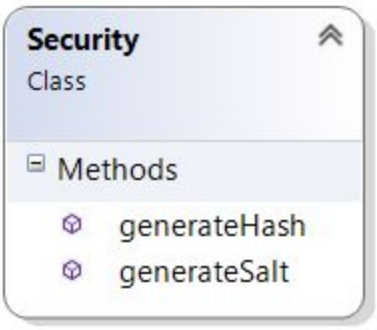
3.14 Security Class

3.14.1 Processing Narrative

The security class store the following information from users:

- User name
- Password

3.14.2 Class Diagram



3.14.3 Design Class Hierarchy

The Security class has no parent or child classes.

3.14.4 Restrictions/Limitations

The user name should be valid email addresses.

3.14.5 Performance Issues

There is no performance issue in this stage.

3.14.7 Processing Detail For Each Operation

- generateHash(Hash: String) - generated a cryptographic random number
- generateSalt(Salt: String) - generated when new account are register or when user change their password

4. User Interface Design

The user interface consists of a main navigation bar, which shows only options related to the user group of the currently logged in user. This navigation bar will allow users to flow between different screens. The screens are organized by user group and feature, and as such, essentially each feature has it's own screen. Relevant screens to the current screen should be easily accessible to the user without using the navigation bar, allowing for a natural flow of the application.

4.1 Description of User Interface

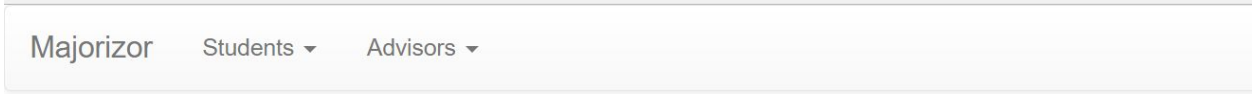
4.1.1 Navigation Bar

Our applications uses a .NET master page to display the same navbar on each page. The navbar code is designed within the master page, and uses the master page code-behind to control the server-side controls within the navbar.

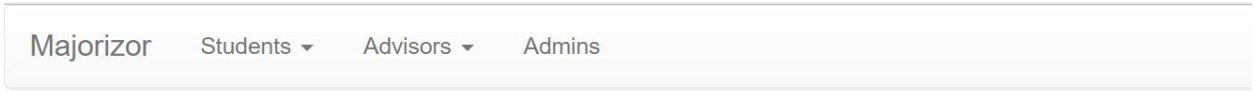
The navbar is displayed differently to each user group. The following list will describe the functionality of each group:

- USER

  - While logged in as a normal application user (In the case of this application, a normal user is a student registered within the application.) the navbar will only show user which navbar options are available to normal users (normal users include the set of students that use the application that are not also advisors or admins).
- ADVISOR



  - While logged in as an Advisor, the user will see the normal student "User" class options as well as the "Advisor" options from the default navbar. This allows advisors to access the same information as the normal users. The information the advisor inputs must match the data in the database, otherwise an exception is thrown.
- ADMIN



    - When the user group is detected as an ADMIN, all other user groups are available to the user. The should allow the Admin user to alter the production database, with their overall knowledge of the application, and maintain all user database and setting within the applications.
    - This user group is only intended for experienced users of the system, as extensive knowledge of the systems involved is limited to a few involved administrators. User's without full application knowledge should not be granted status above Advisor.
- DEFAULT

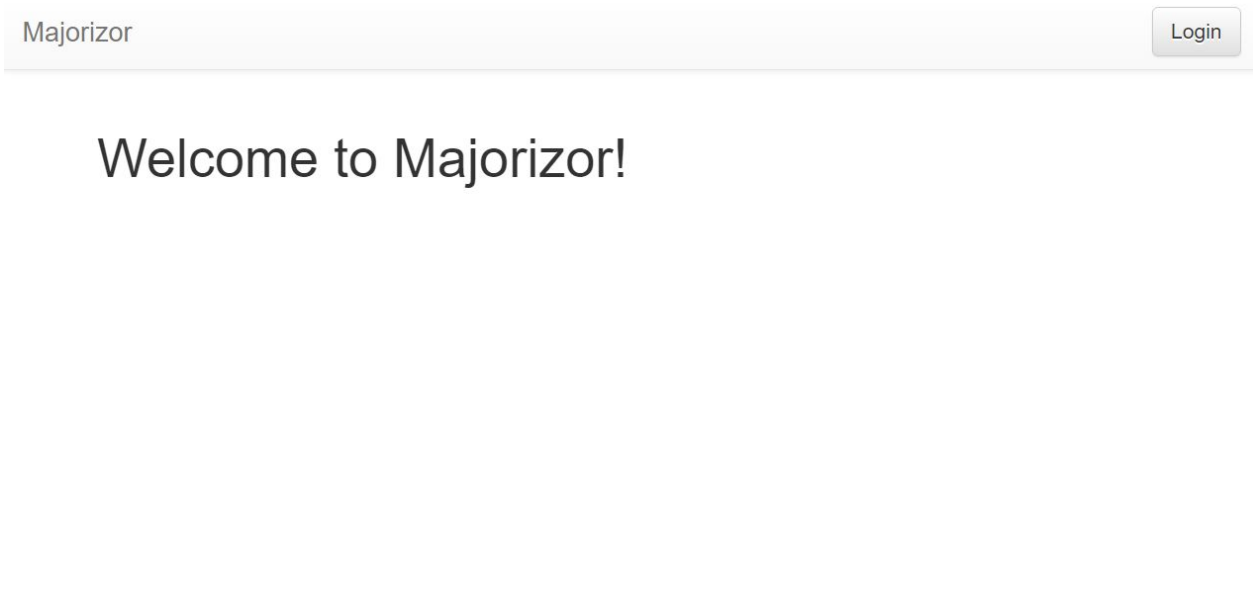
Majorizor

    - This user group exists for the application to understand when a Login request, or other features which require the UserGroup class to be called, to handle all users which return as a non-user type. This means, if a user attempts to login, and the Login method of the AccountController class returns a DEFAULT user group, that the login was unsuccessful, and should report that information to the user. As such, the nav bar should not show any available pages to the current user (which is most likely null).
    - The exception code for the instances mentioned above have not yet been implemented.

The nav-bar also includes the login/registration and logout buttons. The login and registration buttons allow for existing users to sign in, or new users to sign up with the system. The logout button clears the application's server side data, allowing any current users to be released from the system, and allowing a new user to login to that specific session.

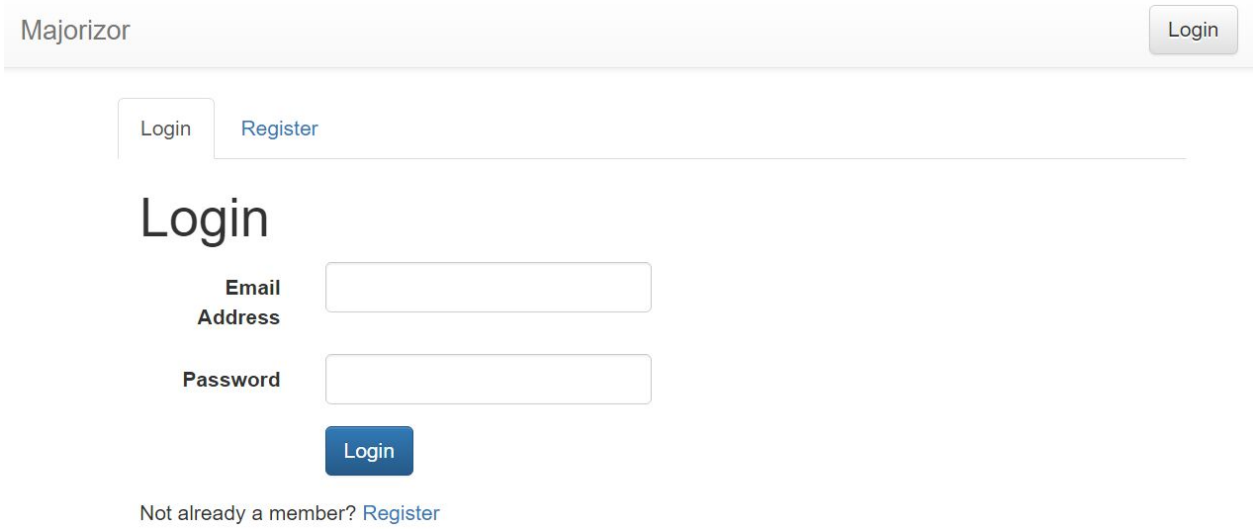
4.1.2 Screens

Main Landing Page



- This page is the default landing page
- A user must login via the button to the top right in order to access any of the majorizors functions.

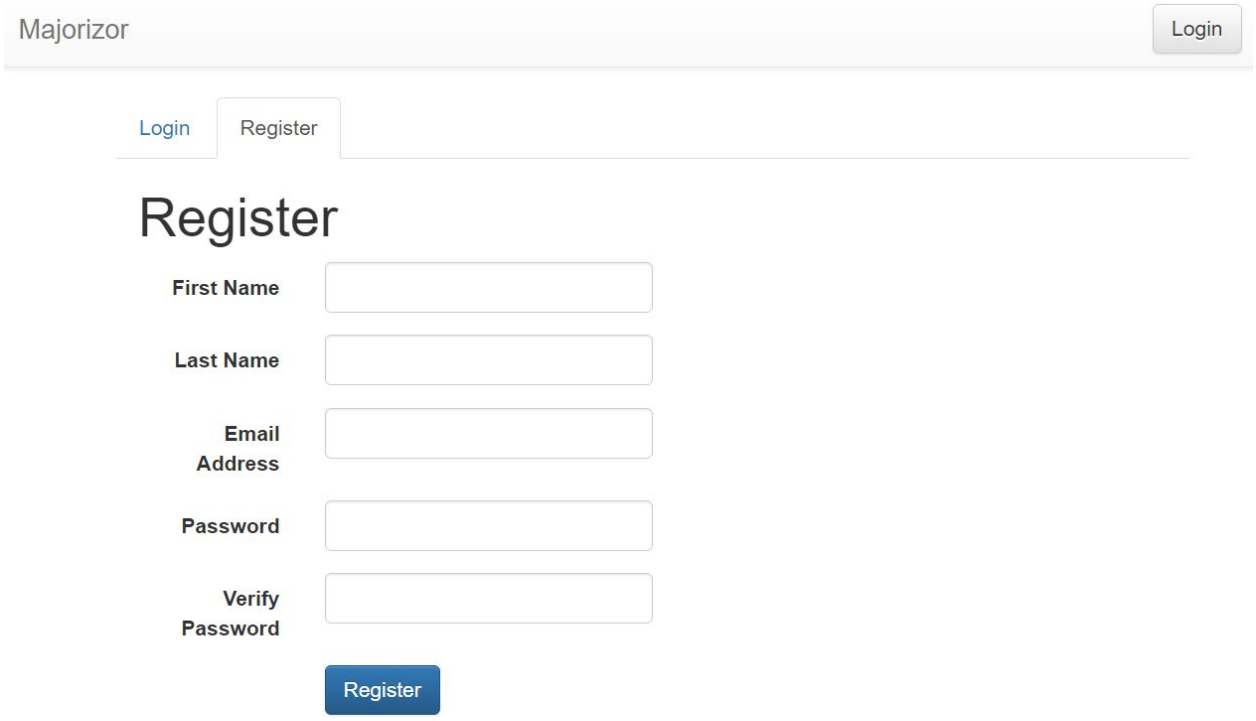
User Login Page



- This page is where all users that have already registered will go to log into the majorizor.
- To login a user must enter the email address associated with their account as well as their password, then press the login button

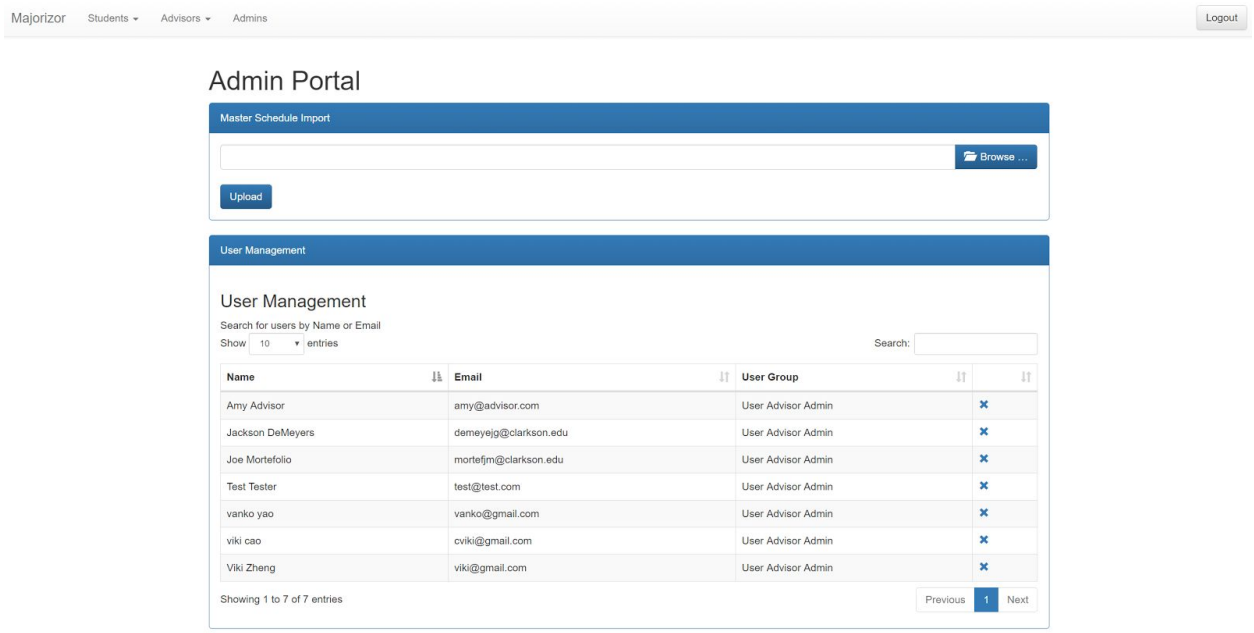
- After the user hits the login button it the system compares the password entered to the password associated with the email entered
- From here a user can also navigate to the registration page from the login page

User Registration Page



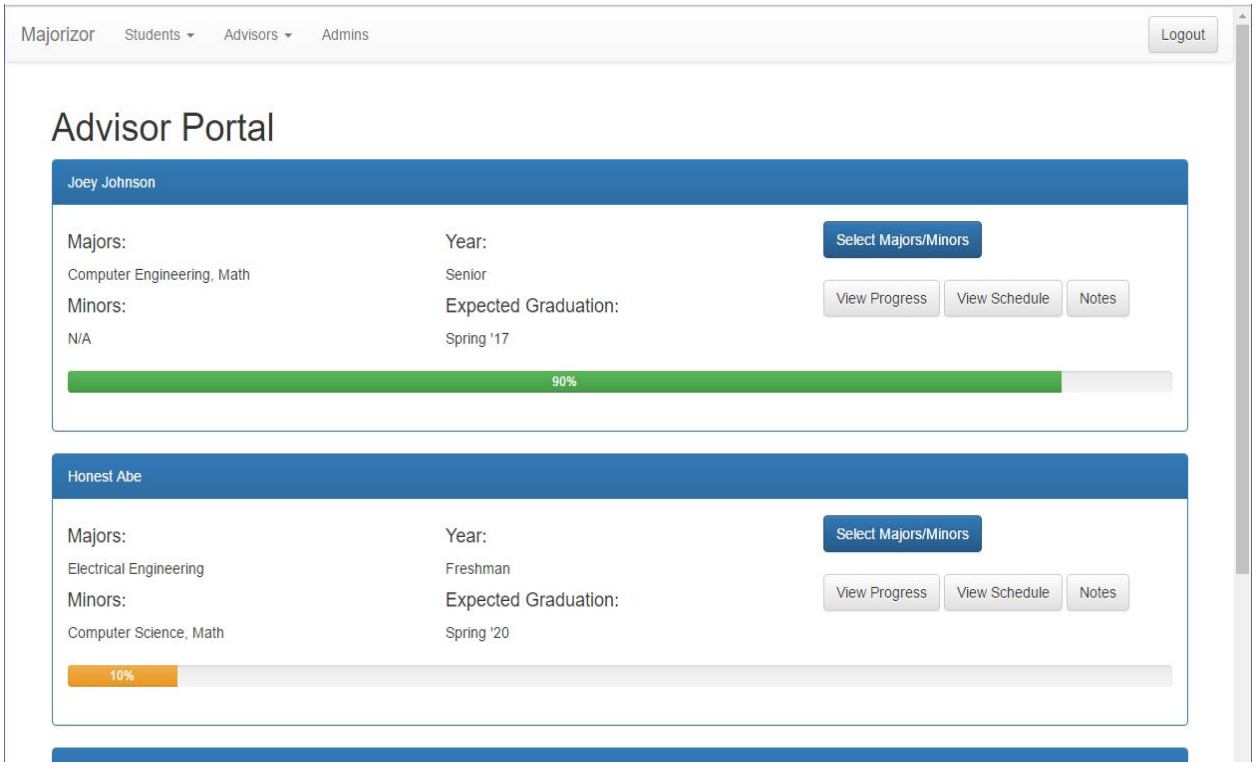
- This page allows a new users to create a majorizor account
- The user must enter their information in required fields and then hit the register button
- From there the user's information will be stored in the database and they will be defaulted to the student user group

Admin Landing Page



- This page contains all of interfaces needed for administrators to complete their core responsibilities
- The first block is the master schedule import interface
- When an admin presses the browse button their file browser will open and they will be able to select a csv file to be parsed and imported.
- When the admin presses the upload button, the file will be parsed and the majorizor will update the master schedule in the database.
- The second block contains a list of all registered users including their name, email, and the user groups that they are members of.
- From this block an admin can alter what user groups a user is a part of as well as search for specific users by name or email

Advisor Overview Page



- This page contains an overview of all of an advisors students and displays important information about each in a clear and organized manner.
- Each student's information is displayed in its own block containing their majors, minors year and expected graduation date. Each block also contains several buttons whose functions are listed below
- The select majors/minors button allows the user to select and add new majors and minors
- The view progress button shows the user a detailed description of the students progress toward their current majors and minors
- The view schedule button allows the user to see the student's current schedule.
- The notes button allows the user to make notes about the student for future reference.

* Please note that the Student Landing page, Major and Minor selection pages, and schedule generation functions are still in development and do not currently have interfaces created.

4.2 Interface Design Rules

Offer Clear, Immediate, and Informative feedback Feedback - Any action performed by a user should result in an immediate response from the system. The response should reflect the importance of the action. That is a substantial action will result in a more substantial response.

Maintain Consistency Across All Users- Make the interface for the different users similar to ensure they are better able to communicate.

Ease of Access- Make the most frequently performed actions the most convenient to perform for the user.

Error Handling- Any fatal errors should be caught before they reach the user, and the user should be informed of any errors that occur.

4.3 Components Available

4.3.1 ASP.NET Components

ASP.NET provides a large collection of useful user interface components. The components used in Majorizor include:

- Master Page
    - Allow you to create a consistent layout for the pages in an application. A single master page will define the standard behavior intended for every screen it is used on. Master pages can contain other .NET controls, and has it's own code-behind page.
    - Majorizor uses a single master page to create the standard layout for the application. The master page includes all frameworks, scripts, and sets up controls which should appear on each page of the application (ie - navbar).
- Content
    - Holds text, markup, and server controls to render to a ContentPlaceHolder control in a master page.
    - Majorizor will use .NET Content sections to hold the HTML and server controls necessary for each screen. These content sections will be rendered to ContentPlaceHolders on the master page.
- ContentPlaceHolder
    - Defines a region for content in an ASP.NET master page.
    - Majorizor uses these placeholders on the master page to render the html and controls for each individual screen. This allows the files for each screen to contain only the HTML and server controls required for that specific screen.
- PlaceHolder
    - Stores dynamically added server controls on the Web page.
    - Majorizor uses placeholders to dynamically create HTML and controls for the screens where the format of the page is based upon the number of results returned from the database.
        - Example - Advisor Landing Page - Each student needs the HTML for a panel built, and filled in with their information. This can be accomplished with a PlaceHolder control and server-side application code.

- Button
    - Displays a push button control on the Web page. Allows for onclick events.
    - Majorizor will use buttons to fire events on the application server.
- DropDownList
    - Represents a control that allows the user to select a single item from a drop-down list.
    - Majorizor will use drop-down lists to allow users to select a single value from a specified group of values. The drop-down lists will fire events when the selected value is changed. The value selected can also be used in other events, fired by buttons.

4.3.2 Bootstrap Components

In addition to the ASP.NET framework components, Majorizor uses the Bootstrap framework, which provides an extensive HTML, CSS, and Javascript library to develop clean and responsive front-ends. Bootstrap will allow us to skin many default HTML elements into beautiful UI elements. Below is a list of the major Bootstrap components used in Majorizor's front-end.

- Nav
    - A wrapper that positions branding, navigation, and other elements in a concise header.
    - Majorizor uses the Bootstrap navbar component to create the main navbar in the application. It allows for easily creating dropdowns and other navigation functionality.
    - Majorizor also uses the nav component to create tabbed views, as seen on the Login/Registration screen.
- Button
    - Bootstrap provides a set of buttons, each with a different style. This allows you to differentiate between different button controls on screens.
    - Majorizor uses bootstrap buttons to style all buttons on the website consistently.
- Panel
    - Creates a container component on the webpage. Allows you to divide content on the webpage between different panels with different headings.
    - Majorizor uses panels to divide sections of each screen. For example, the Admin Landing Page has a panel for each feature available.
- Progress
    - Progress bars provide a means to visually display the progress of a task with a simple and flexible component.
    - Majorizor uses progress bars to dynamically display the progress of a student through their course work. They will provide a visual representation of how close a student is to completing their majors and minors.

4.4 Plugins

4.4.1 Bootstrap File Input

To keep our application's UI elements themed with a common design, a file-picker other than the default .NET file-picker needed to be used. Thus, we are using the "bootstrap-fileinput" plugin developed by Krajee to provide a similarly themed filepicker UI for the application.

Information about the plugin can be found here: http://plugins.krajee.com/file-input/demo

4.4.2 DataTables

Similarly to the file input plugin, we wanted to keep our design consistent throughout, using Bootstrap elements when possible. Therefore, for any UI element stored in a table, we will be using the DataTables plugin developed by datatables.net. This plugin provides a bootstrap themed table, with dynamic, sortable headers, and a search bar which searches on all data in the table. This plugin will provide extra usability to users searching for data in a table with minimal effort from our team, since the plugin is already built.

5. Restrictions, Limitations, and Constraints

As time is our most limiting constraint, much of our limitations and constraints come from the amount of time given to develop the software. This is due to the complication of the application-side ASP.NET server code. Many of the algorithms required to complete the algorithm are complex and may require more time than allocated in one semester.

Another limitation to the application, which is not commonly referenced throughout the documentation, is the requirement for an internet connection. Since all code for this application happens on a server, whether it be a Microsoft IIS server for the .NET application or a MySQL database server, an internet connection is required. Without an internet connection, users will be unable to access the required servers along with any database code. Essentially, the application is useless without an internet connection.

6. Appendix

Appendix A - SQL Create Statements

```
use majorizor;
CREATE TABLE class_days (
  days char(4) NOT NULL,
  descr char(50) NOT NULL,
  PRIMARY KEY (days)
);

CREATE TABLE `course_by_degree` (
  `subject` char(4) NOT NULL,
  `catalog` char(10) NOT NULL,
  `degreeType` char(3) NOT NULL,
  `degree` char(2) NOT NULL,
  KEY `fk_cbd_courses` (`subject`,`catalog`),
  KEY `fk_cbd_degreeType` (`degreeType`),
  KEY `fk_cbd_degree` (`degree`),
    CONSTRAINT `fk_cbd_courses` FOREIGN KEY (`subject`, `catalog`) REFERENCES `courses`
(`subject`, `catalog`),
  CONSTRAINT `fk_cbd_degree` FOREIGN KEY (`degree`) REFERENCES `degrees` (`major`),
    CONSTRAINT `fk_cbd_degreeType` FOREIGN KEY (`degreeType`) REFERENCES `degree_type`
(`degreeType`)
);

CREATE TABLE advisees (
  advisorID int(11) NOT NULL,
  studentID int(11) NOT NULL,
  KEY fk_advisorID (advisorID),
  KEY fk_studentID (studentID),
  CONSTRAINT fk_advisorID FOREIGN KEY (advisorID) REFERENCES user (userID),
  CONSTRAINT fk_studentID FOREIGN KEY (studentID) REFERENCES student_info (userID)
);

CREATE TABLE courses (
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
  name char(100) NOT NULL,
  credits int(1) DEFAULT NULL,
  PRIMARY KEY (subject,catalog)
);

CREATE TABLE degrees (
  major char(2) NOT NULL,
  description char(50) DEFAULT NULL,
  PRIMARY KEY (major)
);
```

```
CREATE TABLE master_schedule (
  courseID int(8) NOT NULL,
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
  section char(2) NOT NULL,
  startTime time DEFAULT NULL,
  endTime time DEFAULT NULL,
  days char(4) DEFAULT NULL,
  PRIMARY KEY (courseID),
  KEY fk_mas_courses (subject,catalog),
  KEY days (days),
   CONSTRAINT fk_mas_courses FOREIGN KEY (subject, catalog) REFERENCES courses (subject,
catalog),
  CONSTRAINT master_schedule_ibfk_1 FOREIGN KEY (days) REFERENCES class_days (days)
);

CREATE TABLE degree_type (
  degreeType char(3) NOT NULL,
  descr char(50) NOT NULL,
  PRIMARY KEY (degreeType)
);

CREATE TABLE student_info (
  userID int(11) NOT NULL,
  year char(10) NOT NULL,
  major1 char(50) NOT NULL,
  major2 char(50) DEFAULT NULL,
  minor1 char(50) DEFAULT NULL,
  minor2 char(50) DEFAULT NULL,
  graduation char(20) NOT NULL,
  KEY fk_info_userID (userID),
  KEY fk_info_student_year (year),
  KEY fk_info_major1 (major1),
  KEY fk_info_major2 (major2),
  KEY pk_info_minor1 (minor1),
  KEY pk_info_minor2 (minor2),
  CONSTRAINT fk_info_major1 FOREIGN KEY (major1) REFERENCES degrees (major),
  CONSTRAINT fk_info_major2 FOREIGN KEY (major2) REFERENCES degrees (major),
  CONSTRAINT fk_info_student_year FOREIGN KEY (year) REFERENCES student_year (student_year),
  CONSTRAINT fk_info_userID FOREIGN KEY (userID) REFERENCES user (userID),
  CONSTRAINT pk_info_minor1 FOREIGN KEY (minor1) REFERENCES degrees (major),
  CONSTRAINT pk_info_minor2 FOREIGN KEY (minor2) REFERENCES degrees (major)
);

CREATE TABLE student_progress (
  studentID int(11) NOT NULL,
  subject char(4) NOT NULL,
  catalog char(10) NOT NULL,
```

```
  KEY fk_progress_id (studentID),
  KEY fk_progress_course (subject,catalog),
   CONSTRAINT fk_progress_course FOREIGN KEY (subject, catalog) REFERENCES courses (subject,
catalog),
  CONSTRAINT fk_progress_id FOREIGN KEY (studentID) REFERENCES user (userID)
);

CREATE TABLE student_year (
  student_year char(10) NOT NULL,
  PRIMARY KEY (student_year)
);


CREATE TABLE user_group (
  user_group char(10) NOT NULL,
  description char(50) DEFAULT NULL,
  PRIMARY KEY (user_group)
);

CREATE TABLE user (
  userID int(11) NOT NULL AUTO_INCREMENT,
  first_name char(50) DEFAULT NULL,
  last_name char(50) DEFAULT NULL,
  email char(50) NOT NULL,
  user_group char(10) DEFAULT NULL,
  PRIMARY KEY (userID, email),
  KEY fk_user_group (user_group),
  CONSTRAINT fk_user_group FOREIGN KEY (user_group) REFERENCES user_group (user_group)
);

CREATE TABLE user_password (
  userID int(11) DEFAULT NULL,
  password char(44) DEFAULT NULL,
  salt char(16) DEFAULT NULL,
  KEY fk_userID (userID),
  CONSTRAINT fk_userID FOREIGN KEY (userID) REFERENCES user (userID) ON DELETE CASCADE
);
```