Final Report

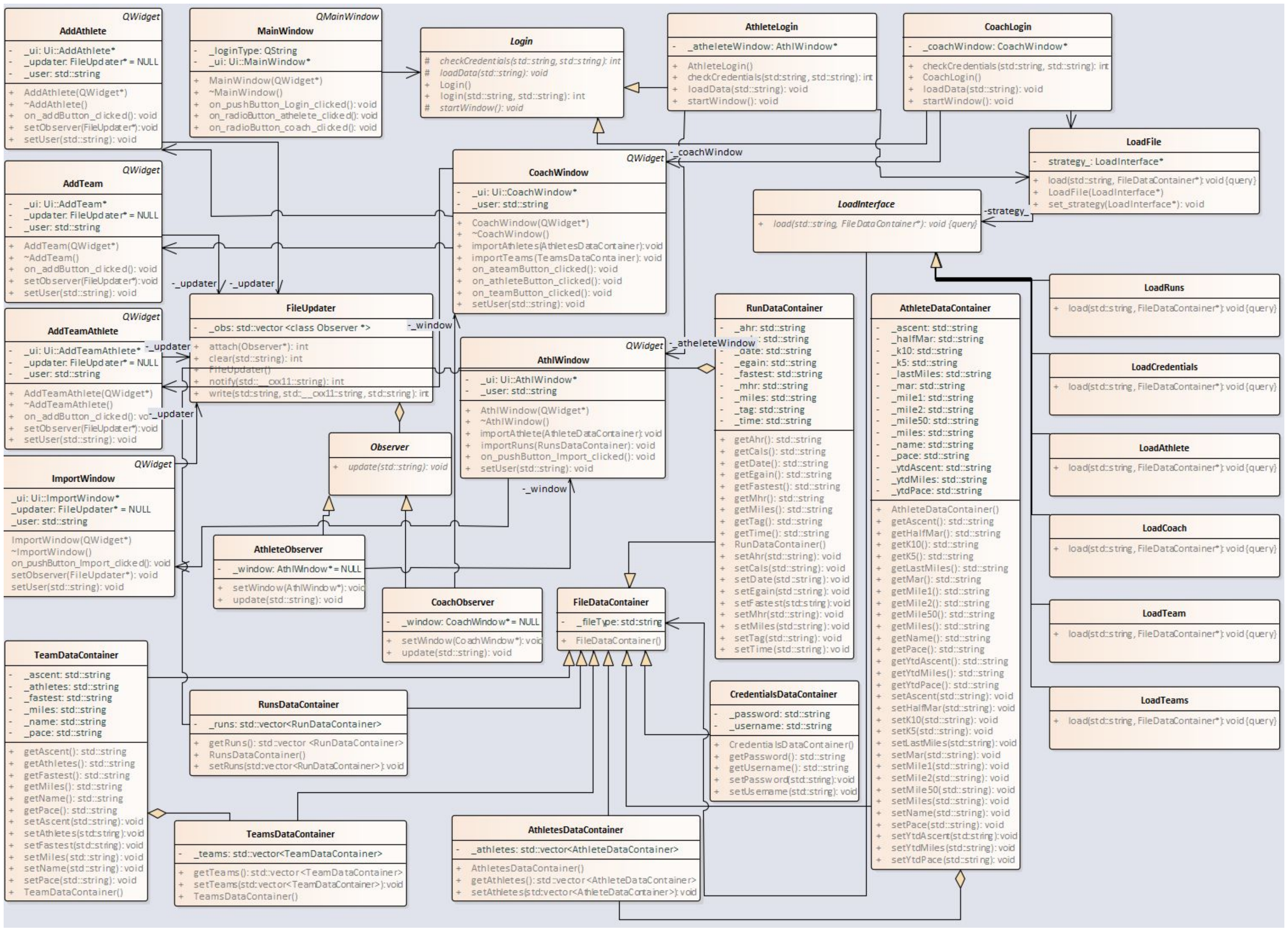
1. Thomas Lillis
2. TrailRunningPal -- A program for logging trail runs to view training progress as well as for coaches to organize and monitor their runners and teams.
3. Features Implemented

UR-ID	**Requirement**
UR-01	Athlete/Coach, can create an account
UR-02	Athlete can input run data (miles, mile times, vertical gain, etc.)
UR-03	Athlete/Coach can view run stats calculated from run data
UR-05	Athlete/Coach can view historical overall stats
UR-08	Coach can add Athlete to team
UR-09	Coach can create team
UR-10	Coach can browse Athletes in team
UR-11	Coach can compare (**numerically** and graphically) Athletes

4. Features not implemented

UR-ID	**Requirement**
UR-04	Athlete/Coach can visualize run data with graphs
UR-06	Athlete/Coach can view historical visualizations
UR-07	Athlete/Coach can set goals for Athlete
UR-11	Coach can compare (numerically and **graphically**) Athletes

5. Final Class Diagram

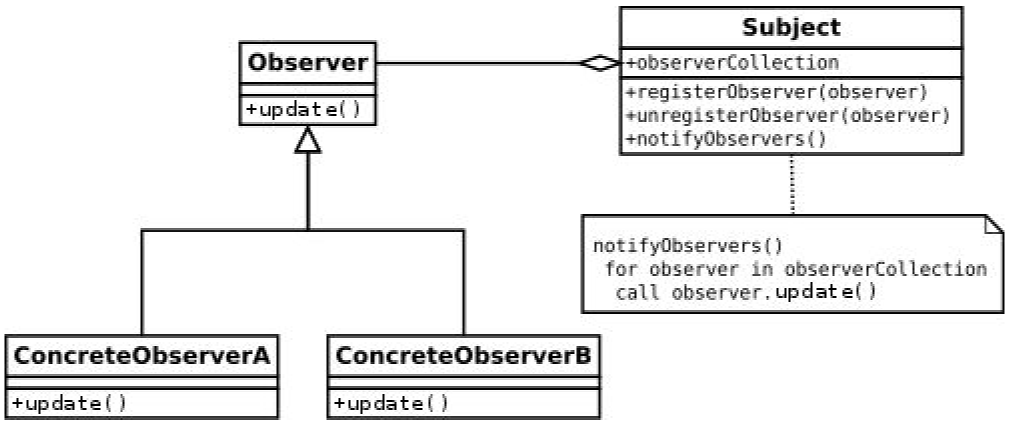

QWidget
AddAthlete
- _ui: Ui::AddAthlete*
- _updater: FileUpdater* = NULL
- _user: std::string
+ AddAthlete(QWidget*)
+ ~AddAthlete()
+ on_addButton_clicked(): void
+ setObserver(FileUpdater*):void
+ setUser(std::string): void
QMainWindow
MainWindow
- _loginType: QString
- _ui: Ui::MainWindow*
+ MainWindow(QWidget*)
+ ~MainWindow()
+ on_pushButton_Login_clicked(): void
+ on_radioButton_athelete_clicked(): void
+ on_radioButton_coach_clicked(): void
Login
# checkCredentials(std::string, std::string): int
# loadData(std::string): void
+ Login()
+ login(std::string, std::string): int
# startWindow(): void
AthleteLogin
- _atheleteWindow: AthlWindow*
+ AthleteLogin()
+ checkCredentials(std::string, std::string): int
+ loadData(std::string): void
+ startWindow(): void
CoachLogin
- _coachWindow: CoachWindow*
+ checkCredentials(std::string, std::string): int
+ CoachLogin()
+ loadData(std::string): void
+ startWindow(): void
LoadFile
- strategy_: LoadInterface*
+ load(std::string, FileDataContainer*): void {query}
+ LoadFile(LoadInterface*)
+ set_strategy(LoadInterface*): void
-strategy_
LoadInterface
+ load(std::string, FileDataContainer*): void {query}
-_coachWindow
QWidget
CoachWindow
- _ui: Ui::CoachWindow*
- _user: std::string
+ CoachWindow(QWidget*)
+ ~CoachWindow()
+ importAthletes(AthletesDataContainer):void
+ importTeams(TeamsDataContainer): void
+ on_ateamButton_clicked(): void
+ on_athleteButton_clicked(): void
+ on_teamButton_clicked(): void
+ setUser(std::string): void
QWidget
AddTeam
- _ui: Ui::AddTeam*
- _updater: FileUpdater* = NULL
- _user: std::string
+ AddTeam(QWidget*)
+ ~AddTeam()
+ on_addButton_clicked(): void
+ setObserver(FileUpdater*):void
+ setUser(std::string): void
-_updater
-_updater
FileUpdater
- _obs: std::vector <class Observer *>
+ attach(Observer*): int
+ clear(std::string): int
+ FileUpdater()
+ notify(std::__cxx11::string): int
+ write(std::string, std::__cxx11::string, std::string): int
-_window
QWidget
AddTeamAthlete
- _ui: Ui::AddTeamAthlete*
- _updater: FileUpdater* = NULL
- _user: std::string
+ AddTeamAthlete(QWidget*)
+ ~AddTeamAthlete()
+ on_addButton_clicked(): void
+ setObserver(FileUpdater*):void
+ setUser(std::string): void
-_updater
-_updater
QWidget
AthlWindow
- _ui: Ui::AthlWindow*
- _user: std::string
+ AthlWindow(QWidget*)
+ ~AthlWindow()
+ importAthlete(AthleteDataContainer): void
+ importRuns(RunsDataContainer): void
+ on_pushButton_Import_clicked(): void
+ setUser(std::string): void
-_atheleteWindow
-_window
Observer
+ update(std::string): void
QWidget
ImportWindow
_ui: Ui::ImportWindow*
_updater: FileUpdater* = NULL
_user: std::string
ImportWindow(QWidget*)
~ImportWindow()
on_pushButton_Import_clicked(): void
setObserver(FileUpdater*): void
setUser(std::string): void
AthleteObserver
- _window: AthlWindow* = NULL
+ setWindow(AthlWindow*): void
+ update(std::string): void
CoachObserver
- _window: CoachWindow* = NULL
+ setWindow(CoachWindow*): void
+ update(std::string): void
FileDataContainer
- _fileType: std::string
+ FileDataContainer()
RunDataContainer
- _ahr: std::string
- _date: std::string
- _egain: std::string
- _fastest: std::string
- _mhr: std::string
- _miles: std::string
- _tag: std::string
- _time: std::string
+ getAhr(): std::string
+ getCals(): std::string
+ getDate(): std::string
+ getEgain(): std::string
+ getFastest(): std::string
+ getMhr(): std::string
+ getMiles(): std::string
+ getTag(): std::string
+ getTime(): std::string
+ RunDataContainer()
+ setAhr(std::string): void
+ setCals(std::string): void
+ setDate(std::string): void
+ setEgain(std::string): void
+ setFastest(std::string): void
+ setMhr(std::string): void
+ setMiles(std::string): void
+ setTag(std::string): void
+ setTime(std::string): void
AthleteDataContainer
- _ascent: std::string
- _halfMar: std::string
- _k10: std::string
- _k5: std::string
- _lastMiles: std::string
- _mar: std::string
- _mile1: std::string
- _mile2: std::string
- _mile50: std::string
- _miles: std::string
- _name: std::string
- _pace: std::string
- _ytdAscent: std::string
- _ytdMiles: std::string
- _ytdPace: std::string
+ AthleteDataContainer()
+ getAscent(): std::string
+ getHalfMar(): std::string
+ getK10(): std::string
+ getK5(): std::string
+ getLastMiles(): std::string
+ getMar(): std::string
+ getMile1(): std::string
+ getMile2(): std::string
+ getMile50(): std::string
+ getMiles(): std::string
+ getName(): std::string
+ getPace(): std::string
+ getYtdAscent(): std::string
+ getYtdMiles(): std::string
+ getYtdPace(): std::string
+ setAscent(std::string): void
+ setHalfMar(std::string): void
+ setK10(std::string): void
+ setK5(std::string): void
+ setLastMiles(std::string): void
+ setMar(std::string): void
+ setMile1(std::string): void
+ setMile2(std::string): void
+ setMile50(std::string): void
+ setMiles(std::string): void
+ setName(std::string): void
+ setPace(std::string): void
+ setYtdAscent(std::string): void
+ setYtdMiles(std::string): void
+ setYtdPace(std::string): void
LoadRuns
+ load(std::string, FileDataContainer*): void {query}
LoadCredentials
+ load(std::string, FileDataContainer*): void {query}
LoadAthlete
+ load(std::string, FileDataContainer*): void {query}
LoadCoach
+ load(std::string, FileDataContainer*): void {query}
LoadTeam
+ load(std::string, FileDataContainer*): void {query}
LoadTeams
+ load(std::string, FileDataContainer*): void {query}
TeamDataContainer
- _ascent: std::string
- _athletes: std::string
- _fastest: std::string
- _miles: std::string
- _name: std::string
- _pace: std::string
+ getAscent(): std::string
+ getAthletes(): std::string
+ getFastest(): std::string
+ getMiles(): std::string
+ getName(): std::string
+ getPace(): std::string
+ setAscent(std::string): void
+ setAthletes(std::string): void
+ setFastest(std::string): void
+ setMiles(std::string): void
+ setName(std::string): void
+ setPace(std::string): void
+ TeamDataContainer()
RunsDataContainer
- _runs: std::vector<RunDataContainer>
+ getRuns(): std::vector <RunDataContainer>
+ RunsDataContainer()
+ setRuns(std::vector<RunDataContainer>): void
CredentialsDataContainer
- _password: std::string
- _username: std::string
+ CredentialsDataContainer()
+ getPassword(): std::string
+ getUsername(): std::string
+ setPassword(std::string): void
+ setUsername(std::string): void
TeamsDataContainer
- _teams: std::vector<TeamDataContainer>
+ getTeams(): std::vector <TeamDataContainer>
+ setTeams(std::vector<TeamDataContainer>): void
+ TeamsDataContainer()
AthletesDataContainer
- _athletes: std::vector<AthleteDataContainer>
+ AthletesDataContainer()
+ getAthletes(): std::vector <AthleteDataContainer>
+ setAthletes(std::vector<AthleteDataContainer>): void

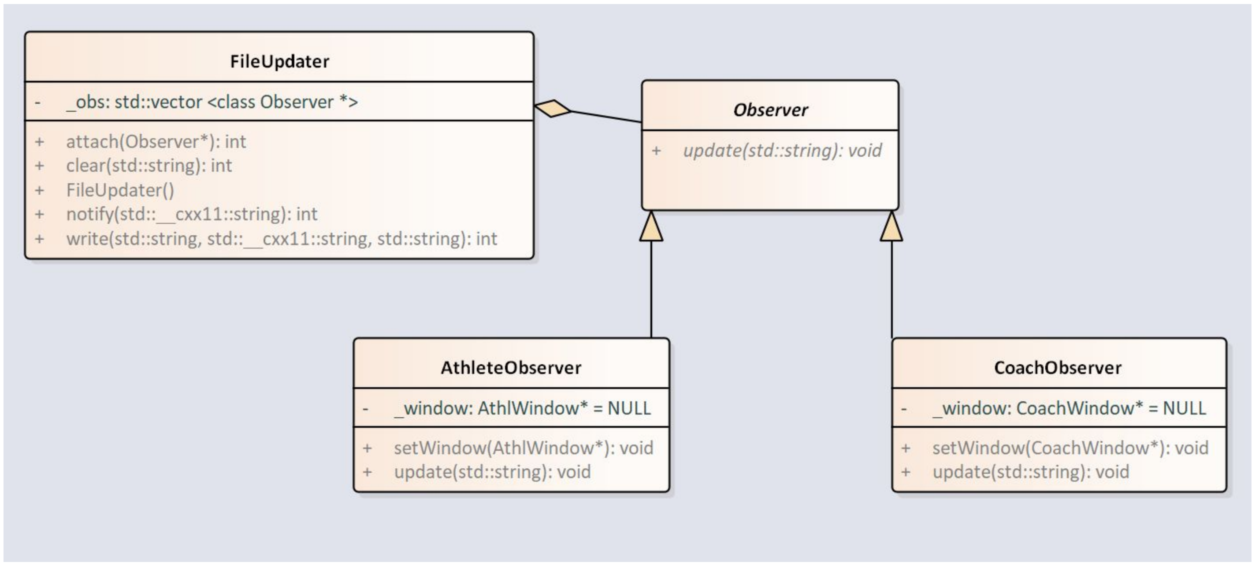
A lot changed between my part 2 diagram and my final diagram. For my part 2 diagram I didn't have any of my design patterns as part of my class diagram. This new diagram includes adding the strategy design pattern for loading different types of files. This can be seen in all the classes that start with "Load". I also added the template design pattern which can be seen in the classes containing "Login" in their names. Lastly, I added the observer design pattern for notifying when the user interface should update. This can be seen in the "Observer" and "FileUpdater" classes. See part 6 for more details about the design pattern implementations.

The only similar parts between my two diagrams are the GUI aspects and a few classes that I mostly used as containers. I tried not to use classes just as containers but it proved helpful incorporating them polymorphically into my strategy design pattern to read different types of data. Another difference between my original and final diagram is I had not planned ahead for GUI widget forms needed for importing data. These can be seen in the classes that start with “Add”. I also removed GUI elements for individual runs and teams.

6.

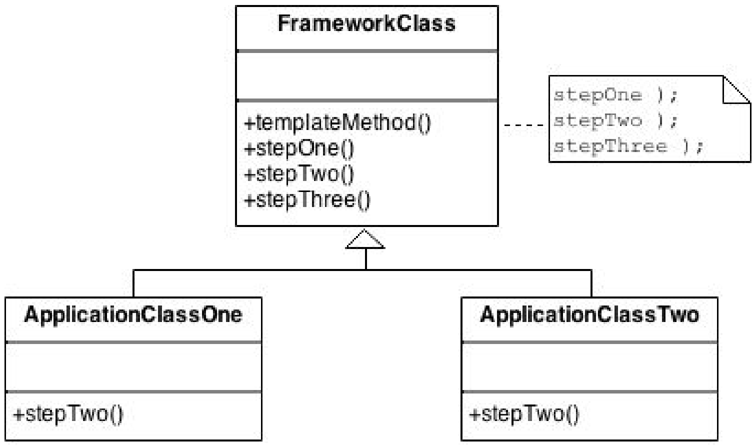
a. <u>Observer</u>

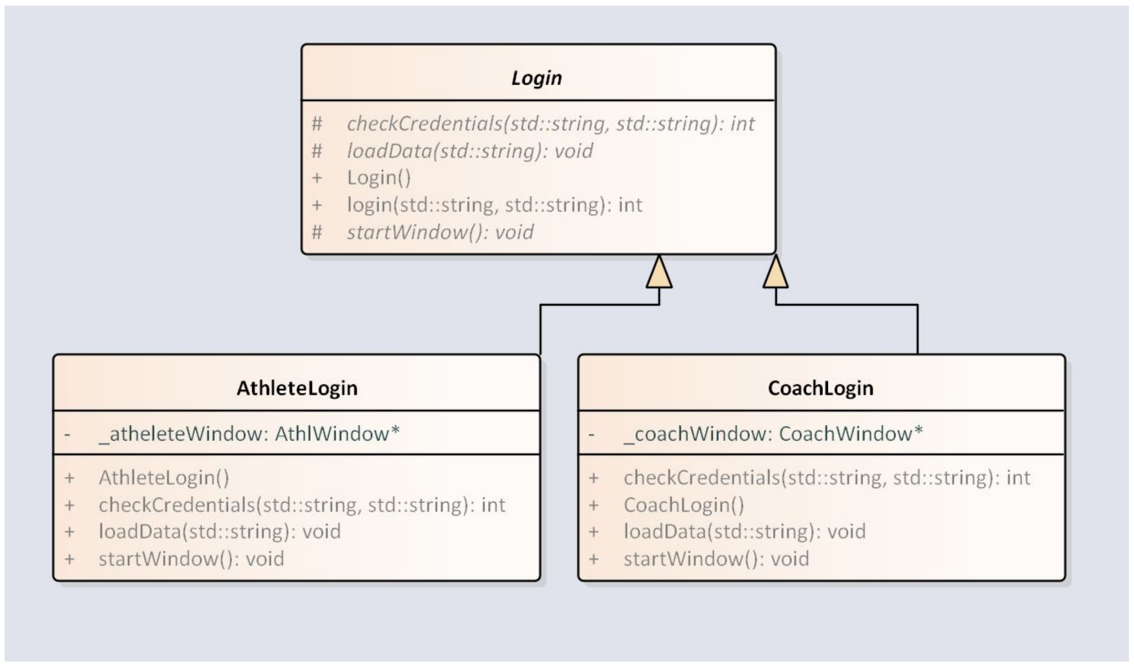



FileUpdater
- _obs: std::vector <class Observer *>
+ attach(Observer*): int
+ clear(std::string): int
+ FileUpdater()
+ notify(std::__cxx11::string): int
+ write(std::string, std::__cxx11::string, std::string): int
Observer
+ update(std::string): void
AthleteObserver
- _window: AthlWindow* = NULL
+ setWindow(AthlWindow*): void
+ update(std::string): void
CoachObserver
- _window: CoachWindow* = NULL
+ setWindow(CoachWindow*): void
+ update(std::string): void

I decided to use the observer design pattern because I needed my GUI to update whenever a data file was prepended in my program. The observer design pattern let achieve this by having observers that have the function update() that is called for each whenever the FileUpdater calls notify() to notify of a new write to a file. When the observers call update() they reread the files (Athlete files for athlete observer and coach files for coach observer) and update their interfaces. Note that both types can be set to reload on a notify because coach data often depends on athletes data and the overhead is low for reloading files. Observer base class is abstract because update() is implemented as a virtual function. The write() function is part of the FileUpdater and is the function that actually writes the new data to a file but is not actually part of the design pattern, notify() just needs to be called after the write() function. Each observer needs to be attached by calling the attach() function and then it will be notified during a notify().

b. Template



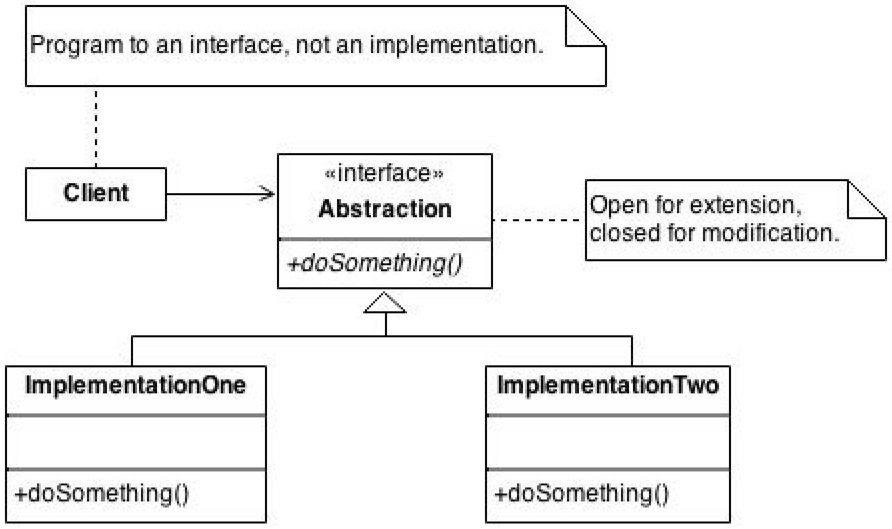

Login
# checkCredentials(std::string, std::string): int
# loadData(std::string): void
+ Login()
+ login(std::string, std::string): int
# startWindow(): void
AthleteLogin
- _atheleteWindow: AthlWindow*
+ AthleteLogin()
+ checkCredentials(std::string, std::string): int
+ loadData(std::string): void
+ startWindow(): void
CoachLogin
- _coachWindow: CoachWindow*
+ checkCredentials(std::string, std::string): int
+ CoachLogin()
+ loadData(std::string): void
+ startWindow(): void

I decided to use the template design pattern for my program because the program needed to follow the same steps whether a coach or an athlete was starting the program. All the steps were the same at a high level but there needed to be modifications depending on the type of user. The steps were:

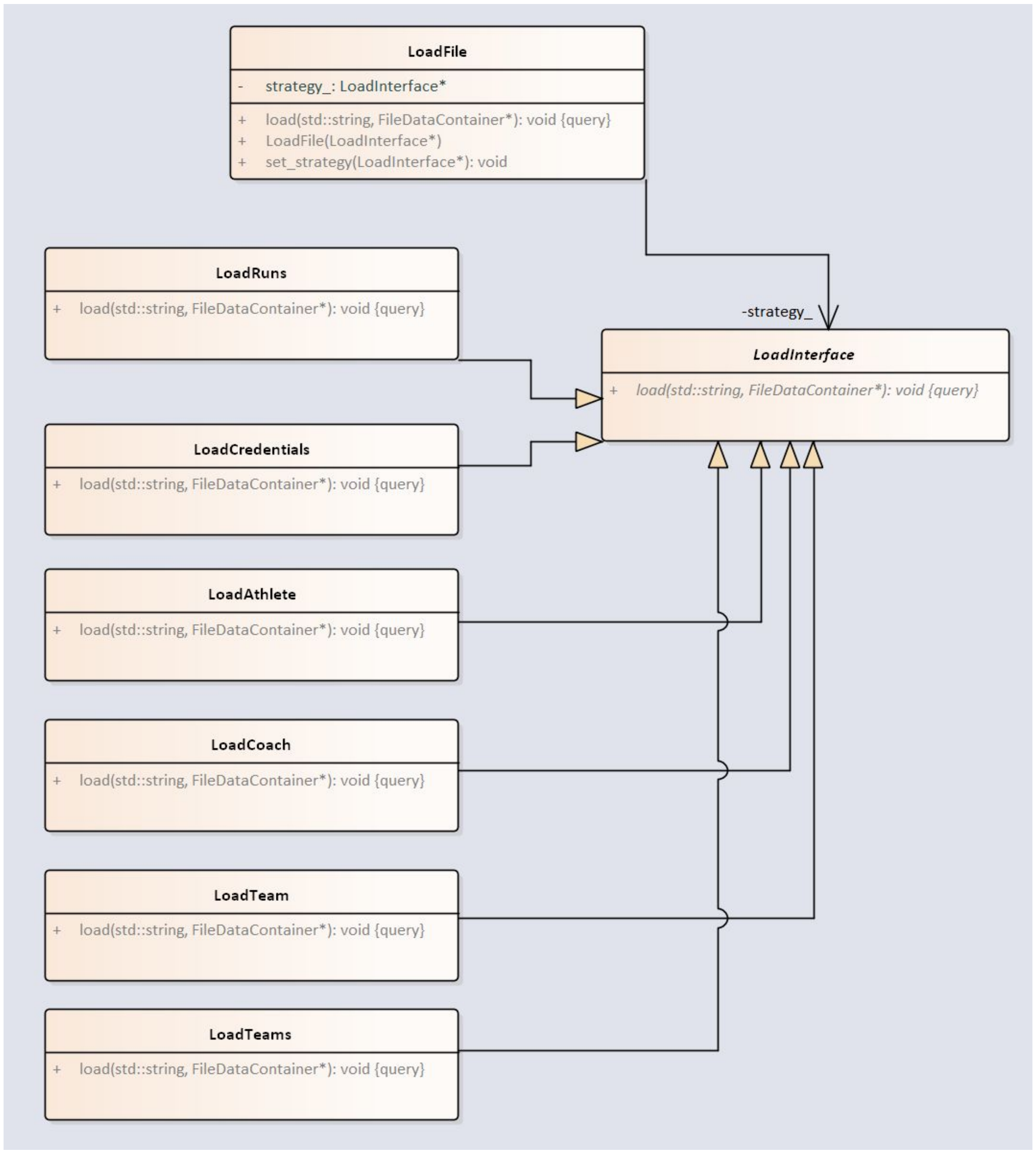
1. Check Credentials
2. Load Data
3. Start Window

Both have very similar ways of checking credentials but then needed to load in completely different types of data and start different types of windows. This was implemented in C++ and I did end up making all three functions in the parent class virtual making the parent class abstract. I did this because I had no defaults for each of the steps so I decided to make them required to be overridden.

c. <u>Strategy</u>



I


LoadFile
- strategy_: LoadInterface*
+ load(std::string, FileDataContainer*): void {query}
+ LoadFile(LoadInterface*)
+ set_strategy(LoadInterface*): void
-strategy_
LoadInterface
+ load(std::string, FileDataContainer*): void {query}
LoadRuns
+ load(std::string, FileDataContainer*): void {query}
LoadCredentials
+ load(std::string, FileDataContainer*): void {query}
LoadAthlete
+ load(std::string, FileDataContainer*): void {query}
LoadCoach
+ load(std::string, FileDataContainer*): void {query}
LoadTeam
+ load(std::string, FileDataContainer*): void {query}
LoadTeams
+ load(std::string, FileDataContainer*): void {query}

decided to implement the strategy design pattern because I had lots of different data files I needed to load in that were each loaded in in a different format. These 6 different file formats varied enough that I decided to implement the strategy pattern. When loading in a file a user will create and instance of the LoadFile class and then give it a strategy class to use to do the actual loading of the file. Then the user can call load file and load in a file of the format. I made load() virtual in LoadInterface making it abstract and load() required to be overridden. There are no interfaces in C++.

7. I learned several important things during this project. First, I learned that it is difficult to plan ahead and even when you spend a lot of time planning for the future there are always things you miss. Once I had my design planned out I hit several roadblocks for things that I had not anticipated. I did not think about how I need to make a unique form for each of my methods of adding new athletes, teams, athletes to coaches and athletes to teams. This quickly added several classes I had not anticipated in my initial design and though they were trivial to add were still micro timesinks. That being said, It was insightful to me how much planning the entire architecture of the program helped. I believe that there would have been many more instances similar to this one had I not planned out the rest of the architure. Though the overhead initially seems intimidating, it is definitely worth it for the development to plan the architecture as much as you can ahead.

   Next, I learned how even when using design patterns how quickly things can get complicated. It seemed sudden that I had 30 classes to implement fairly simple functionality. I think that the design patterns were helpful in making the design clean and extendable but they are not always succinct. It's also easy to look back and see the poor design decisions I made which may have lead to the verbosity of my system which is just due to me being a novice developer.

   Finally, I learned my methods for storing data in files was atrocious and should never be used in a real system. The steps I needed to add a file type and parse through it were very unreasonable. I was considering using a database which I initially thought was overkill for the project which I now see was naive. As I went along implementing each way to read a file as well as the data containers associated with the filetype I quickly knew I had made a mistake. The time I spent doing this could have been spent improving my application elsewhere, possibility implementing more of the features I had planned to implement.