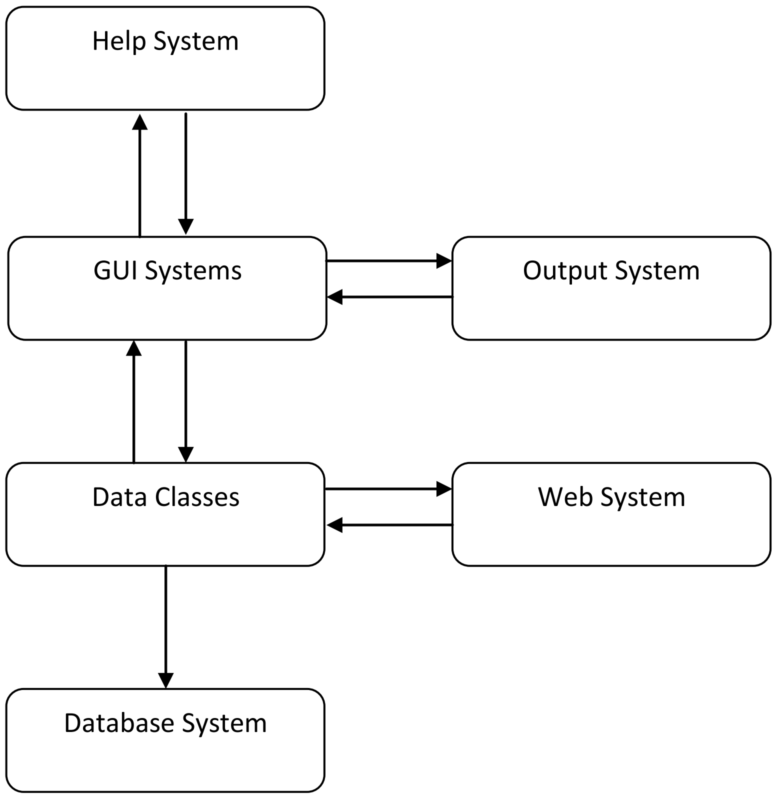
Design and Architecture

Report on the design and architecture of the sub-systems defined in the requirements document.

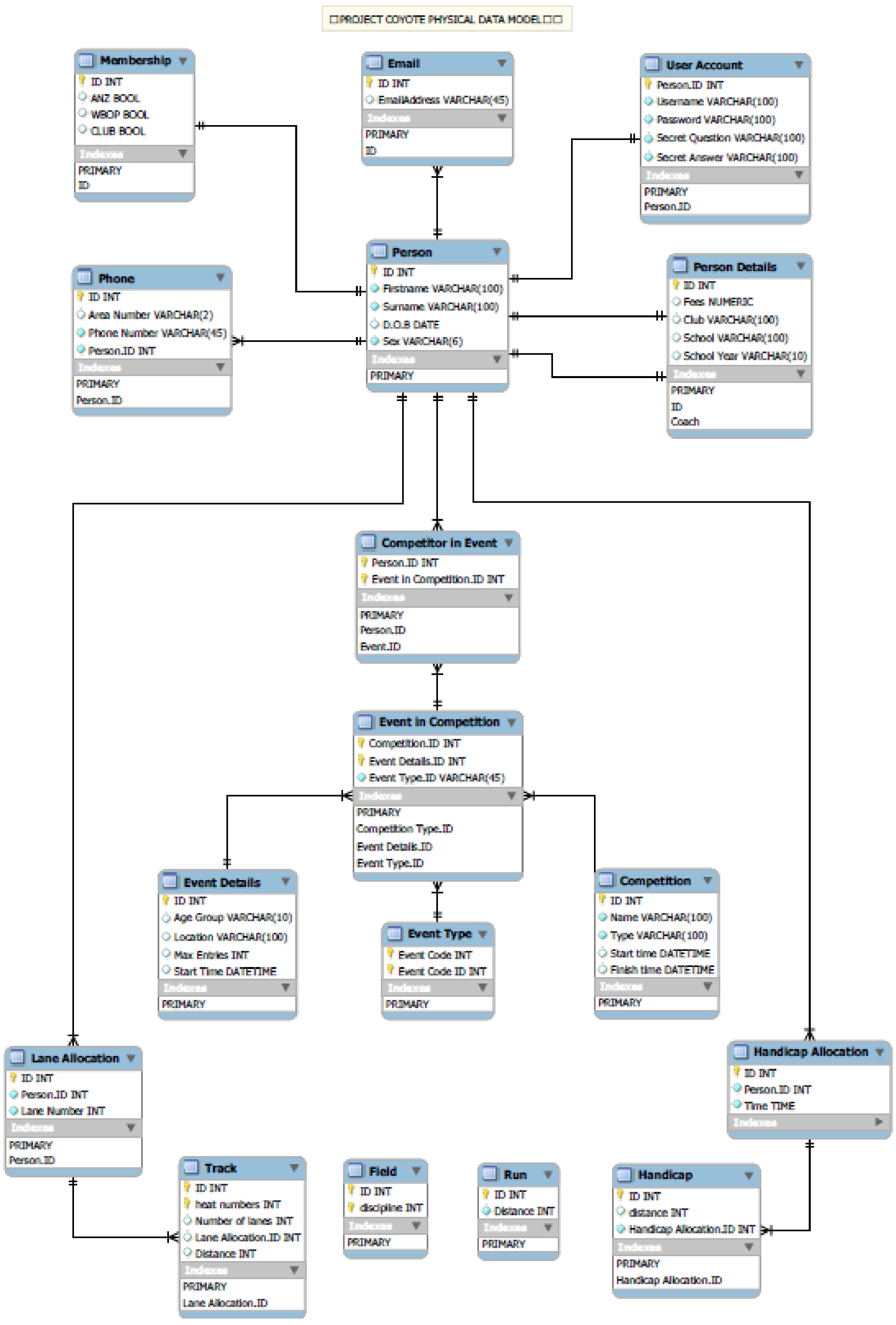
Table of Contents

1. Overview

Project Coyote consists of sex different subsystems: Database System, GUI Systems, Data Classes, Web System, Output and the Help System. A general overview can be seen below.

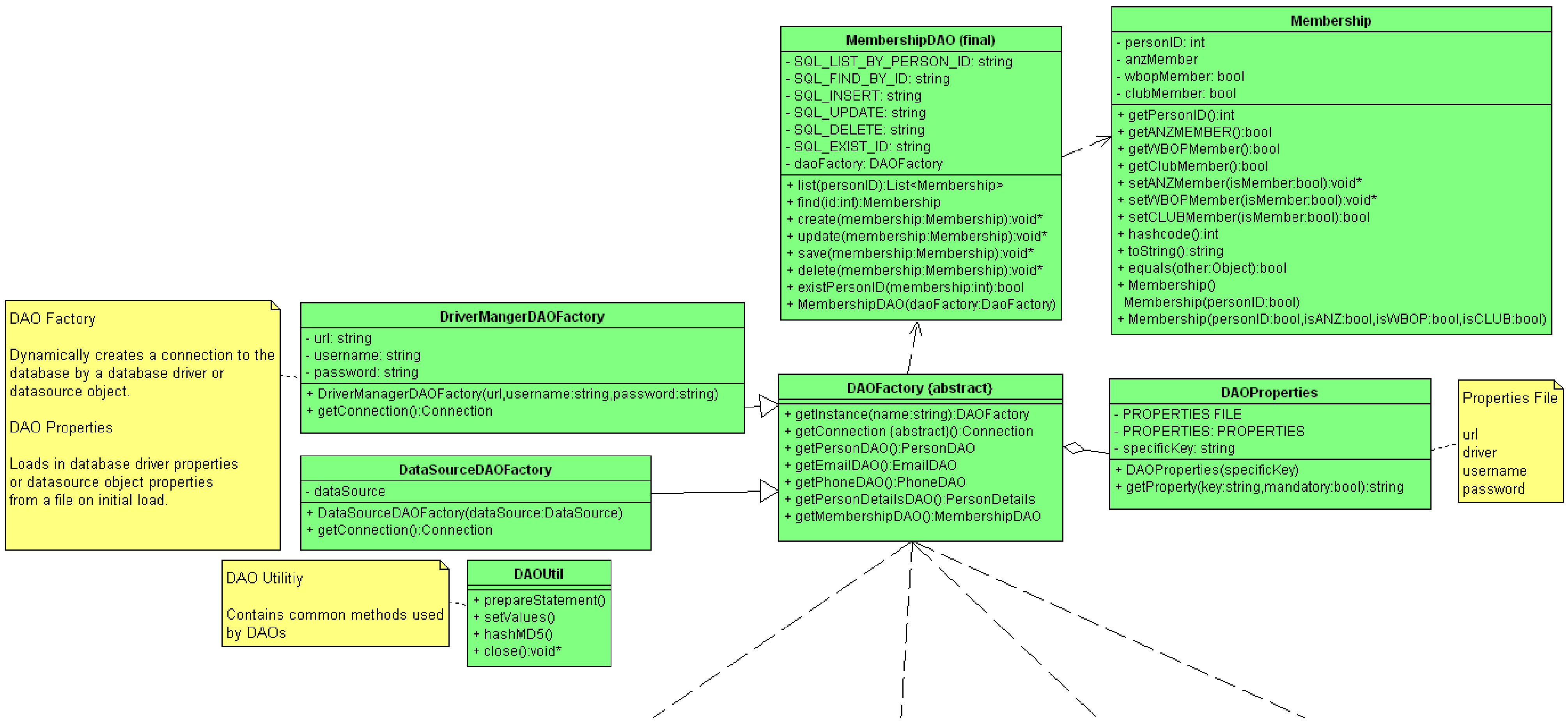


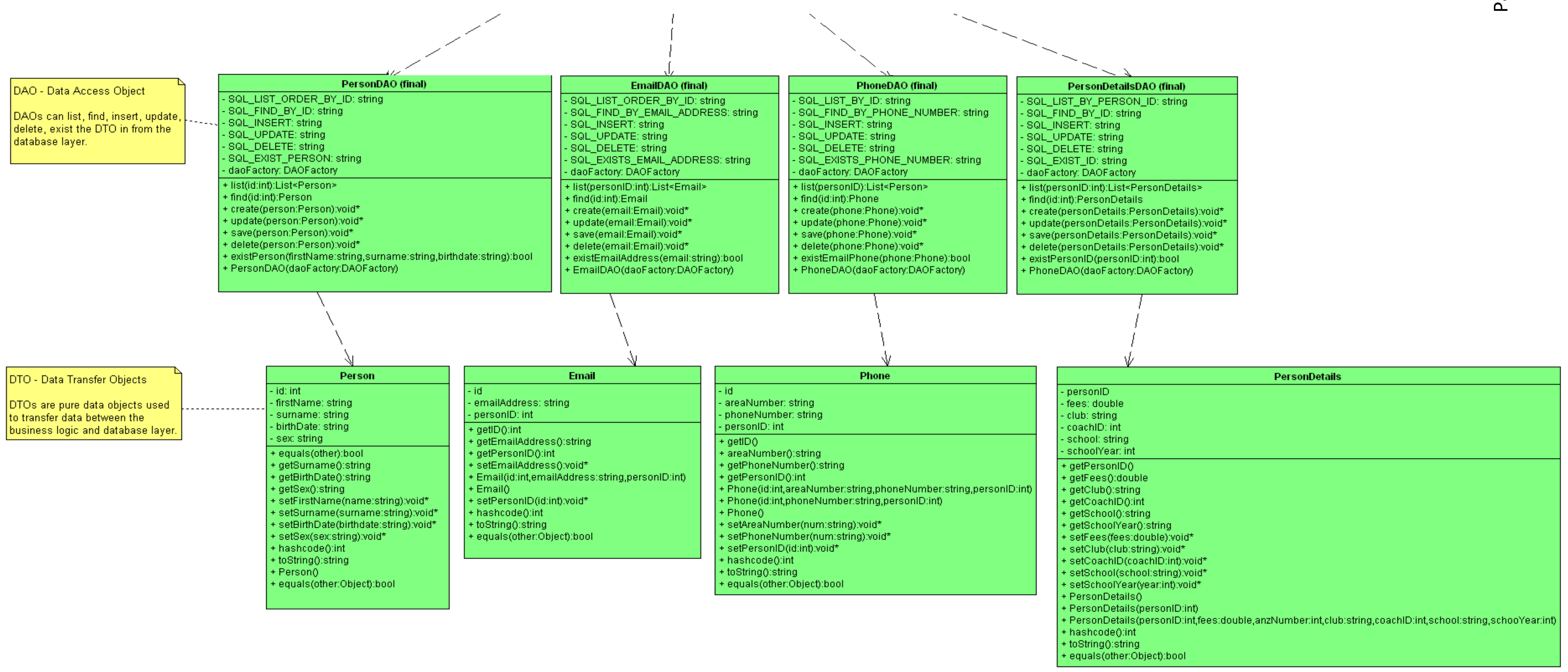
2. Database System



PDM description

Physical Data Model representing the relationships between key entities: People, Competitions and Events. Each Event belongs to a Competition, has Event Details and an Event Type. An Event Type contains an Event code specifying a Track, Field, or Run entity. Track and Handicap entities have foreign keys Lane Allocation and Handicap Allocation respectively, storing the person id and their corresponding lane number or time.


DAO Factory
Dynamically creates a connection to the database by a database driver or datasource object.
DAO Properties
Loads in database driver properties or datasource object properties from a file on initial load.
DriverMangerDAOFactory
- url: string
- username: string
- password: string
+ DriverManagerDAOFactory(url,username:string,password:string)
+ getConnection():Connection
DataSourceDAOFactory
- dataSource
+ DataSourceDAOFactory(dataSource:DataSource)
+ getConnection():Connection
DAO Utilitiy
Contains common methods used by DAOs
DAOUtil
+ prepareStatement()
+ setValues()
+ hashMD5()
+ close():void*
MembershipDAO (final)
- SQL_LIST_BY_PERSON_ID: string
- SQL_FIND_BY_ID: string
- SQL_INSERT: string
- SQL_UPDATE: string
- SQL_DELETE: string
- SQL_EXIST_ID: string
- daoFactory: DAOFactory
+ list(personID):List<Membership>
+ find(id:int):Membership
+ create(membership:Membership):void*
+ update(membership:Membership):void*
+ save(membership:Membership):void*
+ delete(membership:Membership):void*
+ existPersonID(membership:int):bool
+ MembershipDAO(daoFactory:DaoFactory)
Membership
- personID: int
- anzMember
- wbopMember: bool
- clubMember: bool
+ getPersonID():int
+ getANZMEMBER():bool
+ getWBOPMember():bool
+ getClubMember():bool
+ setANZMember(isMember:bool):void*
+ setWBOPMember(isMember:bool):void*
+ setCLUBMember(isMember:bool):bool
+ hashcode():int
+ toString():string
+ equals(other:Object):bool
+ Membership()
Membership(personID:bool)
+ Membership(personID:bool,isANZ:bool,isWBOP:bool,isCLUB:bool)
DAOFactory {abstract}
+ getInstance(name:string):DAOFactory
+ getConnection {abstract}():Connection
+ getPersonDAO():PersonDAO
+ getEmailDAO():EmailDAO
+ getPhoneDAO():PhoneDAO
+ getPersonDetailsDAO():PersonDetails
+ getMembershipDAO():MembershipDAO
DAOProperties
- PROPERTIES FILE
- PROPERTIES: PROPERTIES
- specificKey: string
+ DAOProperties(specificKey)
+ getProperty(key:string,mandatory:bool):string
Properties File
url
driver
username
password

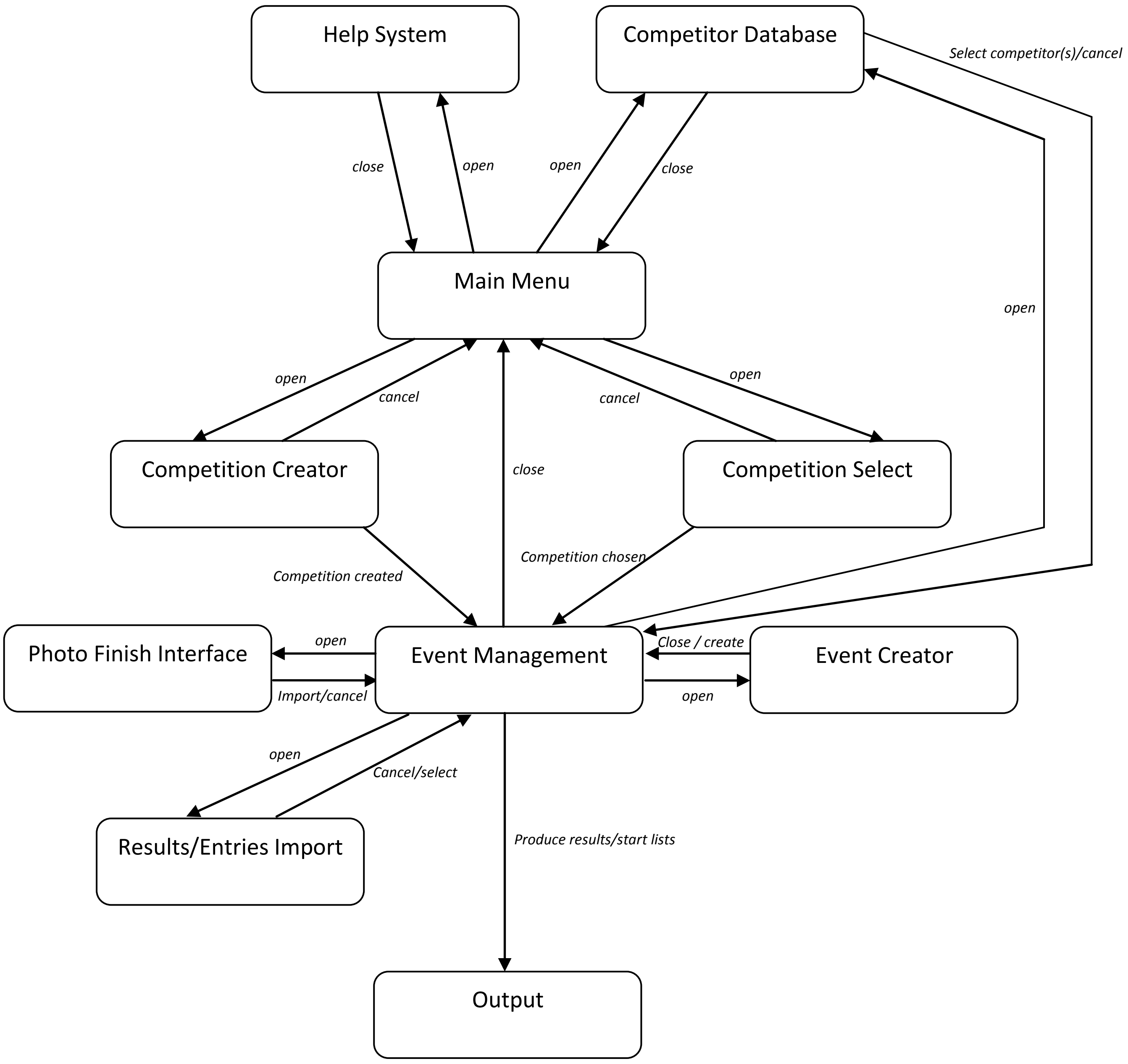

DAO - Data Access Object
DAOs can list, find, insert, update, delete, exist the DTO in from the database layer.
PersonDAO (final)
- SQL_LIST_ORDER_BY_ID: string
- SQL_FIND_BY_ID: string
- SQL_INSERT: string
- SQL_UPDATE: string
- SQL_DELETE: string
- SQL_EXIST_PERSON: string
- daoFactory: DAOFactory
+ list(id:int):List<Person>
+ find(id:int):Person
+ create(person:Person):void*
+ update(person:Person):void*
+ save(person:Person):void*
+ delete(person:Person):void*
+ existPerson(firstName:string,surname:string,birthdate:string):bool
+ PersonDAO(daoFactory:DAOFactory)
EmailDAO (final)
- SQL_LIST_ORDER_BY_ID: string
- SQL_FIND_BY_EMAIL_ADDRESS: string
- SQL_INSERT: string
- SQL_UPDATE: string
- SQL_DELETE: string
- SQL_EXISTS_EMAIL_ADDRESS: string
- daoFactory: DAOFactory
+ list(personID:int):List<Email>
+ find(id:int):Email
+ create(email:Email):void*
+ update(email:Email):void*
+ save(email:Email):void*
+ delete(email:Email):void*
+ existEmailAddress(email:string):bool
+ EmailDAO(daoFactory:DAOFactory)
PhoneDAO (final)
- SQL_LIST_BY_ID: string
- SQL_FIND_BY_PHONE_NUMBER: string
- SQL_INSERT: string
- SQL_UPDATE: string
- SQL_DELETE: string
- SQL_EXISTS_PHONE_NUMBER: string
- daoFactory: DAOFactory
+ list(personID):List<Person>
+ find(id:int):Phone
+ create(phone:Phone):void*
+ update(phone:Phone):void*
+ save(phone:Phone):void*
+ delete(phone:Phone):void*
+ existEmailPhone(phone:Phone):bool
+ PhoneDAO(daoFactory:DAOFactory)
PersonDetailsDAO (final)
- SQL_LIST_BY_PERSON_ID: string
- SQL_FIND_BY_ID: string
- SQL_INSERT: string
- SQL_UPDATE: string
- SQL_DELETE: string
- SQL_EXIST_ID: string
- daoFactory: DAOFactory
+ list(personID:int):List<PersonDetails>
+ find(id:int):PersonDetails
+ create(personDetails:PersonDetails):void*
+ update(personDetails:PersonDetails):void*
+ save(personDetails:PersonDetails):void*
+ delete(personDetails:PersonDetails):void*
+ existPersonID(personID:int):bool
+ PhoneDAO(daoFactory:DAOFactory)
DTO - Data Transfer Objects
DTOs are pure data objects used to transfer data between the business logic and database layer.
Person
- id: int
- firstName: string
- surname: string
- birthDate: string
- sex: string
+ equals(other):bool
+ getSurname():string
+ getBirthDate():string
+ getSex():string
+ setFirstName(name:string):void*
+ setSurname(surname:string):void*
+ setBirthDate(birthdate:string):void*
+ setSex(sex:string):void*
+ hashcode():int
+ toString():string
+ Person()
+ equals(other:Object):bool
Email
- id
- emailAddress: string
- personID: int
+ getID():int
+ getEmailAddress():string
+ getPersonID():int
+ setEmailAddress():void*
+ Email(id:int,emailAddress:string,personID:int)
+ Email()
+ setPersonID(id:int):void*
+ hashcode():int
+ toString():string
+ equals(other:Object):bool
Phone
- id
- areaNumber: string
- phoneNumber: string
- personID: int
+ getID()
+ areaNumber():string
+ getPhoneNumber():string
+ getPersonID():int
+ Phone(id:int,areaNumber:string,phoneNumber:string,personID:int)
+ Phone(id:int,phoneNumber:string,personID:int)
+ Phone()
+ setAreaNumber(num:string):void*
+ setPhoneNumber(num:string):void*
+ setPersonID(id:int):void*
+ hashcode():int
+ toString():string
+ equals(other:Object):bool
PersonDetails
- personID
- fees: double
- club: string
- coachID: int
- school: string
- schoolYear: int
+ getPersonID()
+ getFees():double
+ getClub():string
+ getCoachID():int
+ getSchool():string
+ getSchoolYear():string
+ setFees(fees:double):void*
+ setClub(club:string):void*
+ setCoachID(coachID:int):void*
+ setSchool(school:string):void*
+ setSchoolYear(year:int):void*
+ PersonDetails()
+ PersonDetails(personID:int)
+ PersonDetails(personID:int,fees:double,anzNumber:int,club:string,coachID:int,school:string,schooYear:int)
+ hashcode():int
+ toString():string
+ equals(other:Object):bool

UML diagram

Partial UML implementation representing how data is encapsulated stored and retrieved from a database in the database layer separating business logic from database logic. Refer to yellow notes in the UML diagram for a brief description of each class.

3. GUI Systems

3.1 Overview



3.2 Design Motivation and Priorities

From the outset we decided that the GUI for our solution needed to be a main priority: it is the major way in which the user interacts with the program, they don't really care much about what lies behind it. Their overall impression of the program tends to be based solely upon the GUI and what it lets them do. Consequently, we decided that the GUI needed to be "as simple as possible and no simpler." This entails that it needs to be regular and intuitive, presenting an easy learning curve. We also intend to streamline the interface, making it workflow oriented and designed to make the most common use cases fast. The GUI for our program presents one of our major points of difference from competing software that is much more difficult to use

3.3 Implementation Detail

The GUI itself will be implemented in java using Swing. It will communicate directly with the database classes to the database sub-system.

3.4 Design Choices

We came up with many different ideas during the process of choosing a UI design. Initially, we looked at how we could best present data to the user and make it easy for them to edit it. We came up with various combinations, typically involving tabs that allowed the user to switch quickly between all competitions and all competitors. Within the competitions tab, all competitions would then be displayed. The user would select a competition and the display would show its details and events. Selecting an event would then open it in a new window, allowing it to be edited. Menus and toolbars would feature throughout this design.

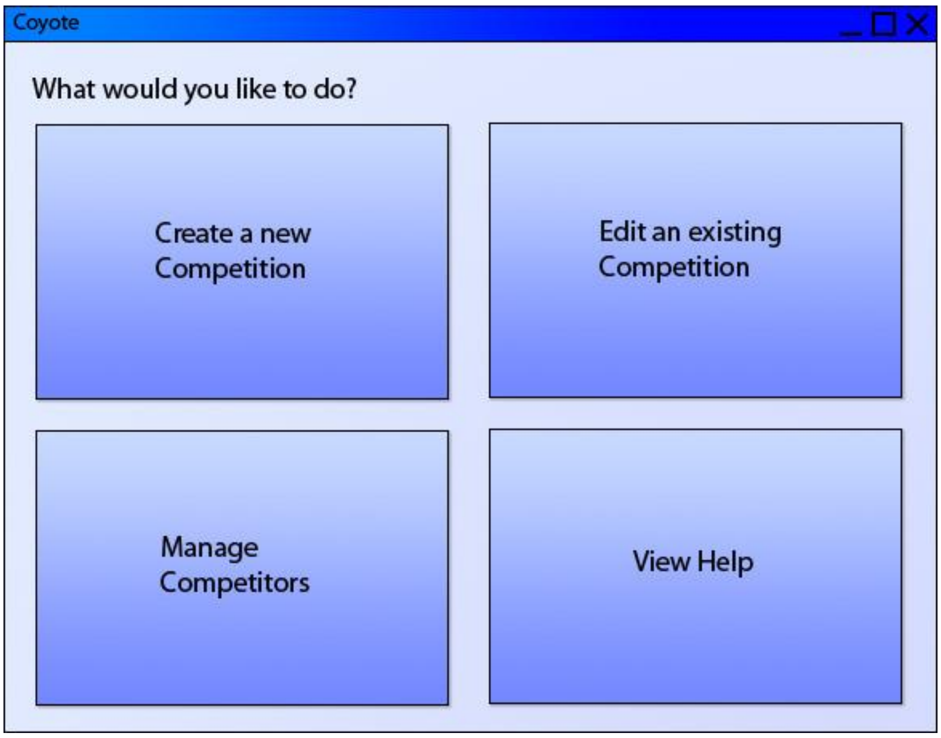
Initially, this idea seemed good. Jesse made mention that this presentation might confuse the user slightly, with various distinctions needing to be made between the different components of a competition, making the point that the less they need to use the program documentation, the better. Braden also raised the objection that the user interface would be displaying a lot of information that the user wouldn't need at the time and thus be chewing up space that could otherwise be used for what they were actually working on. In addition, this wasn't serving to make the most common use cases fast and easy.

After further analysis, we opted to make some changes. First, we decided to present a splash screen to the user when they weren't working on anything, helping to orient it toward the tasks they wanted to perform. To further improve task flow, we did away with tool bars and menus, instead choosing to use a task-oriented ribbon of grouped buttons that updates dynamically based upon what the user is doing (similar to that employed in current Microsoft Office products). Information that isn't needed at the time can now be hidden from view, improving workflow and making the program easier to use.

3.5 Design Sketches and Details

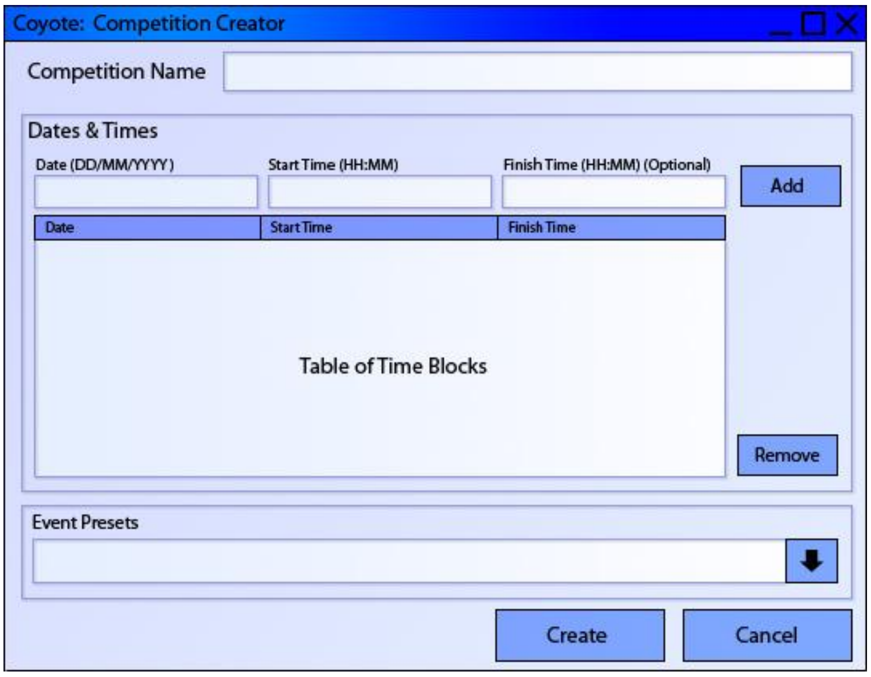
What follows are our current design sketches, showing how we intend the program to look and a flow diagram illustrating the way major interactions change the state of the GUI.

3.5.1 Main Menu



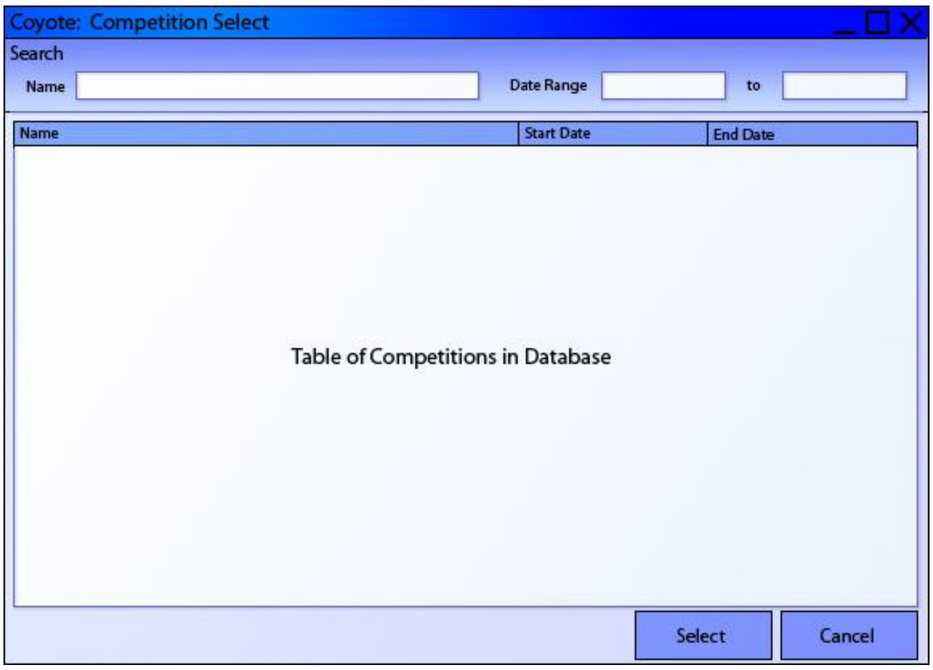
The main menu presents a series of simple options to the user, prompting them to select an action.

3.5.2 New Competition



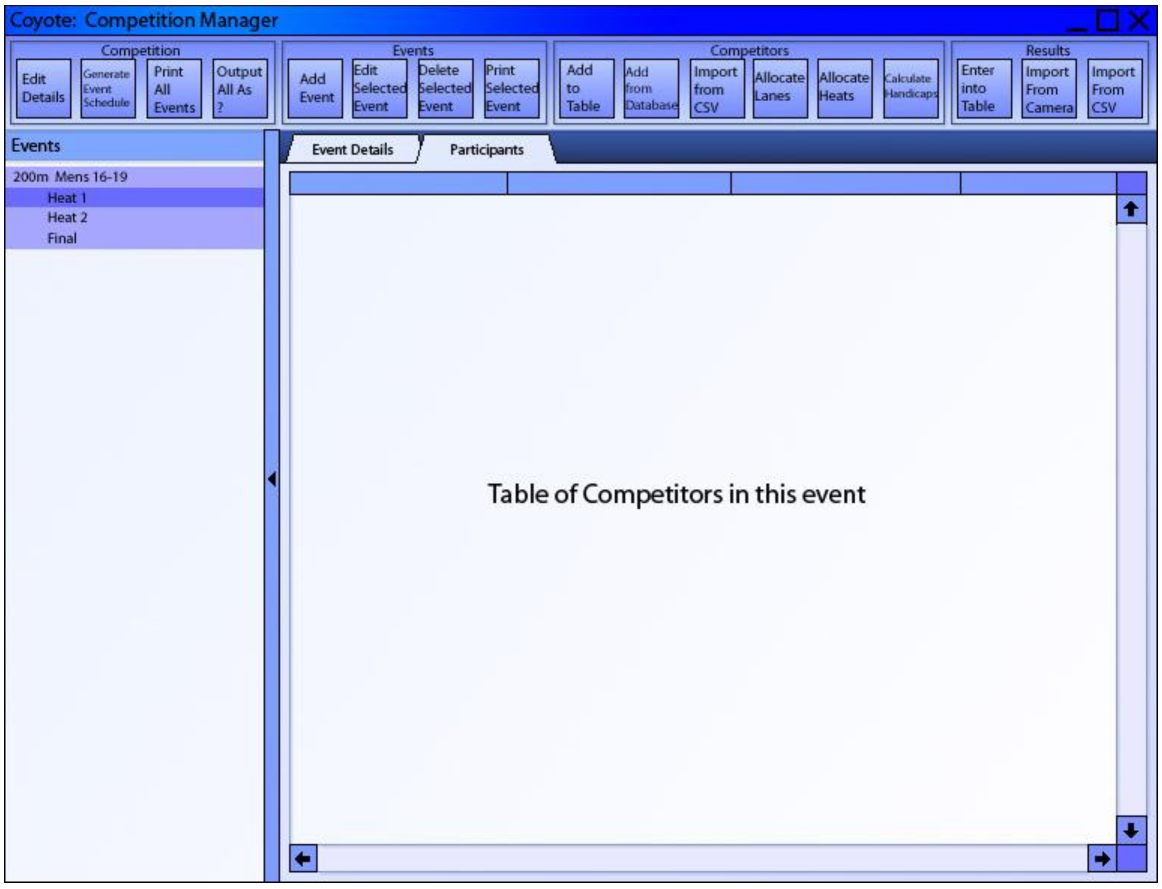
The new competition component allows users to quickly define a new competition, including selecting preset events that it contains based upon the type of competition.

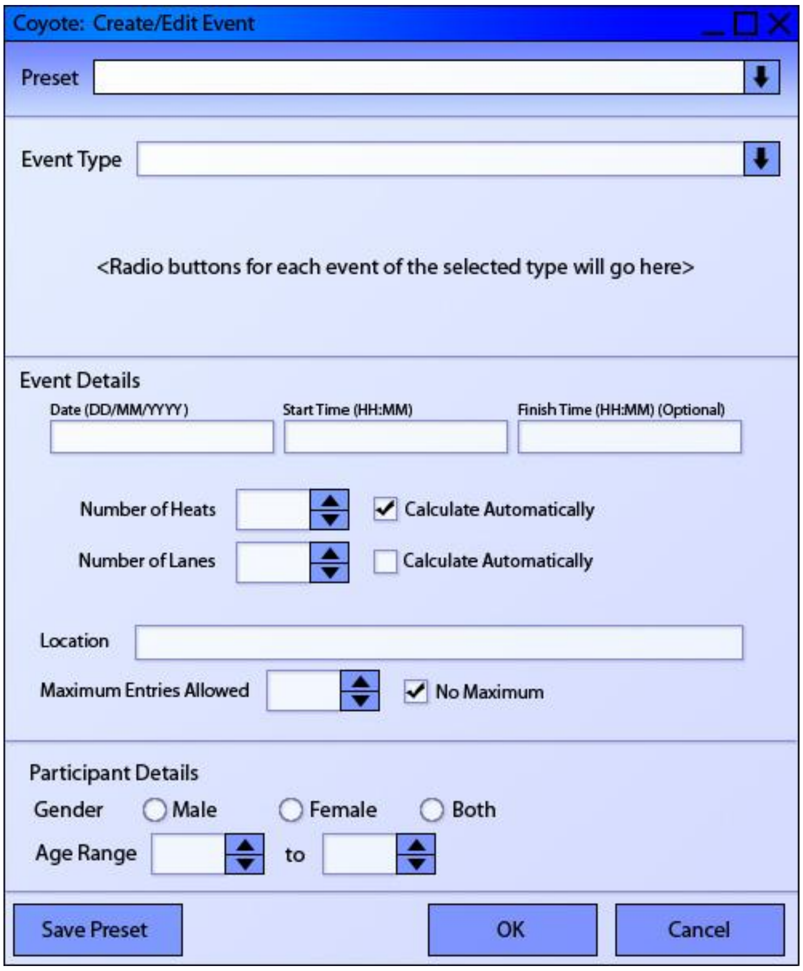
3.5.3 Edit Existing Competition

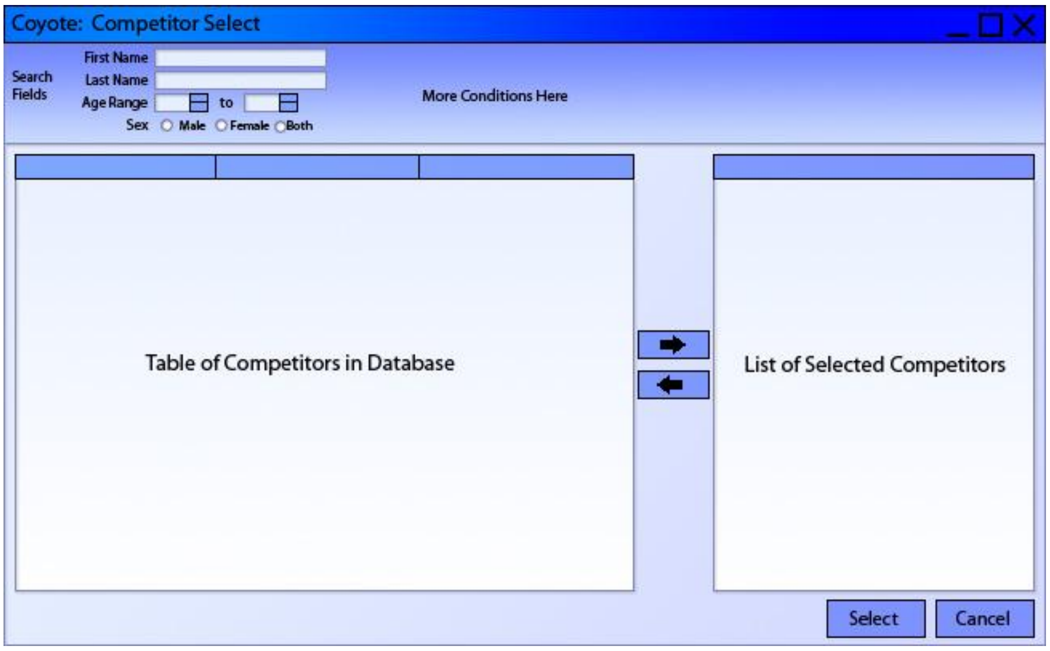


Selecting the option to edit an existing competition allows the user to filter all available competitions to find the one they want.

3.5.4 Work on a Competition

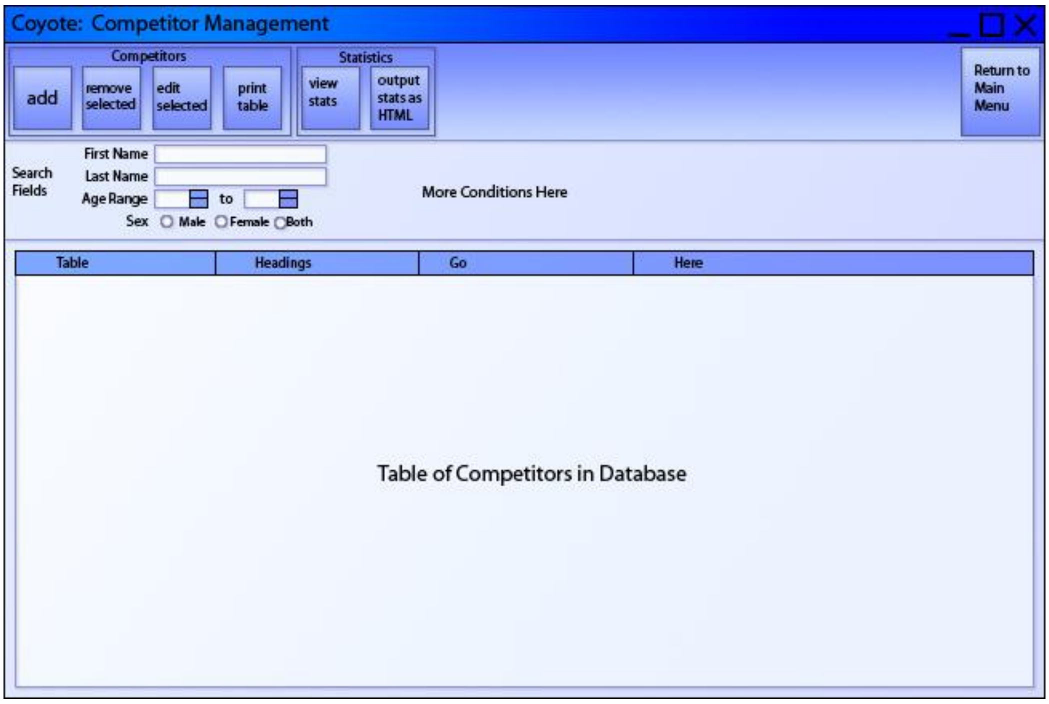


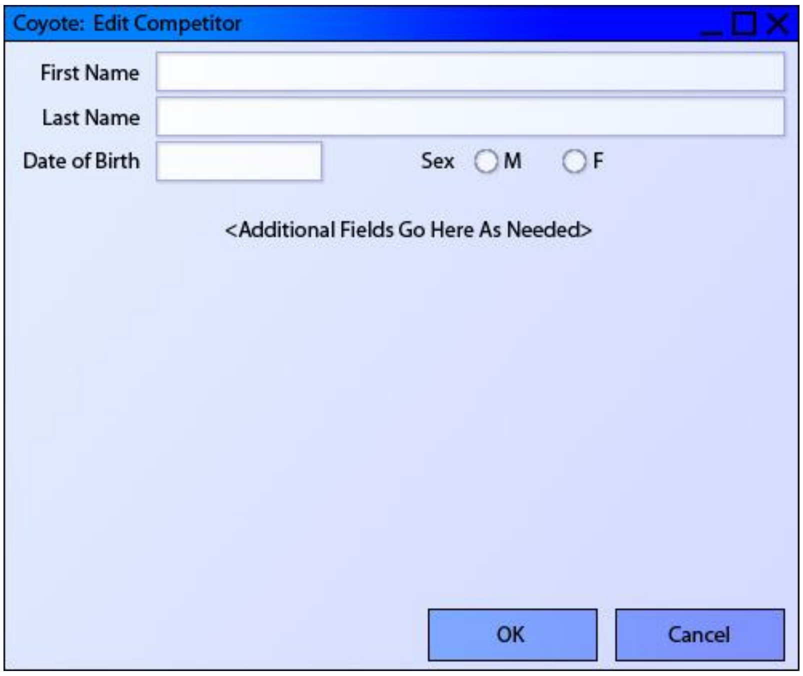




Once a competition has been selected or created, this screen is presented. It allows the user to quickly work on the particular events that it contains. Two modes will be presented for all of the events: one where competitors can be entered into the event and one where the results of the event can be entered. These modes can be switched between using a button in the ribbon strip. The user interface will be updated accordingly when this change occurs. This mode change works on a per event basis rather than competition-wide. New competitors for an event can be entered directly into the table of current competitors (an auto-complete function may be provided). The list of events can be hidden if the user wants to increase screen space for data entry. Clicking add an event or add competitors from database pops up a new window internal to the program as appropriate.

3.5.5 Manage Competitors





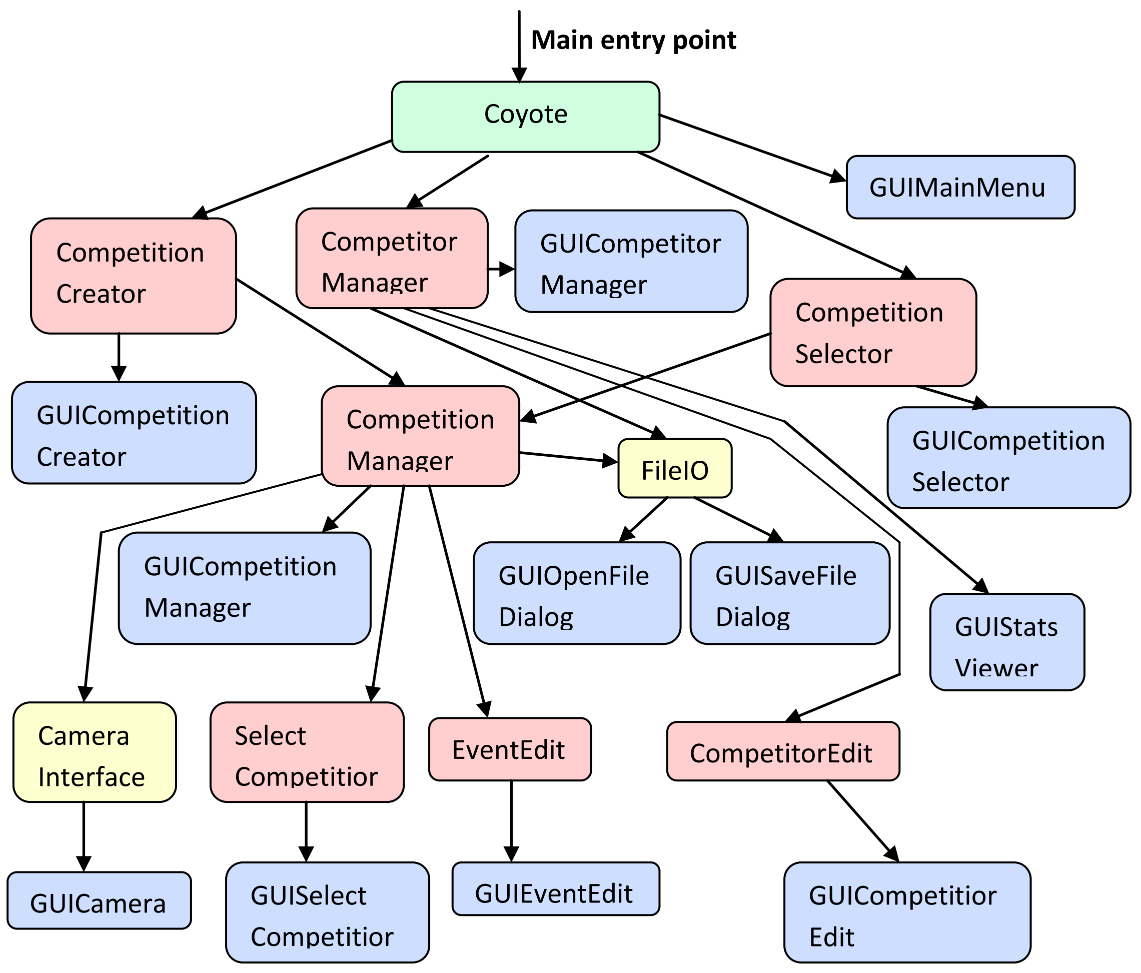
Selecting to add, modify, or view statistics for a competitor opens a new window internal to the program.

3.5.6 View Help

Selecting the help option from the main menu activates the help sub-system, leaving the program running separately, allowing the user to continue working with the program whilst viewing the help information. We also intend to make help available through other means within the GUI, such as pressing the “F1” key.

4. Data Classes

5.1 Overview



Note:

$$\begin{tikzcd} \boxed{A} \arrow[r] & \boxed{B} \end{tikzcd}$$ Means that A calls B.

Colour Key:

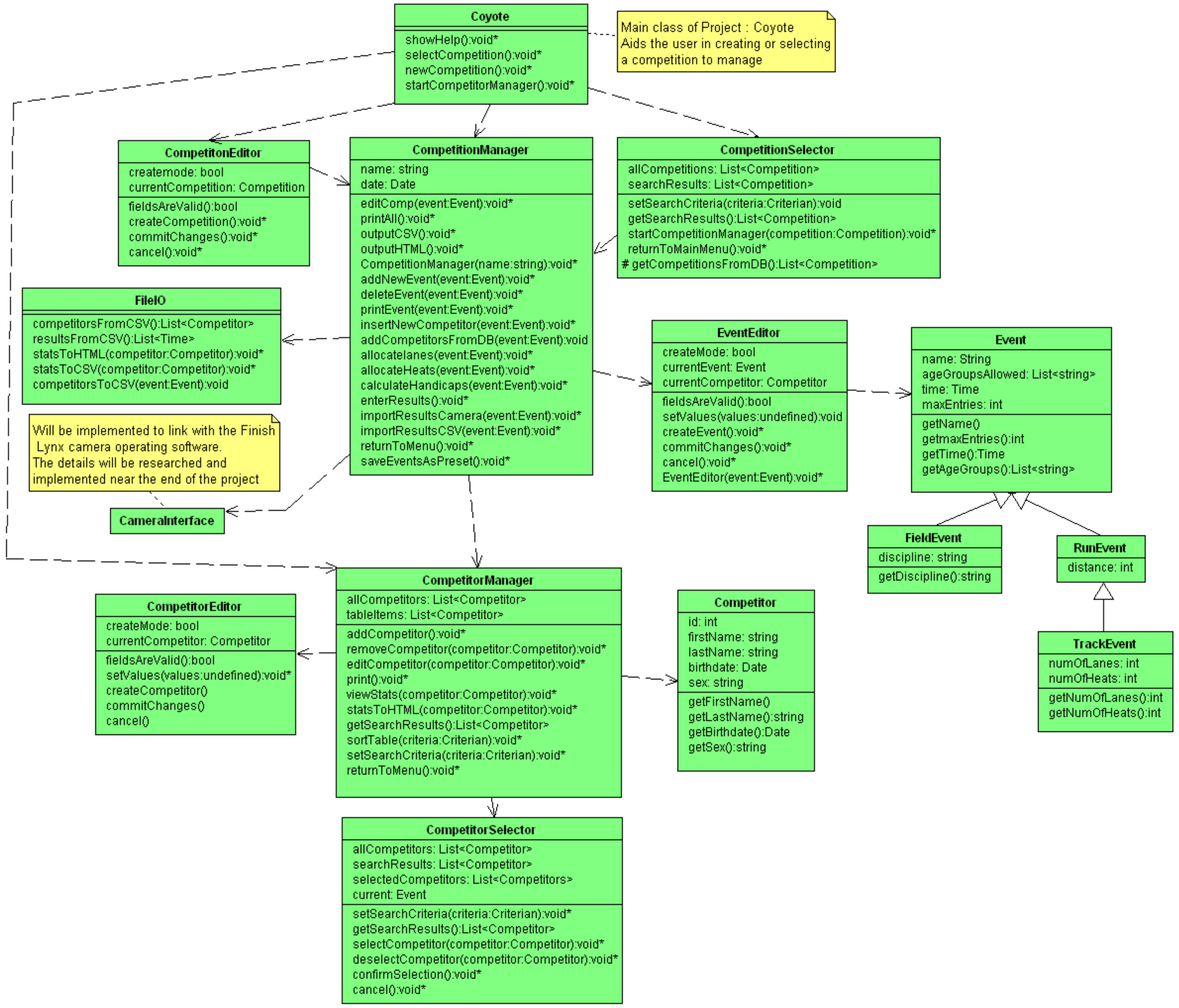
🟩 Main entry point.

🟥 Core program classes that *will* access the database.

🟨 Core program classes that *won't* access the database.

🟦 GUI classes.

5.2 Detailed Class Diagrams


Coyote
showHelp():void*
selectCompetition():void*
newCompetition():void*
startCompetitorManager():void*
Main class of Project : Coyote
Aids the user in creating or selecting
a competition to manage
CompetitonEditor
createmode: bool
currentCompetition: Competition
fieldsAreValid():bool
createCompetition():void*
commitChanges():void*
cancel():void*
CompetitionManager
name: string
date: Date
editComp(event:Event):void*
printAll():void*
outputCSV():void*
outputHTML():void*
CompetitionManager(name:string):void*
addNewEvent(event:Event):void*
deleteEvent(event:Event):void*
printEvent(event:Event):void*
insertNewCompetitor(event:Event):void*
addCompetitorsFromDB(event:Event):void
allocatelanes(event:Event):void*
allocateHeats(event:Event):void*
calculateHandicaps(event:Event):void*
enterResults():void*
importResultsCamera(event:Event):void*
importResultsCSV(event:Event):void*
returnToMenu():void*
saveEventsAsPreset():void*
CompetitionSelector
allCompetitions: List<Competition>
searchResults: List<Competition>
setSearchCriteria(criteria:Criterian):void
getSearchResults():List<Competition>
startCompetitionManager(competition:Competition):void*
returnToMainMenu():void*
# getCompetitionsFromDB():List<Competition>
FileIO
competitorsFromCSV():List<Competitor>
resultsFromCSV():List<Time>
statsToHTML(competitor:Competitor):void*
statsToCSV(competitor:Competitor):void*
competitorsToCSV(event:Event):void
Will be implemented to link with the Finish
Lynx camera operating software.
The details will be researched and
implemented near the end of the project
CameraInterface
EventEditor
createMode: bool
currentEvent: Event
currentCompetitor: Competitor
fieldsAreValid():bool
setValues(values:undefined):void
createEvent():void*
commitChanges():void*
cancel():void*
EventEditor(event:Event):void*
Event
name: String
ageGroupsAllowed: List<string>
time: Time
maxEntries: int
getName()
getmaxEntries():int
getTime():Time
getAgeGroups():List<string>
FieldEvent
discipline: string
getDiscipline():string
RunEvent
distance: int
TrackEvent
numOfLanes: int
numOfHeats: int
getNumOfLanes():int
getNumOfHeats():int
CompetitorEditor
createMode: bool
currentCompetitor: Competitor
fieldsAreValid():bool
setValues(values:undefined):void*
createCompetitor()
commitChanges()
cancel()
CompetitorManager
allCompetitors: List<Competitor>
tableItems: List<Competitor>
addCompetitor():void*
removeCompetitor(competitor:Competitor):void*
editCompetitor(competitor:Competitor):void*
print():void*
viewStats(competitor:Competitor):void*
statsToHTML(competitor:Competitor):void*
getSearchResults():List<Competitor>
sortTable(criteria:Criterian):void*
setSearchCriteria(criteria:Criterian):void*
returnToMenu():void*
Competitor
id: int
firstName: string
lastName: string
birthdate: Date
sex: string
getFirstName()
getLastName():string
getBirthdate():Date
getSex():string
CompetitorSelector
allCompetitors: List<Competitor>
searchResults: List<Competitor>
selectedCompetitors: List<Competitors>
current: Event
setSearchCriteria(criteria:Criterian):void*
getSearchResults():List<Competitor>
selectCompetitor(competitor:Competitor):void*
deselectCompetitor(competitor:Competitor):void*
confirmSelection():void*
cancel():void*

The UML diagram above shows the class structure of the core program (i.e. the middle tier of the system sitting between the user interface and the database). It describes each class's public methods, instance variables, and relationships to other core program classes.

Core Program Classes

class Coyote
This is the entry point of the program, and provides the underlying functionality of the Main Menu GUI. From here the rest of the program is accessed.

class CompetionEditor
This part of the program modifies the values associated with a Competition. It is also responsible for creating new competitions. The user interacts with it via the Competition Editior GUI.

class CompetitionSelector
This class is used to search the database for competitions, select one and load it into a CompetitionManager. The user interacts with it via the Competition Selector GUI.

class CompetitionManager
CompetitionManager is the largest class in the program and it is used to add events to a completion, add participants and results to those events. It also provides a host of tools to help organise a competition and automatically fill in details. Additionally it can output competition details in a variety of useful formats. The user interacts with it via the Competition Manager GUI.

class SelectCompetitors
The purpose of this class is to provide a mean to select several competitors from the database at once and then add them as participants in an event. It provides tools for searching through all of the competitors in the database. The user interacts with it via the Competitors Select GUI.

class EventEditor
This part of the program modifies the variables associated with an Event. It is also responsible for creating new events. The user interacts with it via the Event Editior GUI.

class CompetitorManager
Competitor manager is used to manage competitors in the database. This includes providing functionality to add and remove competitor from the database. Search the database for existing competitors, then View and edit details and statistics of those competitors. The user interacts with it via the Competitor Manager GUI.

class CompetionEditor
This part of the program does the actual work involved in creating and editing competitors in the database. The user interacts with it via the Competition Editior GUI.

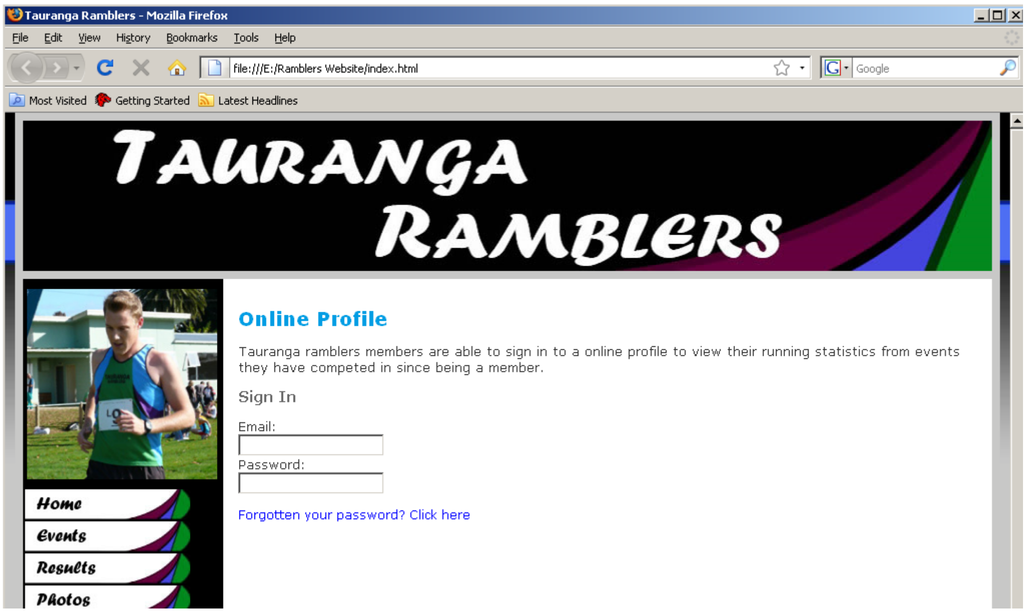
class FileIO
This utility class provides methods for the various file reading and writing functions of the program.

5. Web System

A User that is regularly entering the clubs events can use an online profile to enable them to quickly and easily enter events in the future and view personal statistics. This will be achieved by storing a database online and providing access to it via PHP scripts. The database will be synched with a local version of the system each time the local system gains internet access (usually a short period of time after a competition has completed). PHP scripts are able to quickly access both our MySQL databases and Java programming objects.

The web system will be designed to integrate into the Tauranga Ramblers current website. This means all fonts, colours, buttons and overall appearance will be replicated from the rest of the web system. This will be achieved by linking the web system to the same .css file, as well as using the same button images etc.

The user will be required to log in using an email address and password. See below for a possible solution. This will enable the security of personal information to be maintained, and also ensure all users are unique are legitimate.



Online Profile

Tauranga ramblers members are able to sign in to a online profile to view their running statistics from events they have competed in since being a member.

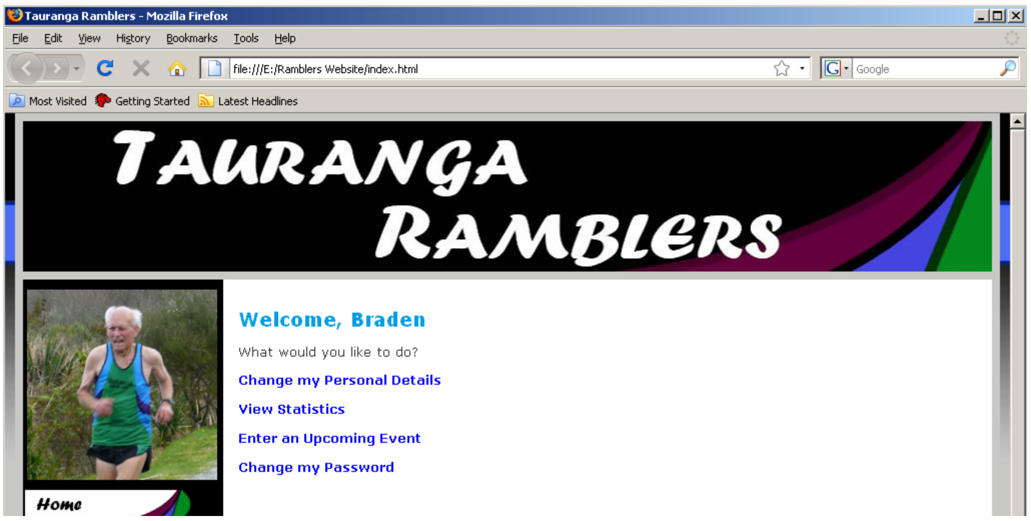
Sign In

Email:

Password:

Forgotten your password? Click here

Once the user has successfully logged in, they will be able to quickly and easily modify personal details or view statistics from previous competitions. This will be achieved by having simple navigation using hyperlinks to the various sections of their profile.


Tauranga Ramblers - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///E:/Ramblers Website/index.html
Most Visited Getting Started Latest Headlines
TAURANGA RAMBLERS
Home

Welcome, Braden

What would you like to do?

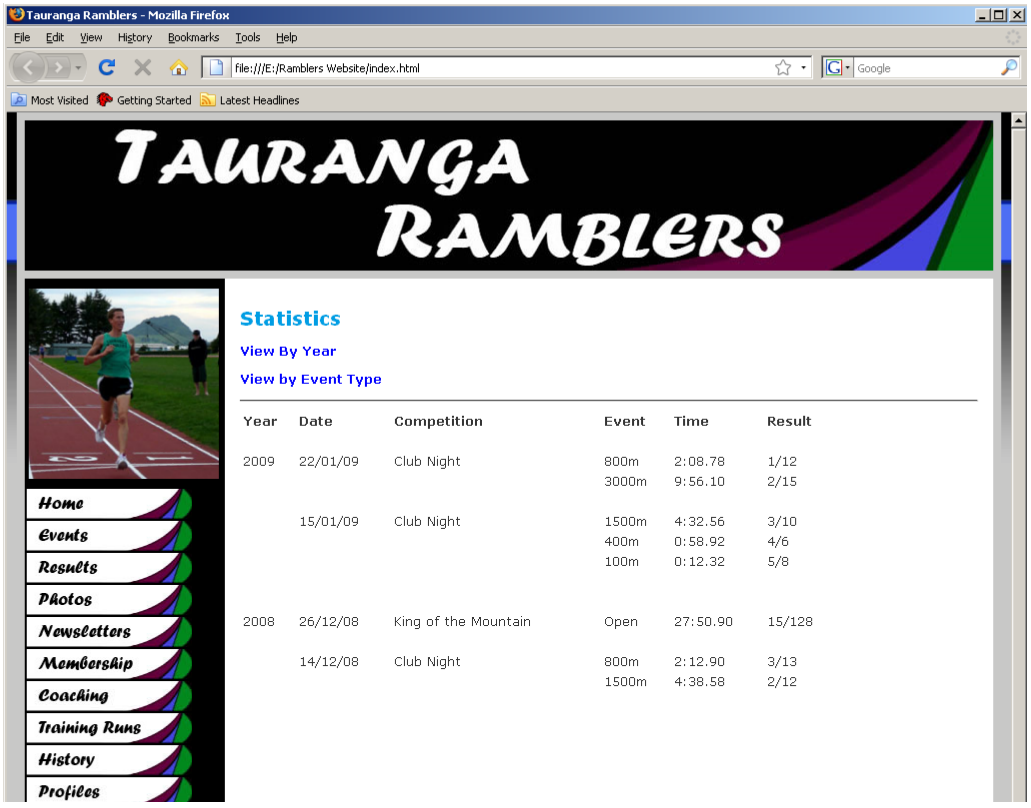
Change my Personal Details

View Statistics

Enter an Upcoming Event

Change my Password

As a member may belong to the club for a very long period of time, the statistics will appear with recent events first. Also, the user will be able to sort the data by event type to enable the competitor to quickly compare event times over a season or multiple seasons.


Tauranga Ramblers - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///E:/Ramblers Website/index.html
Most Visited Getting Started Latest Headlines
TAURANGA RAMBLERS
Home
Events
Results
Photos
Newsletters
Membership
Coaching
Training Runs
History
Profiles

Statistics

View By Year

View by Event Type

Year	Date	Competition	Event	Time	Result
2009	22/01/09	Club Night	800m	2:08.78	1/12
			3000m	9:56.10	2/15
	15/01/09	Club Night	1500m	4:32.56	3/10
			400m	0:58.92	4/6
			100m	0:12.32	5/8
2008	26/12/08	King of the Mountain	Open	27:50.90	15/128
	14/12/08	Club Night	800m	2:12.90	3/13
			1500m	4:38.58	2/12

6. Output

There are a number of possible ways a user will be able to output data into various formats. This will enable the data stored in Project: Coyote to be viewed across multiple platforms and computers.

7.1 Start Lists

The ability to produce start lists is an essential tool in enabling a competition to run smoothly. They inform competitors of lane draws, number of competitors competing, start times, and also help with registration prior to the event starting. The user will be able to select from a range of output formats such csv, HTML and pdf formats. An example output will be formatted as below:

Men's 100M			
Heat 1			
1	Ben Potter	Counties	SM
2	Adam Somerville	CMA	SM
3	Joseph Millar	Tauranga	M19
4	Stephen Buckley	CMA	SM
5	Andrew Catlin	Hamilton City	M19
6	Jonathon Morton	Tauranga	M19
Heat 2			
1	Nicolas Vickers	Lake City	M16
2	John Campbell	ACA	M16
3	Ryan Overmayer	Hamilton City	M16
4	Sam Scott	Lake City	M16
5	Alistair Adams	Hamilton City	M16
6	Gavin Jensen	Manawatu	M16
7	Brent Gough	Tauranga	M16
Heat 3			
1	George Gardiner	Tauranga	M45
2	Dave Rondon	Whakatane	M45
3	Daniel Roose	Papamoa	M45
4	Christian Hotta	Lake City	M50
5	Dave Whitehead	Tauranga	M50
6	Brendan Magill	Tauranga	M45

7.2 Results Output

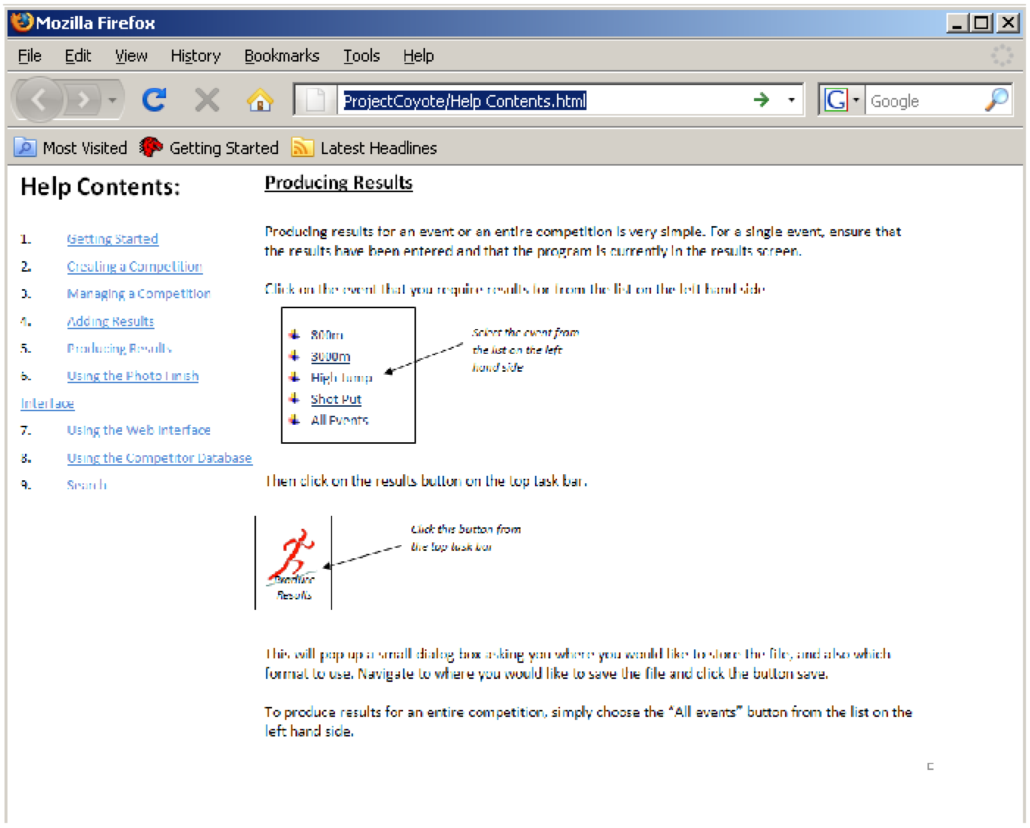
All competitions require a simple and easy method to produce results to inform competitors of their achievements. The user can select to output results for a single event, or for the entire competition into a single file. The user will be able to select from a range of output formats such csv, HTML and pdf formats. An example output will be formatted as below:

Men's 800M				
1	Scott Hilliar	Matamata	M16	2.10.03
2	Ben Rogers	WHAC	M16	2.24.55
3	Andrew Robinson	Mt Maunganui	M19	2.35.54
4	Gavin Smith	Tauranga	M55	2.35.78
5	Malcolm Angell	Tauranga	M45	2.39.25
6	Dave Whitehead	Tauranga	M50	2.40.22

7. Help System

The help system will use an interactive web browser interface. This is to allow an easy method to include screenshots as well as providing a familiar setting for most computer users. It will allow users to search for help criteria, as well as select relevant topics from a menu. The help system is primarily to provide documentation for new or existing users to be able to learn how to use the program.

The help system will be written in a very high level language, to allow computer illiterate users to understand each instruction or explanation. A sample screenshot



Help Contents:

1. Getting Started
2. Creating a Competition
3. Managing a Competition
4. Adding Results
5. Producing Results
6. Using the Photo Finish Interface
7. Using the Web Interface
8. Using the Competitor Database
9. Search

Producing Results

Producing results for an event or an entire competition is very simple. For a single event, ensure that the results have been entered and that the program is currently in the results screen.

Click on the event that you require results for from the list on the left hand side

- 800m
- 3000m
- High Jump
- Shot Put
- All Events

Select the event from the list on the left hand side

Then click on the results button on the top task bar.

Click this button from the top task bar

This will pop up a small dialog box asking you where you would like to store the file, and also which format to use. Navigate to where you would like to save the file and click the button save.

To produce results for an entire competition, simply choose the "All events" button from the list on the left hand side.

Glossary

Competition	Is one or more races at a specific date, time and location
Event	A single competitive contest with multiple competitors usually grouped into age categories or ability
Race	A single competitive contest which involves sprinting/running
Heat	A preliminary sprinting/running race in which the competitor advances to a more important race