



GOLFRECORD

Computing Course Work

Freddie Woods

Analysis	5
Aim	5
Identification of the project	5
Who was the project aimed at?	5
What is a casual golfer?	5
Why casual golfers?	6
Discussions with Clients	6
Mr H	6
Mr G	6
How the Game of Golf Works.	6
Current Score Cards	7
What applications are currently out there?	7
Golf application 1: “Distance Calculator”, creator “Qi Chen”	7
Golf application 2: “Golf Game Book”, creator “GameBook Oy”.	7
Golfer application 4: “Golf Breaks Sunshine”, creator: “Sunshine Golf Breaks Ltd”	8
Golfer application 5: “Golf GPS Range Finder Simple”, creator: “Bryan Thornbury”	8
Evaluation of Research.....	8
User Needs.....	8
Must Be Able To:.....	9
Should Be Able To:	9
Could Be Able To	9
Design.....	10
General Design of the Application	10
Why choose an Object-Oriented approach?	10
Object-Oriented programming	10
The effect on the database	10
Inheritance	10
Why I chose to use Naked Objects.	11
The Main Objects	11
Course:	11
Golfer:	12
Match:	14

Hole:	15
Hole Scores:	16
<i>Hole Scores</i>	16
Possible Further Classes.....	16
Group:	16
Invite:	16
Message:	16
Structure of the database.	16
Entity Relationship diagram with Many to Many	17
Technical Solution	18
Overview	18
Model	19
Objects	19
Properties.....	22
Services	28
Programing Techniques	30
Specifics.....	30
Server	36
Client	38
Generic UI	38
Customisation	38
Database	39
Entity Framework.....	39
Manual Database Mapping.....	40
Database Customization	41
Initializer.....	42
Testing.....	44
Test 1: Creating a New Account.....	44
What the user can and can't do once logged in for the first time.....	44
How to create an account.....	44
Private Accounts	45
Test 2: Enriching the User's Profile	45
Adding Friends	46

Joining a Group	47
Sending Messages	50
Test 3: Creating a Match	51
Browsing Courses	51
Sending The invitations	52
Creating the match	52
Too many players want to join	54
Playing the first hole.	55
Test 4: Finishing a Match	58
Looking through match history	58
Adding a description of how the match went.	59
Evaluation	60
Comparing my project against User Needs	60
Ease of Use	60
Areas of Improvement	60
Conclusion	61
Appendix – All My Code	62
Client	62
Config.json	62
Database	62
GolfRecordDbContext	62
GolfRecordDbInitializer	63
Server	71
NakedObjectsRunsetting	71
Model	73
Club manger	73
Club manager Authoriser	73
Course	75
CourseAuthoriser	77
CourseServices	78
DefaultAuthoriser	78
Enums	79
Facility	79

FacilityAuthoriser	79
FriendInvitation.....	80
Golfer	80
GolferServices	86
Group	87
GroupAuthoriser	88
GroupInvitation.....	89
GroupMessage	89
Hole	90
HoleAuthoriser	92
HoleScoreAbstract	93
HoleServices	93
Invitation	93
Match	94
MatchAuthoriser	97
MatchInvitation.....	98
MatchPlay	98
MatchPlayAuthoriser	100
MatchplayHoleScore	101
MatchServices	102
Message	103
MessageAuthoriser	103
Player	104
PlayerAuthoriser	105
PlayerMessages.....	107
RequestToJoin	107
Stableford.....	107
StablefordAuthoriser	110
StablefordScores	111
Strokeplay	112
StrokeplayAuthoriser	113
StrokeplayScores.....	114
UnregisteredGolfer	115

Analysis

Aim

The aim of my computing project was to create a Golf application which could record scores from matches according to various marking schemes, i.e Stableford, Matchplay and Strokeplay. The golf application was intended for casual players who just wanted to have fun, record their matches and see whether their friends want to play a match or are currently in a match.

Identification of the project

Currently at Stowe Golf Club golfers record their scores on physical score cards. These can be a problem because they are easily lost, damaged and can't cope with rain. In contrast, an online version will have the advantage that a lot more data can be stored, and an algorithm can be used to calculate the score after the handicap and stroke index have been considered. This application will run on a phone, so the scores can be filled during the match. Mr. H, who is a professional golf coach at Stowe Golf Course, agreed with this motion. He acted as my primary client for this project.

As a casual golfer a main reason to play the sport is the community that comes along with it. To help the community thrive, a messaging service and a "Friends" system were incorporated. The application was designed so that a player should be able to see all the other golfers and, if were in a match the user could see which club they were playing at, the current score in that match and who they were playing against.

Another inconvenient aspect of the old card-based system is the score calculation. Some golf scoring systems can be confusing which can lead to mistakes. With an online computer-based system this the application accurately calculates the score for each hole, thereby speeding up the game and allowing more people to play. Finally, the new system can help golfers by providing easy access to details of the clubs with pictures of the course, holes, descriptions of the clubhouses and a website link for each course. The user will thus be able to see how many holes there are, the length of each hole. The final aspect as requested by Mr H is a match history system whereby a user can go through all their past matches to see which holes they struggle on, leave a comment to say what went well or, what when wrong in the match, and note the weather conditions on the day of the match. Being able to go through past matches helps each player to improve.

Who was the project aimed at?

The project was aimed at casual golfers rather than clubs running tournaments. This is because there is a larger gap in the market for casual golfers. At tournament level, there are additional rules required to verify of the results, one of which is that each golfer must score the other golfers but not themselves; therefore more than one scorecard is passed onto the official. This means that a separate scoring system would be required for tournament golf.

What is a casual golfer?

A casual golfer is a golfer who wants to play golf socially but not in a competition. This means they don't need validation of their results as they would in a tournament. A casual golfer will be playing at any time they want if the course is open.

Why casual golfers?

I chose to aim my project at casual golfers because these golfers like to analyse their results from passed matches. They also like to explore new clubs and have access to a database of information on each club. When casual golfers play a match with their friends they do not require validation to the extent of a tournament.

Discussions with Clients

Mr H

As mentioned in the introduction, Mr H is a professional golf coach at Stowe Golf Club. He teaches many golfers how to improve their game, plays tournaments and goes on golfing holidays. He loves to play casually at different clubs but also teaches many golfers and so can get their opinion on the application.

In a conversation with Mr H, I asked what was good about the card-based system and what was wrong with it. He responded: "The old system was very quick and easy to use. At a tournament you just write the score and a tournament official would calculate the actual score based on the match type. In a casual game, the card-based system can be a pain as it takes a while to calculate the score. Another aspect of the card-based system is that everything is on it; the hole name, the yards for each hole and the stroke index. I still believe there is area to improve on the old card-based system. Another problem with the cards are that they are easy to lose and to keep an accurate history of your matches where you can see which hole you struggle on is very hard. This is in which an electronic system will improve the card-based system."

Mr G

Mr G is a teacher at Stowe but regularly helps with the Stowe Golf team. He's very enthusiastic about the sport and will often spend his afternoons and weekends on the golf course. Mr G considers himself a casual golfer since he doesn't take part in any tournaments.

While talking to Mr G, he mentioned that a big part of the game occurs after the round of golf when you are in the club house talking about what went wrong on each hole or who played better. Having a record of all the past matches with who won and how each hole went is a great way to enhance this aspect of the sport.

How the Game of Golf Works.

A description of how the main match types is provided in the Design section of this report with a data flow diagram for each one. For small clubs like the Stowe Golf Club when you want to play a game you just turn up. If you are a member of that club, you can start straight away; but if you are not a member you must first pay for your round of golf. On the course there are certain standards that each golfer must live up to. For example, must first; sweep the bunker you found yourself playing a shot from and you must only tee off once the previous players are out of your range or finished on the hole. When playing your round, you must use the score card provided by the club to record the scores. You can either write down the raw score of the number of strokes or the score based on the match type you are playing. With certain match types there are notations that get written on the card, for example for match play the winner of the hole has a circle around their score. This will be discussed in the Design section under Match Types.

Current Score Cards

Below are two photos for the Stowe Golf Club Scorecard. One of these scorecards has been filled in for a match play match type.

Hole	Player's Name	White Tees	Yellow Tees	Par	Stroke Index	Player's Score
1	John Kelly	506	484	5	12	458
2	Phonemore	412	403	4	4	378
3	John's Wife	135	131	3	18	116
4	Phonemore's Son	401	397	4	2	322
5	Donatello	279	274	4	16	247
6	The Book	206	191	3	10	173
7	John's Wife	530	514	5	8	497
8	Phonemore's Son	157	153	3	14	153
9	The Book	425	400	4	6	301
3051 2960 35 OUT						2945 35 OUT
ENJOY YOUR DAY - PLEASE REPAIR PITCHMARKS						
10	The Book	400	384	4	11	355
11	Paul O'Brien	368	362	4	7	336
12	John's Wife	507	488	5	13	406
13	John's Wife	410	400	4	3	369
14	Amel Carter	401	391	4	1	365
15	Kenneth	368	363	4	15	354
16	John's Wife	149	145	3	17	135
17	Phonemore	390	384	4	9	365
18	Paul O'Brien	583	563	5	5	406
3596 3510 37 IN						3081 37 IN
3051 2960 35 OUT						2945 35 OUT
6647 6476 72 TOTAL						5706 72 TOTAL
HANDICAP						NETT

Hole	Player's Name	White Tees	Yellow Tees	Par	Stroke Index	Player's Score
1	John Kelly	506	484	5	12	458
2	Phonemore	412	403	4	4	378
3	John's Wife	135	131	3	18	116
4	Phonemore's Son	401	397	4	2	322
5	Donatello	279	274	4	16	247
6	The Book	206	191	3	10	173
7	John's Wife	530	514	5	8	497
8	Phonemore's Son	157	153	3	14	153
9	The Book	425	400	4	6	301
3051 2960 35 OUT						2945 35 OUT
ENJOY YOUR DAY - PLEASE REPAIR PITCHMARKS						
10	The Book	400	384	4	11	355
11	Paul O'Brien	368	362	4	7	336
12	John's Wife	507	488	5	13	406
13	John's Wife	410	400	4	3	369
14	Amel Carter	401	391	4	1	365
15	Kenneth	368	363	4	15	354
16	John's Wife	149	145	3	17	135
17	Phonemore	390	384	4	9	365
18	Paul O'Brien	583	563	5	5	406
3596 3510 37 IN						3081 37 IN
3051 2960 35 OUT						2945 35 OUT
6647 6476 72 TOTAL						5706 72 TOTAL
HANDICAP						NETT

Each column of the score card will be described in the Design section under Scorecard, but an area to point out is the red sentence in the middle of the card: "Enjoy your day – please repair pitch marks". Although this is a very small part of the scorecard, it's still an intriguing aspect to add to my version.

What applications are currently out there?

As part of my research for this project, I went on to the app store, downloaded and tested multiple apps to see what ideas they have and what makes a strong application from a user perspective. Below are a few that influenced my design.

Golf application 1: "Distance Calculator", creator "Qi Chen".

Although this isn't a golfing application as previously mentioned in the introduction, it proposes a very interesting design idea. It works first by taking a photo of the golf flag. You then crop the photo so that it's the height of the bottom of the flag to the top. The application uses the size of the flag to calculate the distance to the hole. This application requires a calibration to set up the correct distance and I'm unsure its accuracy, but it is still an interesting idea and one to take into consideration when thinking of what to add to really make my product stand out.

Golf application 2: "Golf Game Book", creator "GameBook Oy".

This is similar to the design of project as I originally conceived and contains much of what I initially thought my project would incorporate. It contains a strong social media section where you can post photos, videos, golf tips and much more. This is an advanced application, but it might be interesting to add a few 5-minute golfing videos for people to improve their game. This application also includes an ability to see which friends are on a course practising or playing in a match or tournament. This a very

beneficial addition to the application. The scoring section of the game is very detailed with thousands of clubs and courses in its database, each with various amounts of data. Upon selecting a course, you have a choice of which type of match you are playing in from the extensive list. You then add the players including members and temporary players who don't have an account. Finally, you add the scores for each player at each hole and when you finish the game it shows the winner. The only downside is the lack of data on matches and on the club. This is something that I aimed to improve on when creating my application.

Golfer application 3: "Golf Weather", creator "GolfWeather".

This application does exactly what the name suggests. It tells you the weather at a chosen course for a given week. It provides the weather for 6:00, 9:00, 12:00, 15:00 and 18:00. The information per time shown is: the temperature, wind speed and whether there is going to be rain. It also tells you the time of sunrise and sunset. These are all important sets of information for deciding whether to play a game. This could have been an interesting concept for the project although I was unsure how I would be able to keep it up to date without doing it manually. There is also a function called "course near me" which takes your location and finds all the courses near that location. This is good for players choosing where to play if they are new to the area.

Golfer application 4: "Golf Breaks Sunshine", creator: "Sunshine Golf Breaks Ltd"

This travel app has a selection of pre-packaged holidays which vary in duration, number of people and rounds of golf they want to play. It also has a selection of green fees at famous clubs when using buggies or walking. This could be added to each of the courses by their managers. This would however also require a payment system to keep the users data secure and ensure the authenticity of the application.

Golfer application 5: "Golf GPS Range Finder Simple", creator: "Bryan Thornbury"

This range finder works differently to the first one but is still interesting. It works by presenting a Google earth view. From this view you can see the golf course and your current position. The next step is to press a position on the map and it will draw the line from your position to where you pressed and calculates the length in yards. You can then press another position and it will calculate the distance from the old position to the new. This can be done as many times as desired and in the top right it shows the total distance. This is perfect for recording the length of multiple shots. When finished you can press the eraser and it will delete all the lines you have made. Adding a Google earth view of the course is great for golfers scouting out the hole they are playing on. To improve on this app, the fair way, rough, bunkers and green could be marked out to provide a clearer view of the hole.

Evaluation of Research

From my research into current applications, I learnt there is a broad range on concepts which could be incorporated into the design: a range finder, a weather system, a photos and video feed, packaged holidays per course and a bird's eye view of the courses.

User Needs

This section sets out the various "must haves" and "nice to haves" for this application

Must Be Able To:

- Create a pro file for a golfer where they can add their information
- Create a match
- A score card for each match
- Add other golfers to the match
- Add a score for each hole in the match
- Work out the winner of the match
- Go through past matches
- Add a friend
- Send a friend a message
- Browse through courses to decide which one to play at
- Be able to see whether friends are in a match

Should Be Able To:

- Create a group for golfers who regularly play at the same course or together
- Make an account private if a user doesn't want to share certain details
- Go back through the scores of a match and change a score
- Send someone who is not a friend a message
- Follow a website link to visit the website for that course
- Look at a bird's eye view of the course and each hole
- Annotate the score to show elements of that Match Type

Could Be Able To

- Look up the current weather status of a course.
- Link to package holidays for courses.
- Provide a note on the match provided by the course.
- Incorporate a range finder to see how far you are from the flag.
- Schedule more than one match.
- Exhibit a calendar showing big events and matches.
- Include an events feed saying what tournaments are where with live score feed.
- Add a membership section to your profile.
- Leave a description of how the match went.

Design

General Design of the Application

The overall product was designed as an application that could be accessed on any device but primarily on a phone which then can be used while playing golf as mentioned in the analysis. The application should be as simple as possible and therefore shall consist as few buttons as possible.

I decided to use a few external sources to aid my project but more importantly to allow me to focus on the business side of the application. This meant I had more time to work on the capability of my program and provide the user with the best possible product given the time constraints. I used an Object-Oriented approach as discussed below. The application was developed on one device using different browsers to simulate the different users. I employ a software called Naked Objects, the purpose of which is set out below.

The main section of the application was the homepage which contained all the functionality. It was split up into separate categories: Matches, Golfers and Courses. I chose these since they are the most important and because each of the user needs from the Analysis can be placed into one of the categories. Keeping a rigid structure for the software was important for writing the code but is also helpful for the user.

Why choose an Object-Oriented approach?

Object-Oriented programming

Object-Oriented programming is an important principle of computer science which allows for the use of many other concepts for example encapsulation, inheritance and polymorphism. It also works with functional programming. In Object Oriented Programming the world is viewed as “Objects”. These objects can either be physical, like a Golfer, or abstract, like a data structure. This was important for my program where there were many physical objects, which need to be incorporated, for example course, hole and golfer, but there were also multiple abstract objects such as, match, score and friends. Each of the objects can interact with each other allowing them to be treated as if they were physical. Each of these objects has properties and methods. A property is a value assigned to an object, for example with golfer as an abstract, this could be the golfer’s name or handicap. An action is a function that the object can perform. This could include adding a value to joining a match.

The effect on the database

In the database objects are treated as tables. Each object has an ID which acts as a primary key. Objects also have relationships meaning that the database will be a relational database. The relations within my database are shown in an entity relationship diagram under structure of the database.

Inheritance

Inheritance is a concept in object-oriented program where there is one super class beneath which there are one or more subclasses. For example Strokeplay is a subclass of the Match superclass. In inheritance, all the knowledge in the superclass can be accessed by a subclass but not the other way around.

Inheritance is useful If two objects have the same properties but different actions. Inheritance was used a lot throughout my program for matches and for golfers where there were different types of accounts.

Why I chose to use Naked Objects.

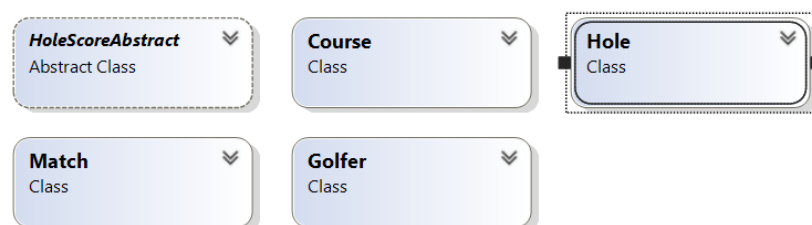
Naked Objects is an architecture used in the business world for the creation of various projects. Naked Objects is a framework consisting of 4 solutions: the client, the server, the model and the database. Each of the solutions is discussed more in depth under the Technical Solution. An object in Naked Objects can be in one of two states: transient and persisted. When an object is transient it hasn't been saved to the database, but when it's been persisted the object will be in the database. An object cannot go from persisted to transient but can go from transient to persisted.

Naked Object provides several files which create the initial interface for you. These files could otherwise take a long time to create and, in a time pressured project most of the time is centred towards the main functionality of the program and having a foundation to build proven very useful. This allowed me to change fonts, to authorize values and classes, and change the colour of objects along with multiple different other customisations. Being able to change the colour of objects is very useful for the user as they can easily differentiate between objects and know what they are doing. The interface also has a menu made for it. Allowing for easy navigation between objects and pages.

Naked Objects automatically maps the database for you although some manual mapping is still required. The use of the automatic data base enabled more time to be spent on the business logic and therefore improved the overall product. Finally, Naked Objects comes with a very strong RESTful API.

The Main Objects

The I objects listed below are the main objects to complete the user's needs. Under each heading is the list of properties that object will have along with their actions which are visible to the user. In this section I describe ICollection and Enums. An ICollection is similar to an array of the same Objects which is portrayed in Naked Objects as a list. This creates the One to Many relationships. A Enum is a property which can only be certain values defined by the creator.



Course:

The Course object

There are no actions on the course but the only type of user who will be able to edit the course will be the manager for that course. This is achieved by authorization defined under Authentication and Authorization A course manager will be a subclass of Golfer. From the course as you can navigate to all the holes by opening the ICollection. To create a new course as a club you will need to speak to a

systems manager who will create the Club Manager and then an empty course will be created to which the Club manager can add the values.

Properties

Properties	Type	Visible to player?	Editable by player	Visible to Club Manager	Editable by Club Manager	Validation	Optionally
ID	Int	No	No	No	No	No	No
Course Name	String	Yes	No	Yes	Yes	No	No
Location	String	Yes	No	Yes	Yes	No	No
Address	String	Yes	No	Yes	Yes	Length	No
Par	Int	Yes	No	Yes	Yes	No	Yes
Club Manager	Golfer	Yes	No	Yes	No	No	No
Website Link	String	Yes	No	Yes	Yes	No	Yes
Yardage	Int	Yes	No	Yes	Yes	No	Yes
Phone Number	Int	Yes	No	Yes	Yes	Length	No
Photo	File Attachment	Yes	No	Yes	Yes	No	No
Holes	Hole collection	Yes	No	Yes	Yes	No	No
Facilities	Enum collection	Yes	No	Yes	Yes	No	No
Course Description	String	Yes	No	Yes	Yes	No	Yes

Golfer:

The Golfer object

A golfer is the account that you as a user will create when joining the application. There are two types of Golfer, Club Manager and Player. Club Manager accounts can only be set up by a systems manager therefore as a new user you will be required to set up as a Player. An important aspect to include is that only when the Username from login matches the Username of that golfer may the object be edited. The only two values which are not editable to the user are Username and Position.

Subclasses

Club Manager

A club manager is a golfer that belongs to a club. This means that they have a club property that is visible to everyone. I created a club manager so that there can be authorization to prevent any type of golfer editing a course.

Player

A player is the casual golfer. This means that most of the users on the system will be Players. A player also has an ICollection of Favourite Courses which is not available to Club Managers. This gives the user quick access to their best courses.

Unregistered Golfer

An unregistered Golfer will be created when adding golfers to a match. These are golfers who have not created an account but are playing with a friend who has an account. All their details will be empty except name, gender and handicap as these are required for calculating scores and identifying the golfer.

Properties and Actions

Each of these Properties and Action are available to both sub-classes.

Properties	Type	Validation	Optional
ID	int	No	No
Full Name	String	No	No
Handicap	int	No	No
Mobile	string	Length	Yes
Gender	Enum	No	No
Position	Enum	No	No
Username	string	No	No
Private Account	bool	No	No
Friends	Golfer (Collection)	Can't have same golfer twice	Yes
My Matches	Match (Collection)	Can't have same match twice	Yes
Action	Input	Validation	Is Visible
Add Friend	Golfer	Golfer isn't already a friend	To all other golfers apart from himself
Create New Match	MatchType (enum)	Date is after today	Only to that Golfer

Match:

The Match Object

Match is the super class for the other match types it contains the collection. Since it is the most important part of the project it needs to be simple to use. To make a Match easy to create the golfer who is creating the match they will be automatically added to the Match. How each of the separate match types work is explained below.

How Handicap and Stroke Index work:

Each golfer has a handicap and each hole has a stroke index. The stroke index is the order of holes rated from most difficult to easiest so on a 9-hole course, 1 would be the hardest and 9 the easiest. If the Golfer has a handicap of 1, then on the hardest hole according to the stroke index (1), they get 1 taken off the amount of shots taken for that hole. If the Golfer has a handicap of 10 for a 9-hole course they get one shot, taken off each hole except the hardest from which and 2 shots are subtracted.

Subclasses of Match

Match Play

This is the most common of match types used for tournaments. It is a two-player match where you compete for winning each hole. The Golfer who wins the most holes wins the match. To win a hole you need to have taken the least amount of shots on that hole.

Stroke Play

This is the simplest of match types as you just record the amount of shots taken for each hole add them up at the end and then subtract each Golfer's handicap from their overall score. The winner is the golfer with the fewest number of shots taken. This match type is for a 4-player match.

Stableford

Stableford is the only match type that I will be using in which the par for a hole has an effect. It is a 4-player match. Like the other matchtypes the handicap takes shots off the final score for each hole based on the stroke index. For each hole if the golfer score after the handicap been taken into account is the same as the par for that hole they score 2 points for that hole. If the golfer takes one fewer shot than par they score 3 points. If they score 1 over par they get 1 point. As soon as a golfer can't score a point they pick up the ball and can move to the next hole. The highest number of points a golfer can score is 6 which corresponds to 4 shots under the par.

Properties and Actions

Property	Type	Validation	Optional
Match Name	String	No	No
Match ID	Int	No	No
Course ID	int	No	No
Course	Course	No	No
Match Creator	Golfer	No	No

Date of Match	DateTime	Must be after today	No
Match type	Enum	No	No
Winner	Golfer	Invisible and not editable until end where is visible	No
Golfers	Golfer (collection)	Not more than match type allows	No
HoleScores	Holescore (collection)	Can't add a negative score	No

Action	Input Type	Validation
Add golfer	Golfer	Hide when enough golfers in the match
Add Score	Int	Can't add a negative score

Hole:

The Hole object:

The Hole object is the individual hole for each course. Since some courses have a different number of Holes, usually 9 or 18, a match has a collection of 9 or 18 holes. The holes are only editable by the course's course manager. In addition, each course has different values to other courses, for example, one course might have white, yellow and red yards and another only white and red. To prevent there being empty values portrayed to the user I use authorization to make them invisible when empty.

Properties

Property	Type	Validation	Optional
Hole Number	Int	Can't be a repeated number	No
Stroke Index	Int	Can't be a repeated number	No
Par	Int	No	No
Photo	File Attachment	No	Yes
Yardage (all types)	Int	No	No

Hole Scores:

Hole Scores

This is the score card portrayed to the user which is visible within the match as an Icollection. The score card will portray their raw and actual score to the golfer and their total score. Since there are only two types of properties there is no table below.

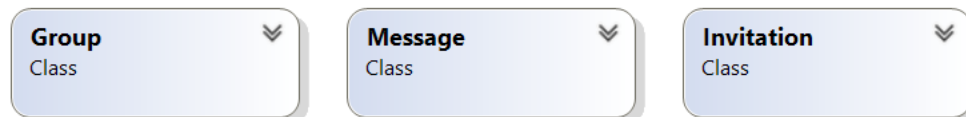
Subclasses of Hole Scores

Since the 3 match types have a different scoring system each of them will have their own personalized score card. Match Play is only a two-player match with a match a hole winner. This means that that in the sub class "Matchplay scores" there is Golfer A actual, raw and total score. Since strokeplay is a tally of scores Total score and actual score will be the same therefore I chose to only have actual score.

Finally, stableford is a has everything, a raw score, actual score and total score.

Possible Further Classes

The possible further classes are the classes required to make the optional user needs.



Group:

A group is a collection of golfers with a group owner and a collection of members. This allows quick access to your friends or to people you play with on a regular basis, there can also be a collection of messages that everyone in the group can see. Only the group owner will be able to edit the group and send invites and all invites sent from other golfers requesting to join be will sent to the group owner.

Invite:

To prevent golfers automatically joining a group, match or becoming friends even if they don't want to an Invite Object which appears on the Golfer object. Each type of invite has the invite type (an enum) as the title, and they are each separate colours. An action also appears as an option where a golfer can accept or decline the invite. Once the invite has either been accepted or declined it will be deleted. To cope with all the different types of invites I can use inheritance and put the invite type, sender, receiver and ID in the superclass and the other values in the subclass.

Message:

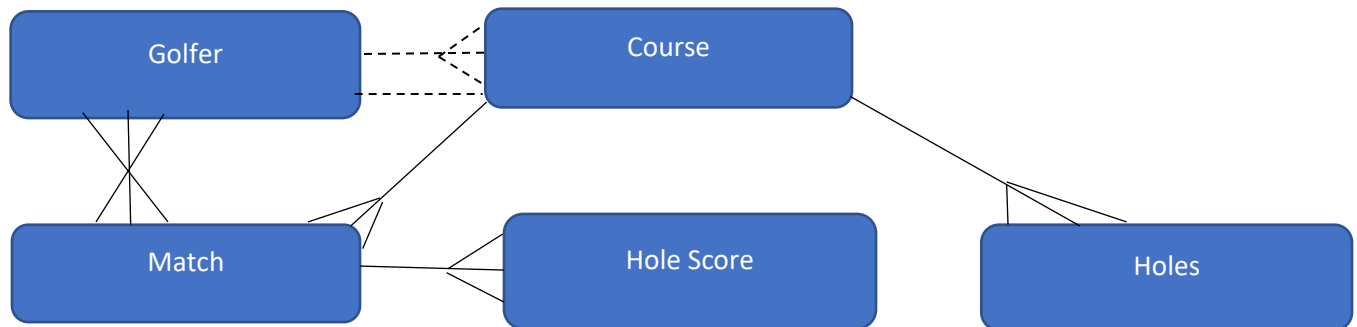
A message is an object with a string, sender, receiver and two methods. The methods will be respond and delete. Responding to a message does not delete the message. It appears as a collection on the golfer or on the group. Only group members are able to see the group message and only the intended golfer is able to see private messages.

Structure of the database.

With Naked Objects the database is automatic. However, there is still a need for manual mapping in parts and the database can be personalised. The figure below shows the entity relationship diagram for the main objects. Since in practice, the database will be a SQL server database where necessary there is

an option for SQL reports to obtain values. For each of the objects the primary keys are an ID which is automatically generated by Naked Objects.

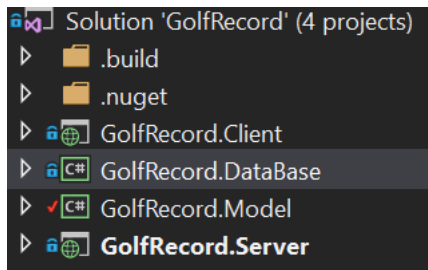
Entity Relationship diagram with Many to Many



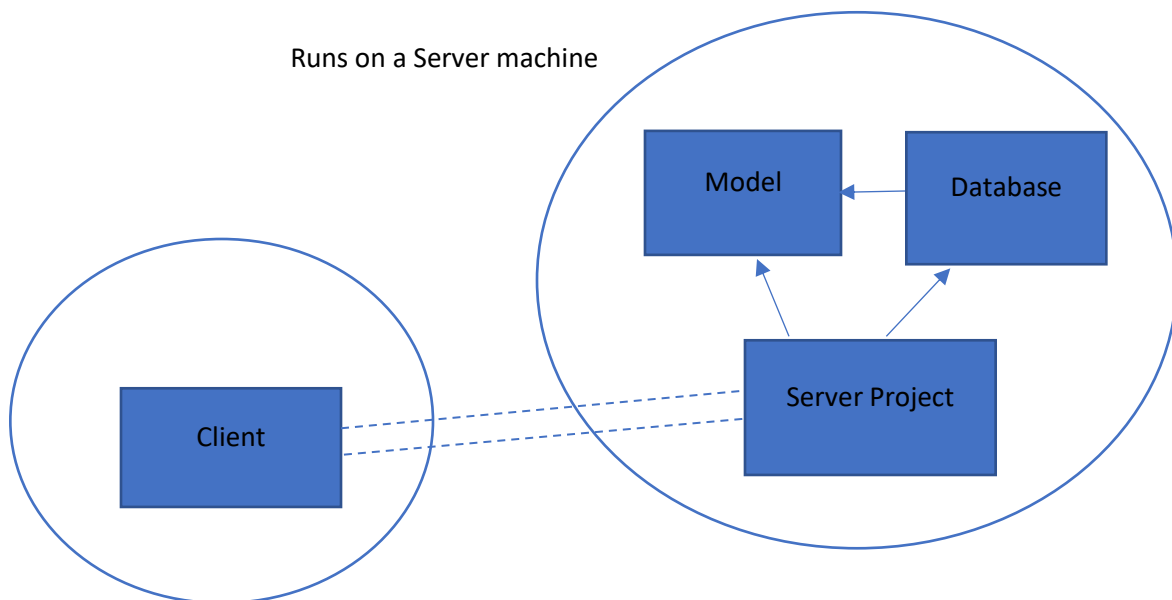
Between golfer and course, I have used dashed line since there are two possibilities with players having favourite courses there is a many to many relationship but with Club manager there is a one to one relationship. The link tables are GolferMatch and GolferCourse using the two primary keys as a compound foreign key.

Technical Solution

Overview



The screen shot above shows the layout of my solution. Since I am using Naked Objects to aid me in the assignment I have been given a template which consists of 4 projects; client, database, model, server. Each of these projects will be discussed below. The use of the template allowed for separation of concerns, this means I can separate the code into their distinct objectives, for example keeping all the code that affects the database separate from all the code that affects the client. In the future, this means that I could create a completely new client for my project without having to edit the rest. The architecture for how my project would be deployed in future is shown by the diagram below. At present the project is being run on one machine including the server and the client, but in the future a separate machine would be used to run the server.

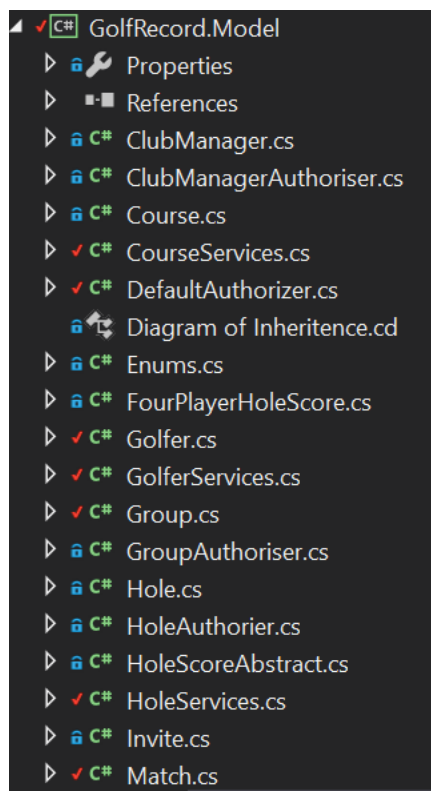


The diagram shows the client is maintained separate from the content that is run on the server machine. This is because the client is generic in the web browser and therefore can be run on a computer or a smartphone; the latter was a requirement from the customer for the project. The dashed line shows how the RESTful API is used to communicate between the client and the server. The other arrows show the dependency of each project, i.e. the Database is dependent upon the model.

The use of naked objects allowed the use of various other code and files to create a more advanced project given the time constraints. This enabled me to focus on the functionality and the business code rather than spending time on the set up and definition of the interface, which although is important would take a considerable length of time.

Model

The model is the most important part of the code as contains all the business logic and the functional application. The model is made up of three types of classes. Each is discussed below. Within the Model, I have defined all of the objects, their services and their authorisers where necessary. Much of the program revolves around the authorization. As a live multiuser application many players or managers have access to different properties and actions. These are also discussed below. The screen shot below shows the classes within the Model project.



Objects

An object is a computing construct used to represent a either a physical object or a more abstract object for example a holescore. An object consists of properties and actions. The code for an Object can be seen below.

```

public class Golfer
{
    public IDomainObjectContainer Container { set; protected get; }

    public GolferServices GolferConfig { set; protected get; }

    public CourseServices CourseConfig { set; protected get; }

    public MatchServices MatchConfig { set; protected get; }

    [NakedObjectsIgnore]
    public virtual int Id { get; set; }

    [Title][MemberOrder(1)]
    public virtual string FullName { get; set; }

    [MemberOrder(2)]
    public virtual int Handicap { get; set; }

    [Optionally][MemberOrder(4)]
    public virtual string Mobile { get; set; }

    [Optionally]
    public virtual Gender Gender { get; set; }

    public virtual Title Position { get; set; }

    public virtual string Username { get; set; }

    #region Friends (collection)
    private ICollection<Golfer> _Friends = new List<Golfer>();

    public virtual ICollection<Golfer> Friends
    {
        get
        {
            return _Friends;
        }
        set
        {
            _Friends = value;
        }
    }

    public void AddFriend(Golfer golfer)
    {
        Friends.Add(golfer);
    }

    [PageSize(3)]

```

At the top I have declared an Object Container this is so that I can access certain actions that are provided by Naked Objects where an object is not provided.

Here I am importing the methods from other classes called services.

Each of these variables are properties that belong to the object. Some of the properties have words in [] these are attributes and will be discussed down below under attributes. A screen shot showing how the interface portrays these properties is under the code. Properties will be discussed below under

An ICollection is Naked Objects version of a group of objects all under the same object type. Here is a group of Golfers named friends. Each ICollection has a method to add a value to the object this is the method below

```

        public IQueryable<Golfer> AutoComplete0AddFriend([MinLength(2)] string
matching)
    {
        return GolferConfig.AllGolfers().Where(g => g.FullName.Contains(matching));
    }
    #endregion

    #region MatchHistory (collection)
    private ICollection<Match> _MatchHistory = null;

    public virtual ICollection<Match> MatchHistory
    {
        get
        {
            return _MatchHistory;
        }
        set
        {
            _MatchHistory = value;
        }
    }
    public void AddMatchHistory(Match match)
    {
        MatchHistory.Add(match);
    }
    public IQueryable<Match> AutoComplete0AddMatchHistory([MinLength(2)] string
matching)
    {
        return MatchConfig.ShowMatches().Where(m => m.MatchName.Contains(matching));
    }
    #endregion

    #region Groups

    public Group CreateNewGroup()
    {
        var group = Container.NewTransientInstance<Group>();
        group.GroupOwner = GolferConfig.Me();
        return group;
    }

    private ICollection<Group> _Groups = new List<Group>();

    public virtual ICollection<Group> Groups
    {
        get
        {
            return _Groups;
        }
        set
        {
            _Groups = value;
        }
    }
    #endregion
}

```

This shows a complementary method. Complementary methods will be discussed below under Complementary methods.

Philip Leny

Actions
Edit
Reload

Add Friend

Create New Group

Full Name: Philip Leny
Handicap: 0
Mobile: 07123 7392833
Gender: Male
Position: Club Manager
Username: fwoodscomp@gmail.com
Course: Pebble Beach

Friends: 3 Items

Adam Chair
Aidan Hopkins
Peter Miller

Match History: Empty

Since the property Course is declared as the object Course is can be used as a link for the user to access the course.

Here are the Icollections I have expanded the collection of friends.

Properties

As mentioned above properties are the values assigned to an object. Some properties are hidden using the [NakedObjectsIgnore] attribute. Attributes will be mentioned below. With Inheritance If an object inherits from another object for example; player inherits from golfer this means that the properties declared in the super class, Golfer are accessible the sub-class player but not the other way around.

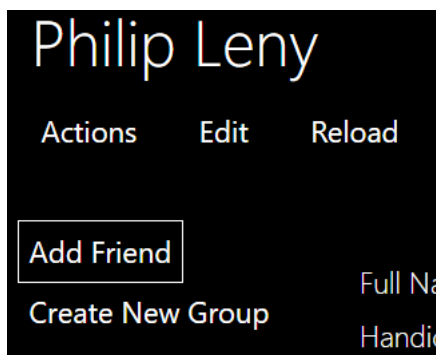
Match Class <div> Fields Properties Completed Container Course CourseConfig CourseID DateOfMatch GolferConfig Golfers HoleScores ID MatchName MatchType Winner Methods </div>	Golfer Class <div> Fields Properties Container CourseConfig Friends FullName Gender GolferConfig Groups Handicap Id MatchConfig MatchHistory Mobile Position Username WithinMatch Methods </div>	Course Class <div> Fields Properties Address Attachment AttContent AttMime AttName CourseDescripti... CourseName Holes Id Location Par PhoneNumber Rating WebsiteLink Yardage Methods </div>
---	--	--

The 4 properties, "Attachment, AttContent, AttMime, AttName:" are used to portray a photo.

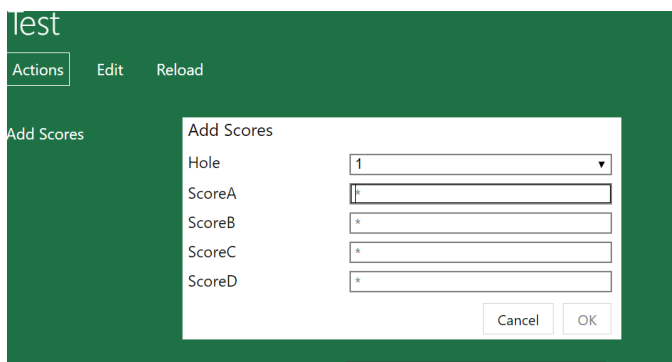
This photo shows the properties of the 3 main objects. As can be seen some objects have a lot of properties therefore it is important to use authorization and attributes, both of which are discussed below.

Actions

Actions are methods defined within an object. They are accessed by the user from within the object by pressing the tab in the top left. By pressing this tab, the list of actions accessible the user is displayed. An example of this is seen below.



To prevent a user from seeing an action you can either use authorization or hide complementary method. Some actions take in parameters. Which using attributes can be optional otherwise Naked Objects will set to necessary by default. If a method requires another object as a parameter the Naked Objects provides a drag and drop interface but the use of complementary methods provides a user with a friendly technique for adding the object. All of these techniques are further discussed below. The screen shot below shows the user interface for entering the parameters for an action.



The astericks within the boxes indicate the parameter is not optional. The OK button is faded and cannot be accessed thereby preventing errors. The Hole parameter is a drop down of the number of holes which automatically increments each time the method is called using complementary methods.

Finally if a parameter is constant it can automatically provided by default. Since an object can either be transient or persisted, a value can be specified while the object is transient before it has been persisted .In this way, the value can be specified before the user can access it.

The code for this is shown below:

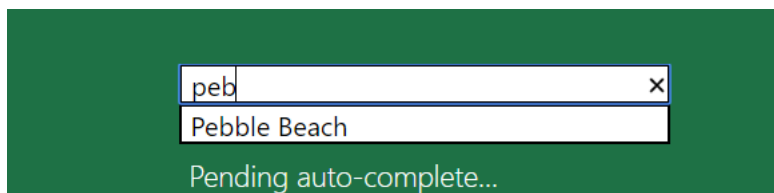
```
public Group CreateNewGroup()
{
    var group = Container.NewTransientInstance<Group>();
    group.GroupOwner = GolferConfig.Me();
    return group;
}
```

Since the variable, “group” is returned after the property group owner is specified, it is still in the transient stage. This is portrayed to the user in the following screen shot:



Complementary Methods

A complementary method is a method which complements another method. An example of this is autocomplete. To make my program more user friendly I have added autocomplete to prevent the user having to type in the full name or search through pages of golfers. An example of this is seen below.



A complementary method is declared by using `<ComplementaryMethod>0<methodto complement>`
An example of this is:

AutoComplete

```
public IQueryable<Golfer> AutoComplete0AddFriend([MinLength(2)] string matching)
```

Naked Object specifies a select group of complementary methods. The use of 0 specifies the difference between the two methods. This means that all the complementary methods are hidden from the user.

Choices

Choices specifies the possible options.

```
public IList<Hole> Choices0AddScores()
```

```

{
    if (HoleScores.Count == 0)
    {
        return Course.Holes.ToList();
    }
    else
    {
        // return Course.Holes.ToList();
        return (from h in Course.Holes
                from s in HoleScores
                where h.Id != s.HoleId //a query across two sources.
                select h).ToList();
    }
}
}
Default
Default is a method for obtaining an automatic
public Hole DefaultAddScores()
{
    int nextHole = 1;
    if (HoleScores.Count > 0)
    {
        nextHole = HoleScores.Max(hs => hs.Hole.HoleNumber) + 1;
    }
    return Course.Holes.First(h => h.HoleNumber == nextHole);
}

```

This provides a drop down menu of all the possible holes

This sets the value for hole to the last value + 1

Test

Actions

Edit

Reload

Add Scores

Add Scores

Hole

5

ScoreA

*

ScoreB

*

Cancel

OK

Course:

Pebble Beach

Date Of Match:

18 Mar 2018

Match Creator:

Philip Leny

Match Name:

Test

Match Type:

Match Play

Winner:

Golfers:

2 Items

Hole Scores:

4 Items

Hole	Score	GolferA	Score	GolferB
1		2		2
2		3		4
3		3		5
4		6		5

As you can see there is 4 scores already added and the method has automatically set itself to 5. This is due to the default complementary method.

The choices method resulted in a drop down just like an enum would. This is very helpful since some courses require 9 holes and some 18 therefore 2 sets of enums would be required.

Validate

In order to ensure the user enters a valid input the complementary method validate can be used. This complementary method takes in the parameters which are entered by the user and either returns a string or a null. When a string is returned it means the Input is invalid. This is used for checking phone

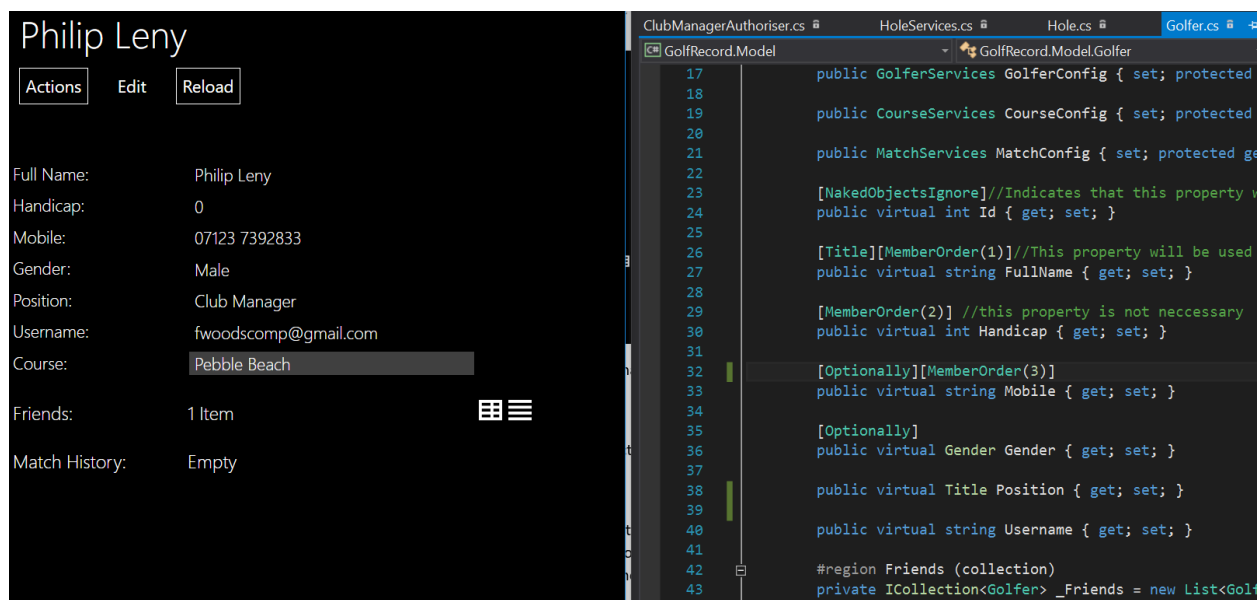
numbers are the correct length, you're not accepting a invite which will break the project but most importantly it is used to check the input for a score. This is shown in the code below. Since the 3 matches have different number of players the Validate method is required in the subclasses.

```
public string ValidateAddScores(Hole hole, int A, int B, int C, int D)
{
    if ((A <= 0) | (B <= 0) | (C <= 0) | (D <= 0))
    {
        return "A score cannot be negative or 0";
    }
    else
    {
        return null;
    }
}
```

If any of the scores are 0 the string will be return. Use of the validate method is shown in the testing section.

Attributes

Attributes are parameters used for the interface portrayed to the user. Attributes can range from ordering properties, hiding properties or declaring a property as optional. There are many different attributes. Below is an example of: MemberOrder, Title, NakedObjectsIgnore, Pagesize and finally Optionally. These are all attributes that I included in my program. The screen shot below shows most of these attributes. It is possible to have more than one attribute per property. Attributes can also be applied to methods.



MemberOrder

MemberOrder is an attribute used to portray the properties to the user in a certain order.

As can be seen from the screen shot the first property is full name the second is handicap and these are the most important. Mobile is next.

Title

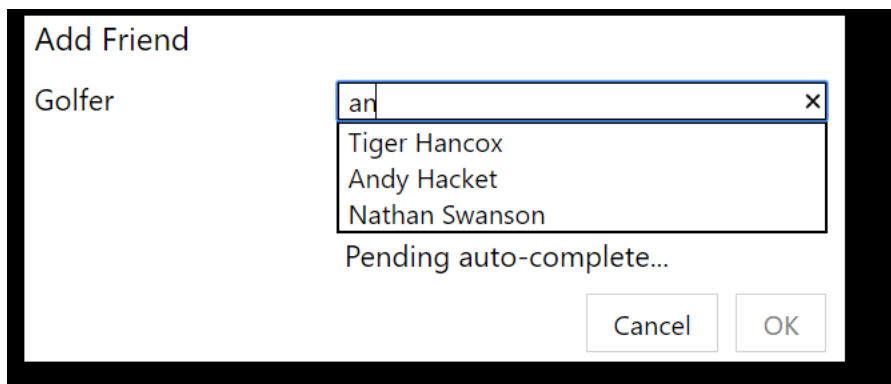
Title declares the title of a certain object. An example of this is the title of Golfer being the Golfer's name. This can be seen in the screen shot as the title.

NakedObjectsIgnore

NakedObjectsIgnore is used to prevent the user from seeing a certain property. As an example the user is prevented from seeing the ID of a golfer. This is because the Id of the golfer provide no value to the golfer but is a necessary value for the program.

PageSize

PageSize is used in a complementary method too autocomplete. When the user is entering the name of the object they are searching for it will provide a list of three possible objects. This can be seen in the screenshot below.



Optionally

This declares a property as optional which therefore doesn't need to be filled in. An example of this is a golfer entering their telephone number. Some users may not want to share such personal information to other users and therefore I have made it optional. When creating an object using a action or service method an optional parameter can be seen as having no asterick.

Editing - Unsaved Player

Save Save & Close Cancel

Full Name: x

Handicap: x

Mobile:

Gender: ▼

Position:

Username:

Match History:

As can be seen the Save and Save & Close actions are available to be used this demonstrates that I can create this golfer without entering a mobile phone number.

Services

Each of the main objects also have a services class which contains the helper methods that are used to access an object. Clicking the main header gives drop down menu of options. These options are the methods from within the services. Service methods are held outside the main class since they are methods that don't require an action to access it. For example if you need to create a golfer but you don't have a golfer to navigate from.

The code for a service is shown below:

```
namespace GolfRecord.Model
{
    public class GolferServices
    {
        #region Injected Services
        //An implementation of this interface is injected automatically by the framework
        public IDomainObjectContainer Container { set; protected get; }
        #endregion

        public IQueryable<Golfer> AllGolfers()
        {
            return Container.Instances<Golfer>();
        }

        public IQueryable<ClubManager> AllManagers()
        {
```

Once again I am using the Container to access methods since I have no object.

These methods return a type IQueryable. This means if there is a lot of that object it returns the data page by page rather than all in one table as with an ICollection

```

        return Container.Instances<ClubManager>();
    }

    public Golfer Me()
    {
        var username = Container.Principal.Identity.Name;
        var user = AllGolfers().Where(g =>
g.Username.ToUpper().Contains(username.ToUpper())).SingleOrDefault();
        if (user == null)
        {
            user = Container.NewTransientInstance<Player>();
            user.Username = Container.Principal.Identity.Name;
            user.Position = Title.Player;
            return user;
        }
        else
        {
            return user;
        }
    }

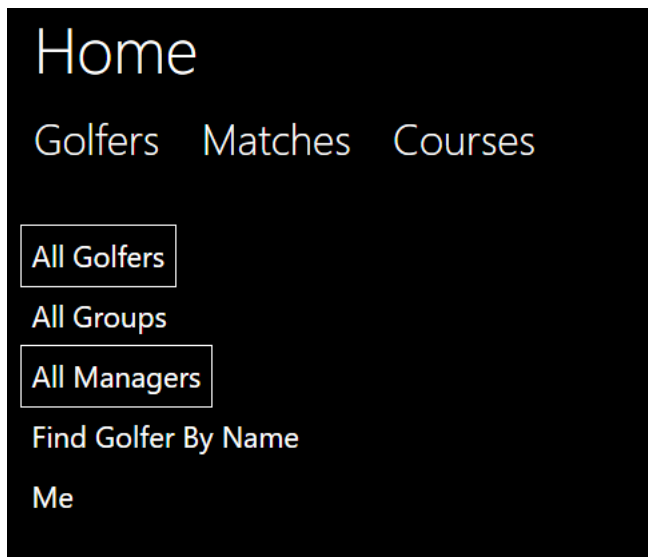
    [NakedObjectsIgnore]
    public bool IsPlayer()
    {
        if (Me() != null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public IQueryable<Golfer> FindGolferByName(string name)
    {
        return AllGolfers().Where(c => c.FullName.ToUpper().Contains(name.ToUpper()));
    }

    public IQueryable<Group> AllGroups()
    {
        return Container.Instances<Group>();
    }
}

```

These are portrayed in the user interface as:



As you can see they don't require an object to access the methods.

Programing Techniques

LINQ

Throughout multiple stages of the program I used the C# multi paradigm ability to add "LINQ" statements to my code whenever deemed useful. An example of this is within the class GolferConfig.

```
public IQueryable<Golfer> FindGolferByName(string name)
{
    //Filters students to find a match to play
    return AllGolfers().Where(c => c.FullName.ToUpper().Contains(name.ToUpper()));
}
```

The use of functional programing here is useful as it is easy to read, because you know exactly the result that you should obtain, its very reliable. Finally, it reduces the amount of statements of code significantly as function program requires no sequential statements. This can be shown in the line of code above. Without a LINQ statement this could have taken multiple lines of code.

Specifics

Algorithms

Within golf there are some relatively complex rules for calculating scores and finding the winner. The most confusing for me was the stableford system. The rules are mention in the design section. To create an algorithm I decided to split the problem up into separate functions and then work from there. This technique worked well.

```
public void AddScores(Hole hole, int ScoreA, int ScoreB, int ScoreC, int ScoreD)
{
    var hs = Container.NewTransientInstance<StablefordScores>();
    Container.Persist(ref hs);
    hs.GolferARawScore = ScoreA;
    hs.GolferBRawScore = ScoreB;
    hs.GolferCRawScore = ScoreC;
    hs.GolferDRawScore = ScoreD;
```

The top two lanes create a Stableford score which will eventual save it to the holescores collection. The scores inputted by the user are then saved as raw scores

```

        hs.Hole = hole;
        int[] Pars = StrokeIndexandHandicapEffectonPar(hole);
        int[] Scores = { ScoreA, ScoreB, ScoreC, ScoreD };
        TotalScoreCalculated(hole, Scores, hs, Pars);
        Container.Persist(ref hs);
        HoleScores.Add(hs);
        if (hole.HoleNumber == Course.Holes.Count)
        {
            FindWinner();
        }
    }
    public string ValidateAddScores(Hole hole, int A, int B, int C, int D)
    {
        if ((A <= 0) | (B <= 0) | (C <= 0) | (D <= 0))
        {
            return "A score can not be negative or 0";
        }
        else
        {
            return null;
        }
    }
    private int[] StrokeIndexandHandicapEffectonPar(Hole hole)
    {
        int[] ModifiedPar = new int[4];
        for (int i = 0; i < 4; i++)
        {
            if ((Golfers.ElementAt(i).Gender == Enums.Gender.Female) &
(hole.RedStrokeIndex != 0))
            {
                if (Golfers.ElementAt(i).Handicap >= hole.RedStrokeIndex)
                {
                    if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex + 18))
& (Course.Holes.Count == 18))
                    {
                        ModifiedPar[i] = hole.RedPar + 2;
                    }
                    else if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex +
9)) & (Course.Holes.Count == 9))
                    {
                        ModifiedPar[i] = hole.RedPar + 2;
                    }
                    else
                    {
                        ModifiedPar[i] = hole.RedPar + 1;
                    }
                }
                else
                {
                    ModifiedPar[i] = hole.RedPar;
                }
            }
            else
            {
                if (Golfers.ElementAt(i).Handicap >= hole.StrokeIndex)
                {

```

If that was the final hole it will automatically find the winner

This is the validate method to ensure the user doesn't enter an incorrect score.


```

        if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex + 18)) &
(Course.Holes.Count == 18))
        {
            ModifiedPar[i] = hole.Par + 2;
        }
        else if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex +
9)) & (Course.Holes.Count == 9))
        {
            ModifiedPar[i] = hole.Par + 2;
        }
        else
        {
            ModifiedPar[i] = hole.Par + 1;
        }
    }
    else
    {
        ModifiedPar[i] = hole.Par;
    }
}
}
return ModifiedPar;
}
private int FindScore(int Score, int Par)
{
    int TotalScore = 0;
    if (Score - Par == 1)
    {
        TotalScore += 1;
    }
    else if (Score - Par == 0)
    {
        TotalScore += 2;
    }
    else if (Score - Par < 0)
    {
        TotalScore += ((Score - Par) - 2) * (-1);
    }
    else
    {
        TotalScore += 0;
    }
    return TotalScore;
}

```

This method is working out first of all which stroke index you will be playing off and secondly based on your handicap how the par is effected.

This calculates the score based on the table shown in the analysis.

```

[NakedObjectsIgnore]
public void TotalScoreCalculated(Hole hole, int[] Scores, StablefordScores hs,
int[] handicaps)
{
    int[] TotalScore = new int[4];
    for (int i = 0; i < 4; i++)
    {
        TotalScore[i] += FindScore(Scores[i], handicaps[i]);
        Scores[i] = TotalScore[i];
    }
    hs.GolferAActualScore = Scores[0];
    hs.GolferBActualScore = Scores[1];
}

```

Calls the method which calculates the scores then saves the score and adds them to the cumulative property (TotalScore)

```

        hs.GolferCActualScore = Scores[2];
        hs.GolferDActualScore = Scores[3];
        TotalScoreA += Scores[0];
        TotalScoreB += Scores[1];
        TotalScoreC += Scores[2];
        TotalScoreD += Scores[3];
        hs.GolferATotalScore = TotalScoreA;
        hs.GolferBTotalScore = TotalScoreB;
        hs.GolferCTotalScore = TotalScoreC;
        hs.GolferDTotalScore = TotalScoreD;
    }

    [NakedObjectsIgnore]
    public void FindWinner()
    {
        int[] TotalScores = { TotalScoreA, TotalScoreB, TotalScoreC, TotalScoreD };
        for (int i = 0; i < 4; i++)
        {
            if (TotalScores.Max() == TotalScores[i])
            {
                Winner = Golfers.ElementAt(i);
            }
        }
        MatchOver = true;
    }
}

```

Finds the golfer with the highest score as that golfer is the winner then ends the match by making MatchOver True.

Authorization & Authentication

Authorization and authentication are the most important aspects of the project since the project is meant to be a multi user application. If there was no authorization the project wouldn't know who was playing and then you could change anyone's user information. To obtain the golfer Identity I am use Auth0 which is frequently used as a login system. I have not created Auth0 I am just using it. Auth0 requires an email to login which is obtained in the Me() method in GolferServices. If the Email has not already been associated with a golfer, this prompts account creation and you need to create a golfer and fill out the requested details. If however the account already exists this is how you access you profile so you can see your invites and other properties.

Each object has an authorizer which contains two functions. IsVisable() and IsEditable(). This means that a user might be able to see something like another golfers name but they can't change it. This enables the capability of multi user.

The code for an authorizer is shown below:

```

public class ClubManagerAuthoriser : ITypeAuthorizer<ClubManager>
{
    public GolferServices GolferServices { set; protected get; }

    public bool IsEditable(IPrincipal principal, ClubManager manager, string
memberName)
    {
        if ((manager.Username == principal.Identity.Name) & (memberName == "Course"))

```

```

        | (memberName == "Username")
        | (memberName == "Position"))
    {
        return false;
    }
    else if (manager.Username == principal.Identity.Name)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public bool IsVisible(IPrincipal principal, ClubManager manager, string
memberName)
{
    if (memberName == "AddMatch")
    {
        return false;
    }
    else
    if ((GolferServices.Me().FullName == null) & (memberName == "SendMessage"))
    {
        return false;
    }
    else if ((manager != GolferServices.Me()) & ((memberName == "PrivateAccount")
        | (memberName == "CreateNewGroup")
        | (memberName == "CreateNewMatch")))
    {
        return false;
    }

    else if ((manager.Friends.Contains(GolferServices.Me())) & (memberName ==
"AddFriend"))
    {
        return false;
    }
    else if ((manager.PrivateAccount == true) & (memberName == "Mobile")
        | (memberName == "Username"))
    {
        return false;
    }
    else if (memberName == "AddMatchHistory")
    {
        return false;
    }
    else if (((manager.Friends.Count == 0) & (memberName == "Friends"))
        | ((manager.Groups.Count == 0) & (memberName == "Groups"))
        | ((manager.Invites.Count == 0) & (memberName == "Invites"))
        | ((manager.Messages.Count == 0) & (memberName == "Messages"))
        | ((manager.MyMatches.Count == 0) & (memberName == "MyMatches")))
    {
        return false;
    }
}

```

Each of these attributes should never been changed.

Allows the user to change the rest of his profile but no one else.

Can't manually add a match to your history

If another user is looking at your account they can't create anything with you in it.

If you are already friends can't add the manager again

If the manager wants his account to be private the mobile and username are hidden

```

else if ((manager.Username == principal.Identity.Name) & (memberName ==
"SendMessage"))
{
    return false;
}
else if ((manager.Username == principal.Identity.Name) &
(manager.Invites.Count == 0) & ((memberName == "AcceptFriendship")
| (memberName == "AcceptGroup")
| (memberName == "AcceptMatch")
| (memberName == "DeclineInvite")
| (memberName ==
"AcceptGroupMember")))
{
    ((manager.Messages.Count == 0) & (memberName == "DeleteMessage"))
    | ((manager.MyMatches.Count == 0) & (memberName == "MyMatches")))
{
    return false;
}
else if ((manager.Username != principal.Identity.Name) & ((memberName ==
"AcceptFriendship")
| (memberName ==
"AcceptGroup")
| (memberName ==
"AcceptMatch")
| (memberName ==
"DeclineInvite")
| (memberName ==
"DeleteMessage")
| (memberName ==
"AcceptGroupMember"))))
{
    return false;
}
else if (manager.Username == principal.Identity.Name)
{
    return true;
}
else
{
    return true;
}
}
}
}
}

```

Can't send yourself a message

If the properties are empty then they are also hidden for a better user interface

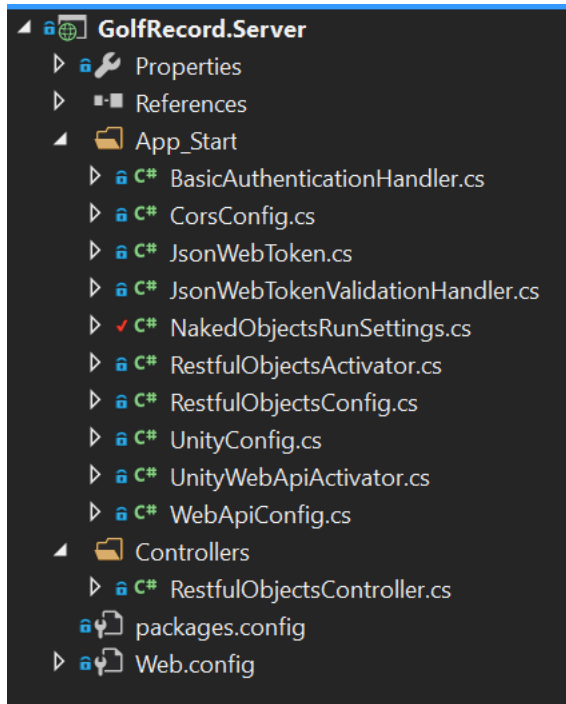
Can't accept someone else's invites or see them

As mentioned in the server all the authorizers must be declared so the server doesn't think they are separate classes.

The testing of my authorization reported in the Testing Section.

Server

The server takes all the logic from the model and the database and uses the logic to create a RESTful API for the client. The client can use this to communicate with the server using the HTTP request, response techniques. I used the server project supplied to me by Naked Objects. As mentioned in the Overview the server would eventually become a separate machine that the clients phone or laptop would access.



I have made configurations to the server within the NakedObjectsRunSettings class. The screen shot of the code is all the changes that I have personally made:

```
private static Type[] Types
{
    get
    {
        return new Type[] {
            typeof(Stableford),
            typeof(Strokeplay),
            typeof(Matchplay),
            typeof(StrokeplayScores),
            typeof(StablefordScores),
            typeof(MatchPlayHoleScore),
            typeof(ClubManager),
            typeof(Player),
            typeof(MatchInvitation),
            typeof(FriendInvitation),
            typeof(GroupInvitation),
            typeof(RequestToJoin),
            typeof(PlayerMessage),
            typeof(GroupMessage)
        };
    }
}
```

Here I had to declare all the subclasses used for inheritance since the compiler doesn't know whether they are a pure object or not.

```

    };
}

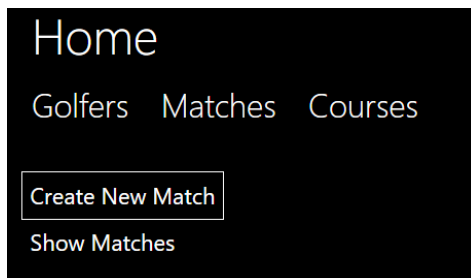
private static Type[] Services
{
    get
    {
        return new Type[] {
            typeof(GolferServices),
            typeof(MatchServices),
            typeof(CourseServices),
            typeof(HoleServices)
        };
    }
}

public static IMenu[] MainMenus(IMenuFactory factory)
{
    return new IMenu[] {
        factory.NewMenu<GolferServices>(true, "Golfers"),
        factory.NewMenu<MatchServices>(true, "Matches"),
        factory.NewMenu<CourseServices>(true, "Courses")
    };
}

```

The services are also defined here so that the server doesn't confuse the services as objects to be added to the menu.

Here I am defining the main menu that is visible at the homepage. The reason for using the services since the homepage isn't an object these are the methods that we can navigate with and retrieve or create domain objects.



This screen shot is the product of the code above. I have expanded the Matches showing all the methods in the MatchServices.

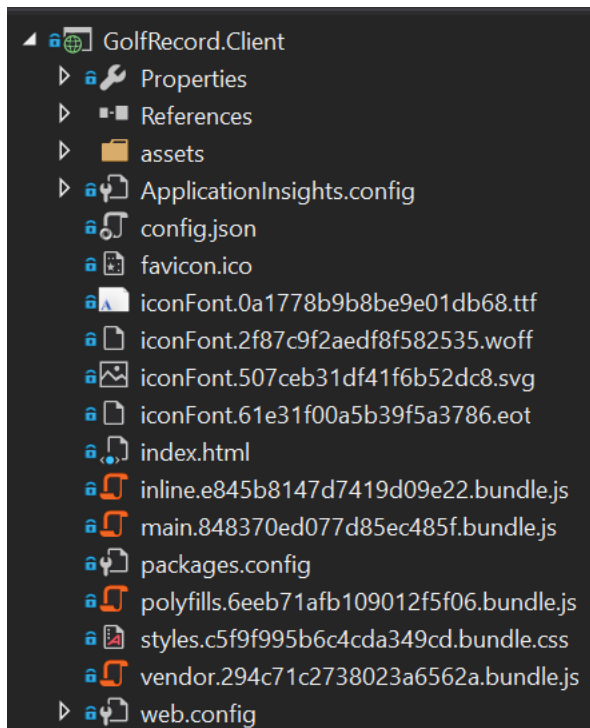
```

public static IAuthorizationConfiguration AuthorizationConfig()
{
    var config = new AuthorizationConfiguration<DefaultAuthorizer>();
    // config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");
    // config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");
    config.AddTypeAuthorizer<MatchStrokePlay, StrokePlayAuthoriser>();
    config.AddTypeAuthorizer<MatchStableFord, StableFordAuthoriser>();
    config.AddTypeAuthorizer<MatchPlay, MatchPlayAuthoriser>();
    config.AddTypeAuthorizer<Player, PlayerAuthoriser>();
    config.AddTypeAuthorizer<ClubManager, ClubManagerAuthoriser>();
    config.AddTypeAuthorizer<Hole, HoleAuthorier>();
    config.AddTypeAuthorizer<Group, GroupAuthoriser>();
    config.AddTypeAuthorizer<Match, MatchAuthoriser>();
    return config;
}
}

```

Finally, I have had to declare all the Authorizers so that the server doesn't think they are objects than our accessed by the user.

Client



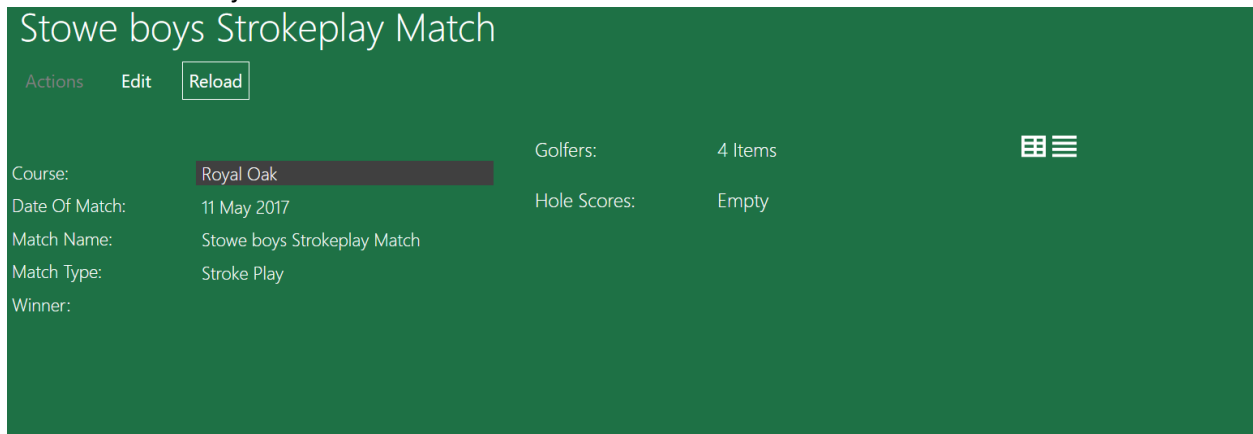
Generic UI

The client is a generic user interface that has been supplied by Naked Objects. There are thousands of lines of code which vary in language. For example, some of the classes are written in HTML such as the `index.html`; other classes are written in css. One of the beneficial factors of using the NakedObjects generic UI arises when executing an action that takes in an object, the user may drag and drop an object in. Here is a screen shot of the user dragging an object into a parameter.

Customisation

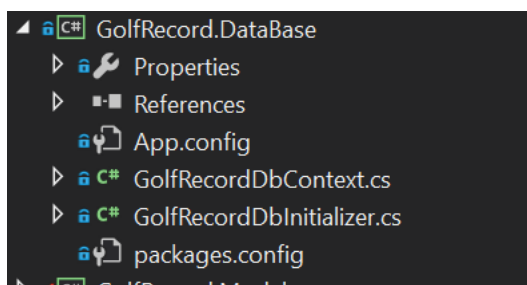
I have made a few changes to the code all of which are in the `config.json` file. Here I have told the solution I am using authentication and an authorizer this is the demo version of `auth0`. I will talk about `auth0` at a later stage in the Specifics section. In addition, in the `config.json` file I have chosen to change a few colours of objects in order to help the user distinguish what they are looking at. An example of this

is for the match object.



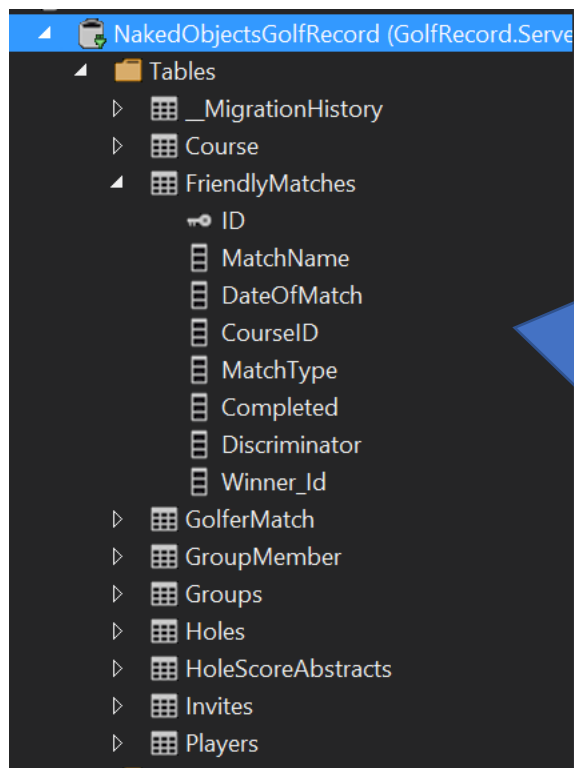
As you can see the Match is green, but the course is black this helps the user to know that from match they can travel to the course by clicking that link which now stands out. Although this has been removed and instead colours are used in the Invites.

Database



Entity Framework

The database employs an object relational mapping framework made by Microsoft called entity framework. This synergizes well with my project as it automatically sets up the database through the code I have written. where each object is a separate table and an ICollection is a 1 to many relationships. The list of tables is shown below.



Here I have expanded the Matches table to show all the columns. The key next to the ID represents the primary key. I have customized the table here to called it "Friendly Matches" In future this system could be used for complete matches with separate rules.

There are a few tables which I have manually mapped which will be talked about in the manual mapping section below. The use of Entity Framework means that I can use SQL reports to obtain certain information. This is useful for analysing the data collected and for a user to check the data for their past matches.

Manual Database Mapping

Whenever the entity framework hasn't quite managed to map the database correctly I have had to manually edit it. This is done by creating a function in the DbContext class which defines a new table with a primary key using modelbuilder. This function is then called in a separate function called OnModelCreating

I have had to manually map a table twice; one is a many to many relationship where many Golfers have many match histories, but a match can belong to many golfers. To solve this, I have defined a link table called Golfer match which can be seen in the screenshot above. This new table is a link table used to turn the many to many into two one to many relationships. In the new table golfer match it uses a compound foreign key as the primary key which are GolferID and MatchId. The code below shows how I have done this. The code is written within the DbContext and is being called using a method from the modelbuilder function which can be seen below the screen shot.

```
private void DefineGolferMatch(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Match>()
        .HasMany(x => x.Golfers)
        .WithMany(x => x.MatchHistory)
        .Map(x =>
        {
            x.ToTable("GolferMatch");
        x.MapLeftKey("GolferId");
            x.MapRightKey("MatchId");
        });
}
```

Here I am stating that the many to many relationship currently exists.

Here I am defining the new tables name and what the compound foreign primary key is.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    DefineGolfer(modelBuilder.Entity<Golfer>());
    DefineMatch(modelBuilder.Entity<Match>());
    DefineCourse(modelBuilder.Entity<Course>());
    DefineGolferMatch(modelBuilder);
    DefineGroupMember(modelBuilder);
}
```

Only the bottom two are creating a new table. The other 3 methods are for customization which will be mentioned in Database Customization.

The second example of manual mapping is a one to itself relationship. This is for the creating groups Where a Golfer may have many groups as with many golfers. Once again to solve this problem I have created a new table called GroupMember which uses a new object that has been created in the model called Groups. A group has a golfer who is the owner and a collection of Golfers who belong to the group.

```
private void DefineGroupMember(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Golfer>()
        .HasMany(x => x.Groups)
        .WithMany(x => x.Members)
        .Map(x =>
        {
            x.ToTable("GroupMember");
            x.MapLeftKey("Id");
            x.MapRightKey("GolferId");
        });
}
```

Group is an entirely new object that I created from with the Model which can be seen In the Model. I created the new object so that I can have two different primary keys and therefor it can be mapped correctly.

Database Customization

In addition to manual mapping I have also been able to customize the database where I feel a better name could be used. An example of this can be seen below.

COURSE	
Id	
NameOfCourse	
Location	
PostCode	
Par	
DescriptionOfCourse	
Rating	
WebsiteLink	
Yardage	
PhoneNumber	

Here is the table for course. I have renamed a few columns in the table so that they read better to the user. DescriptionOfCourse is actually CourseDescription this does change the property when the code is running its only for the database.

```
private void DefineCourse(EntityTypeConfiguration<Course> courseconfiguration )
{
    courseconfiguration.ToTable("Course");
    courseconfiguration.HasKey(c => c.Id);
    courseconfiguration.Property(c => c.Address).HasColumnName("PostalAddress");
    courseconfiguration.Property(c =>
c.CourseDescription).HasColumnName("DescriptionOfCourse");
    courseconfiguration.Property(c =>
c.CourseName).HasColumnName("NameOfCourse");
}
```

Initializer

For creating example data used for test runs I am using an Initializer. It is a separate class within the Database project. The class consists of a group of methods that I have created returning an object which are saved to the Context. These methods are then accessed above with sample data being passed in for each parameter required. There are various other techniques which could be used instead for example SQL Inserts but I chose to use an Initializer as it reads better with its syntactic sugar of "=". This means that there are already golfers and matches inside the program before I run it and so to do a test run I don't need to create 4 separate golfers then add the 4 golfers to the match and then add 18 scores for each golfer. To add example data, I needed to create a help method which returned that object. This method is then called with each value declared and then saved to the database. Changes currently made while the project is running doesn't get saved but for future when the application goes live this would be changed so that matches get saved.

The code below shows an example of a method within the DBInitializer;

I have set the Initializer up so that if the user doesn't enter a match when calling the method is automatically set the type to strokeplay since this is the most common.

```
private Match AddNewMatch(string name, DateTime date, int courseID, MatchType matchType
= MatchType.StrokePlay)
{
    // work for each match type
    Match m = null;
```

Here I am using a switch case statement to ensure the match is one of the three types and therefore doesn't break.

```

switch (matchType)
{
    case MatchType.StrokePlay:
        m = new MatchStrokePlay() { MatchName = name, DateOfMatch = date,
CourseID = courseID, MatchType = matchType };
        break;
    case MatchType.MatchPlay:
        m = new MatchPlay() { MatchName = name, DateOfMatch = date, CourseID
= courseID, MatchType = matchType };
        break;
    case MatchType.StableFord:
        m = new MatchStableFord() { MatchName = name, DateOfMatch = date,
CourseID = courseID, MatchType = matchType };
        break;
    default:
        break;
}
Context.Matches.Add(m);
Context.SaveChanges();
return (m);
}

```


An example of how this method would be called is shown below. Once again at the end of the method I am saving the changes to the context. Although this doesn't need to be at the end of each call but I have done it to be safe.

```

var s9 = AddNewMatch("Stowe Mixed Stableford Team", date1, 4, MatchType.StableFord);
context.SaveChanges();

```

The screen shot below shows all the example matches I have created for testing. There is; an all-male, all-female and mixed gender match for each match type. Some of these matches have scores already added in but some don't.

Page 1 of 1; viewing 11 of 11 items		
Stowe boys Strokeplay Match		
Stowe girls stroke play match		
Stowe Mixed Stroked play Match		
Stowe Boys MatchPlay Team		
Stowe girls MatchPlay Team		
Stowe Mixed MatchPlay Team		
Stowe Mens Stableford Team		
Stowe Womens Stableford Team		
Stowe Mixed Stableford Team		
Test Empty Match		
Strokeplay match with scores except last		

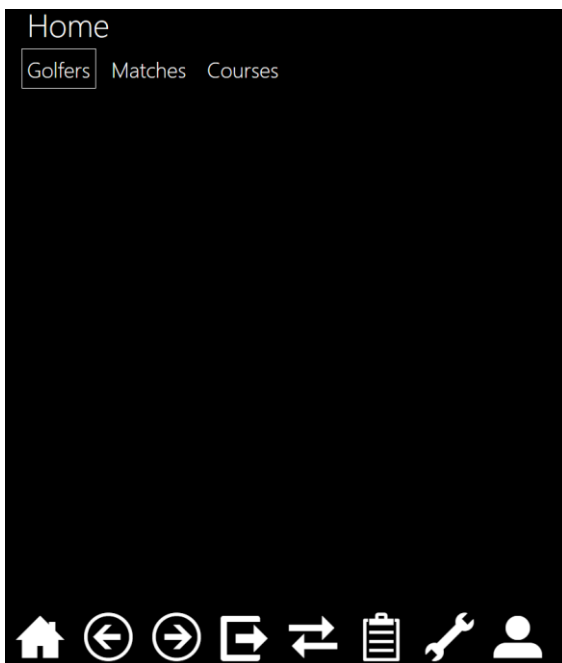
Testing

Since my project is a multi-user application which runs on one device currently I used Microsoft Edge and Google Chrome to simulate two different users on two different devices.

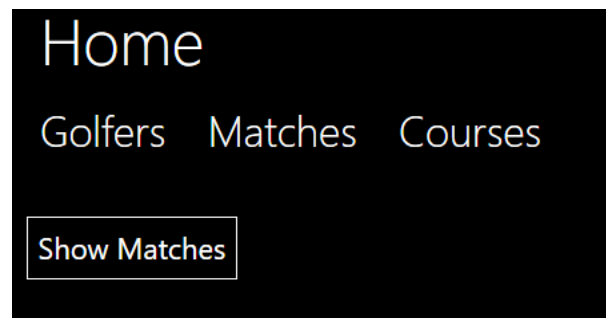
Test 1: Creating a New Account.

Creating a new account is a simple process which involves two steps. The first step is logging in using the auth0. This can be done using a Facebook, Google or Auth0 account. This was mentioned in the Authorization section of the Technical Solution. The screen shot below shows the interface that greets the user after they have just logged to Auth0.

What the user can and can't do once logged in for the first time.



As a new account the user hasn't currently created a golfer. This means that certain aspects of the application should not be accessible. For example, the user shouldn't be capable of creating a match. The new user also shouldn't be able to create a group join a group or send a message to a golfer. All of these methods have been hidden using authorization. As each of these methods and actions are on separate objects, I have only shown the screen shot for not being able to create a new match.



How to create an account.

To create a Golfer the new user must press "Golfers" and then press "Me". Upon pressing "Me" they are met with this screen shot. The "Position" and "Username" is not editable so that only a systems manager can create a club manager or change their position. The username is used to identify the golfer and is taken from the login with Auth0 and therefore it can't be changed.



Private Accounts

The new user then fills in all the details but has left mobile blank as it is an optional parameter. From this moment he can create a new match, join groups and add friends as seen in the expanded actions. The new user then decides to edit their profile and add their phone number, however in this case they want their account to remain private meaning that no-one can see their mobile. To do this they press edit and tick the box labelled private account.

Editing - Test

Save Cancel See field validation message(s).

Full Name: Test

Handicap: 12

Mobile: 032413241234

Incorrect length of mobile number please check

Gender: Male

Position: Player

Private Account: ☒

Username: fwoodscomp2@gmail.com

When the user enters their phone number they make a mistake but due to a validation complementary method added to Mobile, the user is warned and can't save the changes until the mobile is the correct length.

To show the private account is working, another user logs in and looks at the new user's profile. The two screen shots below show the new user's profile: once with private account ticked and the other when the account is not private.

Test

Actions Reload

Add Match

Send Message

Full Name: Test

Handicap: 12

Gender: Male

Position: Player

Test

Actions Reload

Add Match

Send Message

Full Name: Test

Handicap: 12

Mobile: 01234567891

Gender: Male

Position: Player

Username: fwoodscomp2@gmail.com

Test 2: Enriching the User's Profile

For the purposes of these test we shall be using a golfer who has been created in the DBInitializer. The golfer's name is John Smith. The screen shot below shows the Golfer's current profile before the tests have taken place.

Adding Friends

Sending the friend request

John is a new user but some of his friends are already apart of the application, so he decides to add them as friends. To do this he goes onto his profile via the Me method in Golfers and presses Add Friend. Add friend has a complementary method and therefore he should be able to just enter the first 3 characters of his friend's name.

Accepting the friend request

When adding a friend, it sends an invite to that golfer. They are not friends until the recipient, Philip Leny in this case, has accepted the friend request. As mentioned I have changed the colour of each invite. Therefore, for Philip he should have a yellow invite and he should only have the accept friendship and decline invite action available. This is shown in the screen shot below.

When accepting the friendship, Philip drags a yellow invite as the parameter and presses ok. Then they can become friends.

Philip Leny

Actions

Edit

Reload

Add Friend

Create New Group

Create New Match

Full Name:

Philip Leny

Handicap:

0

Mobile:

07123 7392833

Gender:

Male

Position:

Club Manager

Private Account:

☐

Course:

Pebble Beach

Friends:

2 Items

Peter Miller

John Smith

Groups:

1 Item

My Matches:

3 Items

Declining the Invite

If Philip decides to decline the invite, just as he would for accepting an invite he drags and drops the invite into the parameter box and then the invite is removed from his invites and is deleted. In future a message will be sent to the golfer who sent the invite to inform him.

Error Catching

Already Being Friends

If the golfers are already friends and John has sent Philip two invites to become friends, then using a validation method it warns the user and says that they can't accept the invite since they are already friends and Philip needs to delete the friend request.

Accept Friendship

Invite

Friend Invite

You are already friends with this person please delete the invite

See field validation message(s).

Cancel

OK

Adding the wrong Invitation

If Philip accidentally adds the wrong type of invitation it will not let them enter the invite. A red box surrounds the parameter.

Accept Friendship

Invite

* (drop Group Invite

Cancel

OK

Friends:

2 Items

Groups:

2 Items

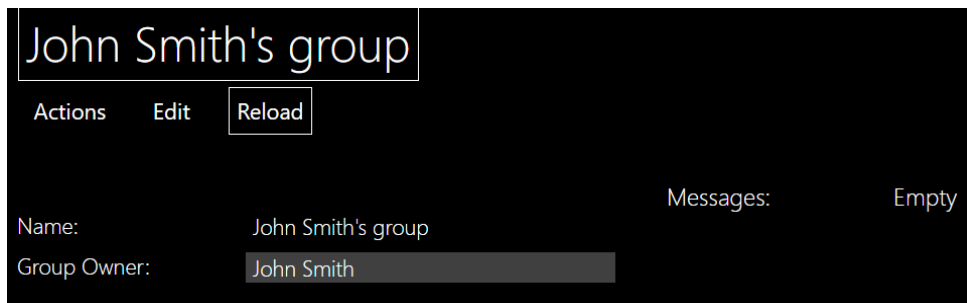
Invites:

2 Items

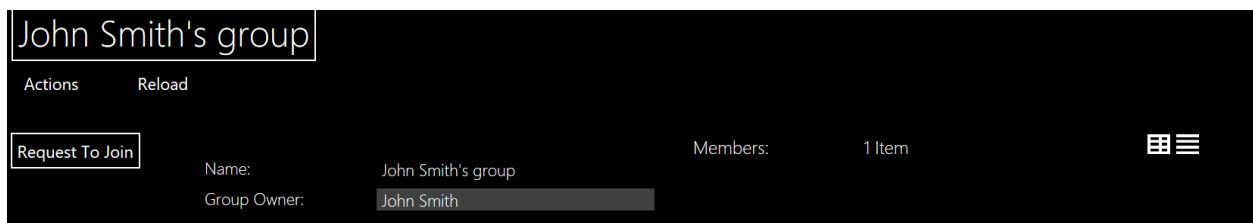
Joining a Group

Creating a group

When creating a group, John uses the method from his profile create new group where he is faced with this screen.



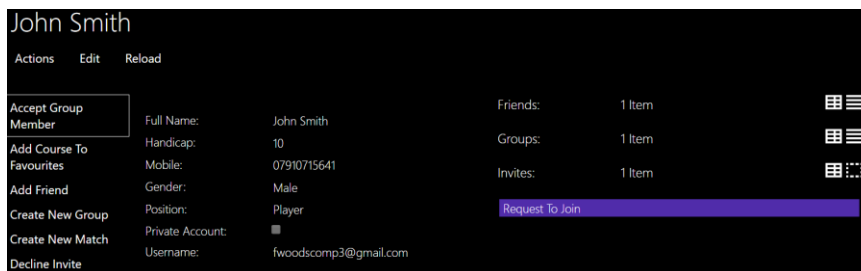
John can press edit and change all the values if he wants to. But if Philip went onto his group they would be faced with:



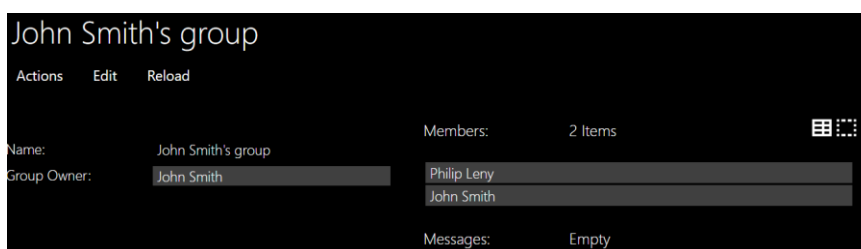
As you can see Philip can't edit the group and the only method he can access is Request To Join . He cannot access add new member and send group messages. Even though there are no messages on the group, Philip can't see the empty Icollection which is an important aspect as it shows privacy from within the group.

Requesting To join the group

Philip decides that he wants to join so he presses request to join. This sends the invite to John allowing him to access the accept group member action. A problem that I encounter is that you can spam Request to Join and all these invites would show up in the invites for John. The screenshot below show the Philip requesting to Join.



Only the accept group member action is on show.



Philip Leny is now a member. As you can see John Smith is also a member now. This is because it automatically adds when he does an action or edits.

Being sent an invitation to join a group

If John wanted to add Philip and Philip hasn't requested to join then he can use one of the actions available to him from the group object. The method Add New Member has the complementary method autocomplete.

Add New Member

Golfer

Pending auto-complete...

Philip Leny

Actions Edit Reload

Accept Group
Add Friend
Create New Group
Create New Match
Decline Invite

Accept Group
Invite

Cancel OK

Full Name: Philip Leny
Handicap: 0
Mobile: 07123 7392833
Gender: Male

Friends: 2 Items
Groups: 1 Item
Invites: 1 Item
My Matches: 3 Items

Group Invite

John Smith's group

Actions Edit Reload

Name: John Smith's group
Group Owner: John Smith

Members: 2 Items

Messages: Empty

Now that Philip is apart of the group he can add his friends and send messages but he can't edit the groups' name or group owner, only the group owner can.

John Smith's group

Actions Reload

Add New Member
Send Group Message

Name: John Smith's group
Group Owner: John Smith

Members: 2 Items
Messages: Empty

Group Messages

As shown in the above screen shots you can only see the group messages if you are apart of the group The same applies for sending the messages, The screen shot below shows John sending a group message.

A problem with persistence meant that I had to create a message and then allow the message sender to edit the message.

John Smith

Actions

Edit

Reload

Senders Name:

John Smith

Attachment:

No image

Content:

Please press edit to enter your response.

Group:

John Smith's group

Sender:

John Smith

Editing - John Smith

Save

Cancel

Senders Name:

John Smith

Attachment:

No image

Content:

Found a really nice new golf club

Group:

John Smith's group

Sender:

John Smith

A problem that I encountered is that you can't add an Attachment through the edit, so instead I added a new method "Add or Change Attachment" to which any golfer can add an attachment to a message.

John Smith

Actions

Edit

Reload

Add Or Change Attachment

Add Or Change Attachment

New Attachment

Choose file

No file chosen

Cancel

OK

Senders Name:

John Smith

Attachment:

No image

Content:

Please press edit to enter your response.

Group:

John Smith's group

Sender:

John Smith

Sending Messages

Any golfer should be able to send a message to any golfer.

Sending the Message

John wants to ask Philip whether he wants to play a match. To do this John goes on to Philip's profile and then presses actions. The only action that John should see is Send Message.

Philip Leny

Actions

Reload

Send Message

Full Name:

Philip Leny

Handicap:

0

Gender:

Male

Position:

Club Manager

Course:

Pebble Beach

Friends:

2 Items

Groups:

1 Item

My Matches:

3 Items

This screen shot shows that the only action visible is send Message.

John Smith

Actions Edit Reload

Add Or Change Attachment

Respond To Message

Senders Name: John Smith

Attachment: No image

Content: Hey do you want to play a game

Reciever: Philip Leny

Sender: John Smith

Here Philip can add a photo. A bug that I have found is that he can Respond To Message. I will change the authorizer to prevent the Message sender from seeing.

Philip Leny

Actions Reload

Full Name: Philip Leny

Handicap: 0

Gender: Male

Position: Club Manager

Course: Pebble Beach

Friends: 2 Items

Groups: 2 Items

Messages: 1 Item

My Matches: 3 Items

John Smith

The message that John just sent Is now in Philip's messages which only Philip can see.

Responding to a message

To respond to a message, he uses an action that was shown above. This is the same as send message, but it swaps around the sender and receiver.

Test 3: Creating a Match

Browsing Courses

John now wants to start playing with some friends. He's enriched his profile but doesn't know which courses are good, where they are and all their information. So to start, he browses through the course.

The screen shots below show John looking at a course and then John looking at one of the holes in that course.

Pebble Beach

Actions Reload

Address: 1700 17 - Mile Drive, PebbleBeach, CA 93953

Club Manager: Philip Leny

Course Description: The sport of golf is at its worldwide best at Pebble Beach Resorts. Whether you want to play the most exciting closing hole in golf, finally master the toughest hole on the PGA TOUR or simply anticipate walking in the footsteps of golf's greatest names, we invite you to become a part of the incomparable experience that is Pebble Beach.

Course Name: Pebble Beach

Location: Croatia

Par: 72

Phone Number: (800) 877-0597

Photo Of Course:

Facilities: Empty

Holes: 18 Items

1

Actions Reload

Hole Number: 1

White Yards: 400

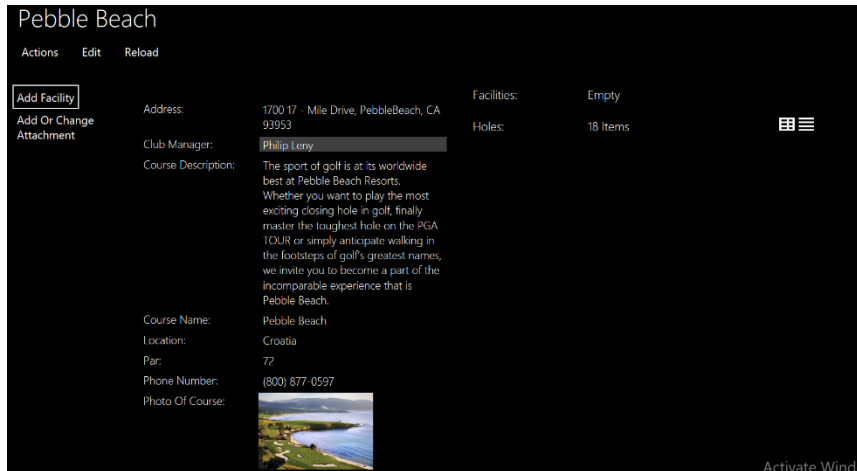
Par: 3

Stroke Index: 2

Course: Pebble Beach

Photo Of Hole:

As you can see from the screenshots when John looks at the course or the hole he can see a photo of each and only the information that has been filled in for Hole. John can't see the edit action and actions is greyed out since neither are available to him. However, they are available to the Club manager which can be seen below.



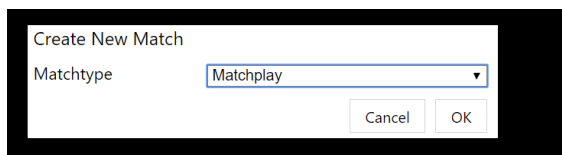
Since Facilities is a collection of Enum, Naked Object will not allow this. To overcome this problem I created an Object called Facilities and then inside each Facility is an enum and an ID. The Facility Objects were manually made in the DBInitializer and were added using the Add Facility method using an autocompleted Action. The Facilities can't be changed and are uneditable.

Sending The invitations

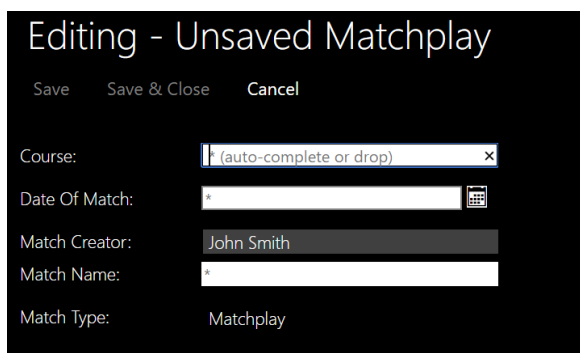
John decides that Pebble Beach is the correct course for him and now he wants to create his match and send the invites asking his friends if they want to join. To do this John must first create a match.

Creating the match

Creating a match is simple. It can be done in two locations, either from your profile or from the Match main menu. On creating a match the first step is to choose which matchtype you want to play.



After choosing your match type you can fill out the data about your match.



Since creating a match is the same method for each match type I am only showing it for one match type.

Once John has created his match he can add some players. He has two options. He can either send an invite to a friend or he can choose to play with someone who is not registered on the app. Both scenarios will be shown below. Upon calling any action John is automatically added to the group of golfers.

The screenshot shows a match creation interface for a '1 on 1 against Philip Leny' match. At the top, there are buttons for 'Actions', 'Edit', and 'Reload'. Below these, the match details are listed: Course (Pebble Beach), Date Of Match (15 Apr 2018), Match Creator (John Smith), Match Name (1 on 1 against Philip Leny), Match Type (Matchplay), and Winner (empty). To the right, the 'Golfers' section shows '1 Item' and a list containing 'John Smith'. The 'Hole Scores' section is currently 'Empty'.

In addition, on John profile is a collection called My Matches and this match is automatically added. This is where John can see his match history.

The screenshot displays the 'My Matches' collection, which contains '1 Item'. The item is a match titled '1 on 1 against Philip Leny'.

Adding a golfer by sending an invite works the same as any other invite it can either be declined or accepted. The method is autocompleted as shown below.

The screenshot shows a 'Send Invite' dialog box. On the left, there is a sidebar with 'Add Unregistered Golfer' and 'Send Invite' options. The main area is titled 'Send Invite' and contains a 'Golfer' label. A text input field has 'Ph' entered, and a dropdown menu is open showing suggestions: 'Philip Leny' and 'Bethany philip'. Below the suggestions, it says 'Pending auto-complete...'. At the bottom right, there are 'Cancel' and 'OK' buttons.

When adding an unregistered Golfer, a few details need to be entered: name, gender and handicap. Since this golfer has no username he will not be able to fill in any details and John will have to do everything.

1 on 1 against Philip Leny

Actions Edit Reload

Add Unregistered Golfer
Name
Handicap
Gender
Cancel OK

Golfers: 1 Item
John Smith
Hole Scores: Empty

Course: Pebble Beach
Date Of Match: 15 Apr 2018
Match Creator: John Smith
Match Name: 1 on 1 against Philip Leny
Match Type: Matchplay

John decides that he wants to play with a friend called Henry who isn't registered.

1 on 1 against Philip Leny

Actions Edit Reload

Course: Pebble Beach
Date Of Match: 15 Apr 2018
Match Creator: John Smith
Match Name: 1 on 1 against Philip Leny
Match Type: Matchplay
Winner:

Golfers: 2 Items

Full Name	Handicap	Mobile	Friends	Gender	Groups	My Matches	Position	Private Account	Username
John Smith	10	07990715641	1 Item	Male	1 Item	1 Item	Player		fwoodscomp3@gmail.com
Henry	12		Empty	Male	John Smith	Empty	Empty	Empty	1 Item

Another point is that if John decided that he wanted to edit the course he wouldn't be able to change the Winner.

Now that there are two golfers in the match, and since matchplay only requires two golfers the Add Scores method will become available to the Golfer and the adding an unregistered Golfer or Send Invite methods will not be available.

1 on 1 against Philip Leny

Actions Edit Reload

Add Scores

Course: Pebble Beach
Date Of Match: 15 Apr 2018
Match Creator: John Smith
Match Name: 1 on 1 against Philip Leny
Match Type: Matchplay
Winner:

Too many players want to join

When adding Henry John forgot that an invite had previously been sent to Philip who now accepts the invite.

Error message received from server

Message: Object reference not set to an instance of an object.

Code: InternalServerError(500)

Description: A software error has occurred on the server.


Stack Trace :

```
at GolfRecord.Model.Golfer.AcceptMatch(MatchInvitation invite) in C:\GolfRecord\GolfRecord.Model\Golfer.cs:line 275
at NakedObjects.Core.Util.DelegateUtils.<>c__DisplayClass10_0'2.<ActionHelper1>b_0(Object target, Object[] param)
at NakedObjects.Core.Util.DelegateUtils.<>c__DisplayClass7_0.<WrapException>b_0(Object target, Object[] param)
at NakedObjects.Meta.Facet.ActionInvocationFacetViaMethod.Invoke(INakedObjectAdapter inObjectAdapter, INakedObjectAdapter[] parameters, lifecycleManager, IMetamodelManager manager, ISession session, INakedObjectManager nakedObjectManager, IMessageBroker messageBroker, transactionManager)
at NakedObjects.Core.Spec.ActionSpec.Execute(INakedObjectAdapter nakedObjectAdapter, INakedObjectAdapter[] parameterSet)
at NakedObjects.Facade.Impl.FrameworkFacade.ExecuteAction(ActionContext actionContext, ArgumentsContextFacade arguments)
at NakedObjects.Facade.Impl.FrameworkFacade.<>c__DisplayClass38_0.<ExecuteObjectAction>b_0()
at NakedObjects.Facade.Impl.FrameworkFacade.MapErrorsUI(Func`1 f)
```

This created an error as there will be too many players in the match. To prevent this error occurring I edited the Accept match action to send a message to Philip saying there are too many golfers in the match already.

```
public void AcceptMatch(MatchInvitation invite)
{
    if ((invite.match.MatchType == MatchType.Matchplay) & (invite.match.Golfers.Count == 2))
    {
        PlayerMessage msg = Container.NewTransientInstance<PlayerMessage>();
        msg.Content = "This match is already full";
        Container.Persist(ref msg);
        Messages.Add(msg);
    }
    else if (invite.match.Golfers.Count == 4)
    {
        var msg = Container.NewTransientInstance<PlayerMessage>();
        msg.Content = "This match is already full";
        Container.Persist(ref msg);
        Messages.Add(msg);
    }
    else if (invite.match.Golfers.Contains(invite.Receiver))
    {
        var msg = Container.NewTransientInstance<PlayerMessage>();
        msg.Content = "You are already appart of this match";
        Container.Persist(ref msg);
        Messages.Add(msg);
    }
    {
        invite.match.Golfers.Add(invite.Receiver);
    }
}
```

Now if Philip decided to accept the invite, a message informing Philip there are too many players in the match appears.

Messages:		1 Item			
Senders Name		Attachment	Content	Reciever	Sender
			This match is already full		

Playing the first hole.

Now that the match has been created and there are the correct number of people in the match, John can start playing the match.

Inputting an invalid score for a golfer

Since the lowest number of shots, a Golfer can take is one, a score can't be negative or zero.

Add Scores

Hole

1

ScoreA

0

ScoreB

3

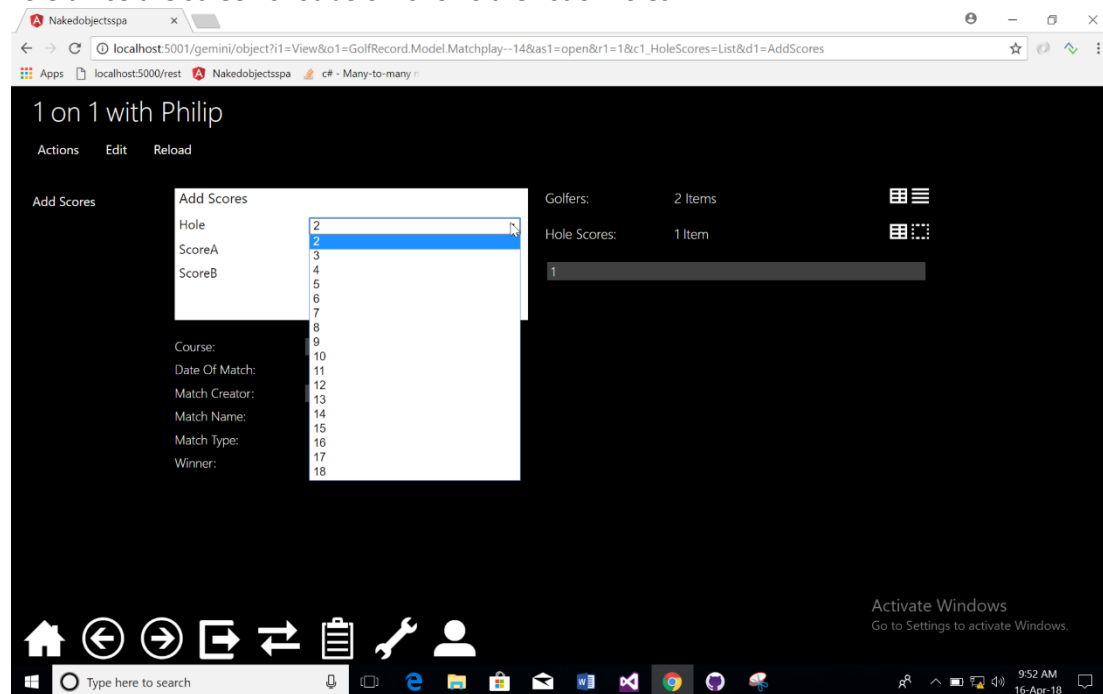
A score can not be negative or 0

Cancel

OK

Editing a score

John accidentally added the wrong score in the Add Scores method and needs to edit a score. Since this is done through the super class it is the same for each match. As the scores are being filled out the list of holes to choose from is reduced. This means that the user can chose to skip hole but not play the same hole twice the screen shot below shows the list of holes.



After John has entered the first score he can't go back as it's not an option on the drop-down menu. This means to edit the score he must press on the hole score and then change each score manually working out the handicap and total scores in their head using the edit action.

Editing - 1

Save Cancel

Hole:

Golfer A Raw Score:

Golfer A Actual Score:

Golfer A Total Score:

Golfer B Raw Score:

Golfer B Actual Score:

Golfer B Total Score:

Hole Winner:

Score calculation

In each of the scores there will be a raw score and an actual score.

Match play

Matchplay also has a hole winner since that is how the scoring system works. For the first few holes there is no handicap effect hence the actual and raw scores are the same until they reach hole 6.

Hole	Golfer A Raw Score	Golfer A Actual Score	Golfer A Total Score	Golfer B Raw Score	Golfer B Actual Score	Golfer B Total Score	Hole Winner
1	3	3	1	4	4	0	John Smith
2	4	4	0	3	3	2	John Smith
3	3	3	0	3	3	2	
4	3	3	1	4	4	2	Philip Leny
5	3	3	2	4	4	2	Philip Leny
6	3	3	2	3	2	3	John Smith
7	3	3	3	4	4	3	Philip Leny

When Hole Winner is empty that means that the result for that hole was a draw.

Stroke play

There are no special additions to the score card for stroke play. Since the handicap is only used at the end, the actual score is the cumulative column of results and the total score is taken out.

Hole	Golfer A Raw Score	Golfer A Actual Score	Golfer B Raw Score	Golfer B Actual Score	Golfer C Raw Score	Golfer C Actual Score	Golfer D Raw Score	Golfer D Actual Score
1	3	3	3	3	3	3	3	3
2	3	6	3	6	4	7	3	6
3	3	9	4	10	3	10	5	11

Stableford

Hole	Golfer A Raw Score	Golfer A Actual Score	Golfer A Total Score	Golfer B Raw Score	Golfer B Actual Score	Golfer B Total Score	Golfer C Raw Score	Golfer C Actual Score	Golfer C Total Score	Golfer D Raw Score	Golfer D Actual Score	Golfer D Total Score
1	3	2	2	4	2	2	4	1	1	5	1	1

The golfers in the match have a handicap of 0, 15,1,10 respectively. The stroke index is 2 and the par is 3. This means that the respective pars are 3,4,3,4 and that since A and B got par and C, D were 1 over they get 2,2,1,1 points respectively. Which is correct.

Test 4: Finishing a Match

Upon finishing a match, the program automatically works out the winner and prevents the add score method from being called. Below is the test for each of the match types. When there is a draw the winner stays blank but the add scores is still hidden and the bool MatchOver = true.

John has just finished the 9th hole and is adding the scores to the card of a 9-hole course.

1 on 1 with Philip

Actions Edit Reload

Add Post Match Description

Course: Royal Oak

Date Of Match: 11 May 2017

Description Of Match:

Match Creator: John Smith

Match Name: 1 on 1 with Philip

Match Over: ☒

Match Type: Matchplay

Winner: Philip Leny

Golfers: 2 Items

Hole Scores: 9 Items

AS you can see the only method available to john is the add post-match description and that property is shown. In addition, the match Over property is ticked showing the match Is over.

Looking through match history

Even though the match is over this match is still in my matches collection on his profile. This means that he can look though the scores.

Hole	Golfer A Raw Score	Golfer A Actual Score	Golfer A Total Score	Golfer B Raw Score	Golfer B Actual Score	Golfer B Total Score	Hole Winner
1	3	3	0	4	2	1	John Smith
2	3	3	0	4	3	1	
3	3	3	1	6	5	1	Philip Leny
4	3	3	2	5	4	1	Philip Leny
5	4	4	2	5	4	1	
6	4	4	2	4	3	2	John Smith
7	3	3	3	5	4	2	Philip Leny
8	3	3	3	4	3	2	
9	3	3	4	5	4	2	Philip Leny

Activ
Go to

Adding a description of how the match went.

Now that the match is over he can write a description of how he felt the match went. Using the method and that way Philip can also see how the match went.

Description Of Match: It was a windy day

Evaluation

In this section I shall evaluate my program against the Objectives which were set out in the Analysis. This will be done by comparing them against the User Needs. In addition, I will locate areas with possible improvements and a list of changes and additions that would be made if I were to have more time. It is important to note that the application is currently run of a computer but for future purposed would be designed in to an application.

Comparing my project against User Needs

My project has fulfilled all the Must Do's from the User needs. The user can create a profile which is shown from the Me action in Golfers. Using this profile, the user can create a match add his friends or send invites to other golfers. While playing this match a score card is created to the user. A positive from my project is that only one golfer is required to enter the scores and it is shown to all the other golfers. This means that the other golfers don't have to fill in the scores as well but can see if the golfer which is entering the scores is cheating. As shown in my testing the winner is automatically calculated. From the "my matches" collection on your profile you can go through past matches and see the score card on each hole. A community can be formed around the friends and even the group messages that were conceptualized in the Should be able to section. The use of photos on each hole and photos of a course mean that browsing and searching for the best course is helpful and easy. To embellish this having each course, have a club manager that looks after the course's profile means that they will stay up to date and if any new facilities have been created the program will know straight away. Finally, since all the matches are public you can see where your friends are playing and the date of their match.

Ease of Use

Due to the use of complimentary methods like autocomplete to bring the amount of drag and drop down to a minimum. A personal favorite area is in the match when adding a score. The use of choices and default where a list of holes that haven't been played on has been created is a great aspect. This allows the user to skip holes and realize which ones they have missed. I believe the naming of my properties and actions have helped the user to understand what is going on although, one area to improve on is when the golfer first logs in. The use of the Me action in Golfers to create the profile is untrivial and even though it works well at creating the method a new user might be confused on how to create a new account.

Areas of Improvement

In future I were to spend more time on this project, the first thing that I would change is the user interface. Now the user is faced with dark colors and tabs. To improve on this, I would have a background that the user could choose from. The user could also have a profile picture which doesn't need to be a face could be anything. In future I would also aim to reduce the stableford algorithm. I believe that is could still get refined down and use aspects from the stroke play and match play algorithms this will improve the efficiency of my project. I have already mentioned that the first time the user logs in would be improved upon. A more advanced improvement would be to take the GPS position of the phone and locate that on the map of the hole. This means the user could see their position on the photo of the hole helping them with club choice for their next shot. In addition, as pointed out in the design having a little comment on the score card saying, "replace the pitch marks", or a personalized message from the course portrayed to the user though the match is another aspect that would be improved. An aspect that was conceptualized while programing but never created was also an

appointments section for coaches where golfers could book a time slot, or the coaches could keep a diary.

Conclusion

Although the golfing application is still run on a computer, through exhaustive testing I have managed to remove the minor bugs from the system and create a functional golfing application that could be used on a course. The work spent on messages, groups and friends will bring together a strong community and enable players, course managers and coaches to further enhance their enjoyment of the game.

Appendix – All My Code

In this section have copied all my code which I wrote.

Client

Config.json

```
"colors": {  
  "typeMap": {  
  
    },  
  "subtypeMap": {  
    "GolfRecord.Model.GroupInvitation": 2,  
    "GolfRecord.Model.MatchInvitation": 3,  
    "GolfRecord.Model.FriendInvitation": 4,  
    "GolfRecord.Model.RequestToJoin": 5  
  }  
},
```

Database

GolfRecordDbContext

namespace GolfRecord.DataBase

```
{  
    public class GolfRecordDbContext : DbContext  
    {  
        public GolfRecordDbContext(string dbName) : base(dbName)  
        {  
            Database.SetInitializer(new GolfRecordDbInitializer());  
        }  
        public DbSet<Match> Matches { get; set; }  
        public DbSet<Golfer> Golfers { get; set; }  
        public DbSet<Course> Courses { get; set; }  
        public DbSet<Hole> Holes { get; set; }  
        public DbSet<HoleScoreAbstract> HoleScore { get; set; }  
        public DbSet<StrokeplayScores> StrokeplayScores { get; set; }  
        public DbSet<StablefordScores> StablefordScores { get; set; }  
        public DbSet<MatchPlayHoleScore> MatchPlayScore { get; set; }  
        public DbSet<Invitation> Invite { get; set; }  
        public DbSet<Group> Groups { get; set; }  
        public DbSet<Facility> Facilities { get; set; }  
  
        //      public DbSet<Friend> Friend { get; set; }  
        //      public DbSet<Message> Message { get; set; }  
        protected override void OnModelCreating(DbModelBuilder modelBuilder)  
        {  
            DefineGolfer(modelBuilder.Entity<Golfer>());  
            DefineMatch(modelBuilder.Entity<Match>());  
            DefineCourse(modelBuilder.Entity<Course>());  
            DefineGolferMatch(modelBuilder);  
            DefineGroupMember(modelBuilder);  
        }  
        private void DefineCourse(EntityTypeConfiguration<Course> courseconfiguration )  
        {  
            courseconfiguration.ToTable("Course");  
        }  
    }  
}
```

```

        courseconfiguration.HasKey(c => c.Id);
        courseconfiguration.Property(c => c.Address).HasColumnName("PostalAddress");
        courseconfiguration.Property(c =>
c.CourseDescription).HasColumnName("DescriptionOfCourse");
        courseconfiguration.Property(c =>
c.CourseName).HasColumnName("NameOfCourse");
    }

    private void DefineMatch(EntityTypeConfiguration<Match> matchconfiguration)
    {
        matchconfiguration.ToTable("FriendlyMatches");
    }

    private void DefineGolferMatch(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Golfer>()
            .HasMany(x => x.MyMatches)
            .WithMany(x => x.Golfers)
            .Map(x =>
            {
                x.ToTable("GolferMatch");
            x.MapLeftKey("MatchId");
                x.MapRightKey("GolferId");
            });
    }

    private void DefineGroupMember(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Golfer>()
            .HasMany(x => x.Groups)
            .WithMany(x => x.Members)
            .Map(x =>
            {
                x.ToTable("GroupMember");
                x.MapLeftKey("Id");
                x.MapRightKey("GolferId");
            });
    }
    private void DefineGolfer(EntityTypeConfiguration<Golfer> golferconfiguration)
    {
        golferconfiguration.ToTable("Players");
    }
}

```

```

}

```

GolfRecordDbInitializer

```

public class GolfRecordDbInitializer : DropCreateDatabaseAlways<GolfRecordDbContext>
{
    public DateTime date1 = new DateTime(2017, 05, 11);
    private GolfRecordDbContext Context;
    protected override void Seed(GolfRecordDbContext context)
    {
        this.Context = context;
    }
}

```



```

#region Courses
var ro = AddNewCourse("Royal Oak", "Scotland", "http://www.golfroyaloak.com",
"Royal Oak Golf Club is a beautiful, and challenging 9 Hole Golf Course with a great
location, (10 minutes from the downtown core, 15 minutes from Swartz Bay Ferry
Terminal).", "540 Marsett Place, Victoria B.C. V8Z-5M1", 32, 2000, " 250 658 - 1433");
var pb = AddNewCourse("Pebble Beach", "Croatia",
"https://www.pebblebeach.com/golf/", "The sport of golf is at its worldwide best at
Pebble Beach Resorts. Whether you want to play the most exciting closing hole in golf,
finally master the toughest hole on the PGA TOUR or simply anticipate walking in the
footsteps of golf's greatest names, we invite you to become a part of the incomparable
experience that is Pebble Beach.", "1700 17 - Mile Drive, PebbleBeach, CA 93953", 72,
6828, "(800) 877-0597");
var ss = AddNewCourse("Stowe Golf Club", "England",
"https://www.stowe.co.uk/house/venue-hire/golf-club", "Stowe has a 9-hole course situated
in the historic setting of Lancelot 'Capability' Brown's landscaped garden. The Club has
an extensive range of social gatherings and competitions to get involved in.", "Stowe
House Preservation Trust, Stowe, Buckingham, MK18 5EH", 66, 4500, "01280 818282");
var st = AddNewCourse("Silverstone Golf Club", "England", "
http://www.silverstonegolfclub.co.uk/", "Set on the Buckinghamshire/Northamptonshire
border and surrounded by forest this beautiful 18 hole parkland course was designed by
David Snell and offers the golfers a great challenge", "Silverstone Road, Stowe,
Buckingham MK18 5LH", 72, 6600, "01280-850005");
context.SaveChanges();

AddNewHole(pb, 1, 3, 400, 2);
AddNewHole(pb, 2, 4, 600, 3);
AddNewHole(pb, 3, 3, 420, 4);
AddNewHole(pb, 4, 3, 450, 6);
AddNewHole(pb, 5, 4, 620, 9);
AddNewHole(pb, 6, 3, 410, 11);
AddNewHole(pb, 7, 3, 350, 1);
AddNewHole(pb, 8, 3, 450, 5);
AddNewHole(pb, 9, 4, 550, 7);
AddNewHole(pb, 10, 4, 600, 15);
AddNewHole(pb, 11, 4, 700, 18);
AddNewHole(pb, 12, 3, 600, 17);
AddNewHole(pb, 13, 4, 500, 8);
AddNewHole(pb, 14, 3, 410, 10);
AddNewHole(pb, 15, 4, 575, 12);
AddNewHole(pb, 16, 3, 550, 13);
AddNewHole(pb, 17, 4, 600, 14);
AddNewHole(pb, 18, 3, 525, 16);
Context.SaveChanges();
AddNewHole(ro, 1, 3, 400, 1);
AddNewHole(ro, 2, 3, 420, 2);
AddNewHole(ro, 3, 4, 525, 6);
AddNewHole(ro, 4, 3, 410, 4);
AddNewHole(ro, 5, 4, 530, 7);
AddNewHole(ro, 6, 5, 650, 9);
AddNewHole(ro, 7, 3, 410, 3);
AddNewHole(ro, 8, 3, 430, 5);
AddNewHole(ro, 9, 4, 500, 8);
Context.SaveChanges();

AddNewHole2(ss, 1, "Home Park", 122, 108, 3, 13, 122, 3, 15);
AddNewHole2(ss, 2, "Chatham", 281, 254, 4, 9, 268, 4, 9);
AddNewHole2(ss, 3, "Copper Beech", 296, 285, 4, 11, 296, 4, 11);
AddNewHole2(ss, 4, "Rotunda", 409, 329, 4, 1, 409, 4, 1);

```

```

AddNewHole2(ss, 5, "Hog Pond", 186, 170, 3, 5, 186, 3, 5);
AddNewHole2(ss, 6, "Doric", 277, 277, 4, 17, 277, 4, 13);
AddNewHole2(ss, 7, "Old Acacia", 327, 317, 4, 7, 281, 4, 7);
AddNewHole2(ss, 8, "Lakeside", 302, 302, 4, 7, 281, 4, 7);
AddNewHole2(ss, 9, "Caroline", 112, 112, 3, 15, 112, 3, 17);
AddNewHole2(ss, 10, "Home Park", 122, 108, 3, 14, 122, 3, 16);
AddNewHole2(ss, 11, "Chatham", 281, 254, 4, 10, 268, 4, 10);
AddNewHole2(ss, 12, "Copper Beech", 296, 285, 4, 12, 296, 4, 12);
AddNewHole2(ss, 13, "Rotunda", 409, 393, 4, 2, 409, 4, 2);
AddNewHole2(ss, 14, "Hog Pond", 186, 170, 3, 6, 186, 3, 6);
AddNewHole2(ss, 15, "Doric", 242, 221, 4, 18, 242, 4, 14);
AddNewHole2(ss, 16, "Old Acacia", 312, 312, 4, 4, 312, 4, 4);
AddNewHole2(ss, 17, "Lakeside", 282, 274, 4, 8, 274, 4, 8);
AddNewHole2(ss, 18, "Caroline", 112, 112, 3, 16, 112, 3, 18);
Context.SaveChanges();

AddNewHole2(st, 1, "Jim's Folly", 506, 494, 5, 12, 458, 5, 6);
AddNewHole2(st, 2, "Pentimore", 412, 403, 4, 4, 378, 4, 10);
AddNewHole2(st, 3, "Deb's Mere", 135, 131, 3, 18, 116, 3, 18);
AddNewHole2(st, 4, "Father's Barn", 401, 397, 4, 2, 322, 4, 2);
AddNewHole2(st, 5, "Swallowtail", 279, 274, 4, 16, 247, 4, 12);
AddNewHole2(st, 6, "The Break", 206, 191, 3, 10, 173, 3, 14);
AddNewHole2(st, 7, "MiddleMead", 530, 514, 5, 8, 497, 5, 4);
AddNewHole2(st, 8, " Pheasant's Walk", 157, 153, 3, 14, 153, 3, 16);
AddNewHole2(st, 9, " The Halfway", 425, 409, 4, 6, 301, 4, 8);
AddNewHole2(st, 10, "The Borehole", 400, 384, 4, 11, 305, 4, 17);
AddNewHole2(st, 11, "Red Ditches", 388, 382, 4, 7, 336, 4, 7);
AddNewHole2(st, 12, "Holly Hill", 507, 498, 5, 13, 406, 5, 13);
AddNewHole2(st, 13, "Nite Mere", 410, 400, 4, 3, 369, 4, 3);
AddNewHole2(st, 14, "Amen Corner", 401, 391, 4, 1, 385, 4, 1);
AddNewHole2(st, 15, "Earlswood", 368, 363, 4, 15, 354, 4, 11);
AddNewHole2(st, 16, "Christie's Creek", 149, 145, 3, 17, 135, 3, 15);
AddNewHole2(st, 17, "Buttockspire", 390, 384, 4, 9, 365, 4, 5);
AddNewHole2(st, 18, "Wet Leys", 583, 563, 5, 5, 406, 5, 9);
context.SaveChanges();

#endregion

#region MatchCreators
var CM = AddNewClubManager("Philip Leny", 0, "07123 7392833", Gender.Male,
pb, "fwoodscomp@gmail.com");
var TH = AddNewClubManager("Tiger Hancox", 13, "56473 829106",
Gender.Male,ro, "few@more.com");
var DS = AddNewClubManager("Dandy Smith", 3, "01238 942343",
Gender.Female,st);
var NS = AddNewClubManager("Nathan Swanson", 12, "01296 749916",
Gender.Male,ss);
var JH = AddNewGolfer2("Jimmy Hart", 10, "01234 123414", Gender.Male);
var TL = AddNewGolfer2("Theresa Lolly", 22, "05324 234519", Gender.Female);
var OW = AddNewGolfer2("Obi Wan", 10, "12345 098765", Gender.Male);
var AE = AddNewGolfer2("Albert Einstein", 0, "57324 321414", Gender.Male);
var MC = AddNewGolfer2("Marie Curie", 15, "01234 132443", Gender.Female);
var JB = AddNewGolfer2("Jim Breithaupt", 0, "01942 872356", Gender.Male);
context.SaveChanges();
#endregion

#region Matches
var s1 = AddNewMatch("Stowe boys Strokeplay Match", date1, 1,TH);

```

```

Context.SaveChanges();
s1.Golfers.Add(TH);
var p2 = AddNewGolfer(s1, "Rory Gabriel", 14, "01296 234324", Gender.Male);
AddNewGolfer(s1, "Rookie Player", 12, "07810 675443", Gender.Male);
AddNewGolfer(s1, "Adam Chair", 13, "01234 753234", Gender.Male);
context.SaveChanges();
var s2 = AddNewMatch("Stowe girls stroke play match", date1, 2, DS);
Context.SaveChanges();
s2.Golfers.Add(DS);
AddNewGolfer(s2, "Rafa Lauren", 16, "19876 543210", Gender.Female);
AddNewGolfer(s2, "Roger Perry", 16, "10202 304050", Gender.Female);
AddNewGolfer(s2, "Andy Hacket", 8, "01020 030405", Gender.Female);
Context.SaveChanges();

var s3 = AddNewMatch("Stowe Mixed Stroked play Match", date1, 3, NS);
context.SaveChanges();
s3.Golfers.Add(NS);
AddNewGolfer(s3, "Rafferty Reeves", 8, "01289 743123", Gender.Male);
AddNewGolfer(s3, "Bethany philip", 12, "01233 123414", Gender.Female);
AddNewGolfer(s3, "Rachel wright", 3, "01234 321413", Gender.Female);
Context.SaveChanges();

var s4 = AddNewMatch("Stowe Boys MatchPlay Team", date1, 2, JH,
MatchType.Matchplay);
context.SaveChanges();
s4.Golfers.Add(JH);
AddNewGolfer(s4, "Andrew Tait", 2, "01234 3214312", Gender.Male);
Context.SaveChanges();
var s5 = AddNewMatch("Stowe girls MatchPlay Team", date1, 4, TL,
MatchType.Matchplay);
Context.SaveChanges();
s5.Golfers.Add(TL);
AddNewGolfer(s5, "Mary Jane", 5, "01324 590123", Gender.Female);
Context.SaveChanges();

var s6 = AddNewMatch("Stowe Mixed MatchPlay Team", date1, 3, OW,
MatchType.Matchplay);
Context.SaveChanges();
s6.Golfers.Add(OW);
AddNewGolfer(s6, "Mace Windu", 5, "98754 123415", Gender.Female);
Context.SaveChanges();

var s7 = AddNewMatch("Stowe Mens Stableford Team", date1, 2, AE,
MatchType.Stableford);
context.SaveChanges();
s7.Golfers.Add(AE);
AddNewGolfer(s7, "Max Born", 2, "12345 123456", Gender.Male);
AddNewGolfer(s7, "Isaac Newton", 5, "56392 123441", Gender.Male);
AddNewGolfer(s7, "Nikola Tesla", 7, "09832 111111", Gender.Male);
Context.SaveChanges();

var s8 = AddNewMatch("Stowe Womens Stableford Team", date1, 1, MC,
MatchType.Stableford);
context.SaveChanges();
s8.Golfers.Add(MC);
AddNewGolfer(s8, "Rosamund Flip", 18, "01135 353426", Gender.Female);
AddNewGolfer(s8, "Genie Booch", 4, "01823 988132", Gender.Female);

```

```

        context.SaveChanges();
        var s9 = AddNewMatch("Stowe Mixed Stableford Team", date1, 4, JB,
MatchType.Stableford);
        context.SaveChanges();
        s9.Golfers.Add(JB);
        AddNewGolfer(s9, "Saul Muliple", 5, "01492 845483", Gender.Male);
        AddNewGolfer(s9, "Mary Teapot", 9, "01832 144324", Gender.Female);
        AddNewGolfer(s9, "Linda Green", 14, "01234 123441", Gender.Female);
        Context.SaveChanges();

        var s11 = AddNewMatch("Strokeplay match with scores except last", date1, 1,
CM, MatchType.Strokeplay);
        context.SaveChanges();
        s11.Golfers.Add(CM);
        AddNewGolfer(s11, "Noah Castillo", 6, "01728 123412", Gender.Male);
        AddNewGolfer(s11, "Cody Turner", 6, "01383 132414", Gender.Male);
        AddNewGolfer(s11, "Aidan Hopkins", 5, "01256 122122", Gender.Male);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(0), 3, 4, 3, 4);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(1), 3, 4, 4, 4);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(2), 4, 3, 3, 4);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(3), 3, 5, 3, 3);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(4), 3, 4, 5, 3);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(5), 4, 5, 6, 5);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(6), 5, 4, 4, 4);
        AddScoreStrokePlay(s11, ro.Holes.ElementAt(7), 3, 4, 5, 4);

        // AddScoreStrokePlay(s2,

        var MP = AddNewGolfer2("Peter Miller", 1, "08188 464638", Gender.Male,
"woodssy@gmail.com");
        context.SaveChanges();

        AddFriend(CM, MP);
        AddFriend(MP, CM);
        context.SaveChanges();
        var TSP = AddNewMatch("Test Strokeplay Match", date1, 1, MP,
MatchType.Strokeplay);
        context.SaveChanges();
        TSP.Golfers.Add(MP);
        TSP.Golfers.Add(CM);
        TSP.Golfers.Add(TH);
        TSP.Golfers.Add(p2);

        var TMP = AddNewMatch("Test Matchplay Match", date1, 2, CM,
MatchType.Matchplay);
        context.SaveChanges();
        TMP.Golfers.Add(MP);
        TSP.Golfers.Add(CM);
        Context.SaveChanges();

        var TSF = AddNewMatch("Test Stableford Match", date1, 1, MP,
MatchType.Stableford);
        context.SaveChanges();
        TSF.Golfers.Add(MP);
        TSF.Golfers.Add(CM);
        TSF.Golfers.Add(TH);

```

```

TSF.Golfers.Add(p2);
Context.SaveChanges();
#endregion

#region Facilities
AddNewFacility(Facilities.ATM);
AddNewFacility(Facilities.ChangingRooms);
AddNewFacility(Facilities.ClubHouse);
AddNewFacility(Facilities.ConferenceRoom);
AddNewFacility(Facilities.FreeWifi);
AddNewFacility(Facilities.Hotel);
AddNewFacility(Facilities.ProShop);
AddNewFacility(Facilities.Restaurant);
AddNewFacility(Facilities.WeddingVenue);
Context.SaveChanges();
#endregion

EditCourse(pb, CM);
EditCourse(ro, TH);
EditCourse(st, DS);
EditCourse(ss, NS);
Context.SaveChanges();

#region Groups
var group1 = AddNewGroup("Philip's Group", CM);
Context.SaveChanges();
group1.Members.Add(CM);
group1.Members.Add(MP);
group1.Members.Add(JB);
group1.Members.Add(MC);
group1.Members.Add(AE);
Context.SaveChanges();
MP.PrivateAccount = true;
Context.SaveChanges();
#endregion

var Me = AddNewGolfer2("John Smith", 10, "07910715641", Gender.Male,
"fwoodscomp3@gmail.com");
Context.SaveChanges();

AddFriend(Me, CM);
AddFriend(CM, Me);
var group2 = AddNewGroup("John's Group", Me);
Context.SaveChanges();
group2.Members.Add(Me);
group2.Members.Add(CM);
context.SaveChanges();

var MainMatch = AddNewMatch("1 on 1 with Philip", date1, ro.Id, Me,
MatchType.Matchplay);
MainMatch.Golfers.Add(Me);
MainMatch.Golfers.Add(CM);
context.SaveChanges();

var MainMatch2 = AddNewMatch("Strokeplay", date1, pb.Id, Me,
MatchType.Strokeplay);

```

```

        MainMatch2.Golfers.Add(Me);
        MainMatch2.Golfers.Add(CM);
        MainMatch2.Golfers.Add(AE);
        MainMatch2.Golfers.Add(JB);
        context.SaveChanges();

        var MainMatch3 = AddNewMatch("Stableford", date1, pb.Id, Me,
MatchType.Stableford);
        MainMatch3.Golfers.Add(Me);
        MainMatch3.Golfers.Add(MC);
        MainMatch3.Golfers.Add(MP);
        MainMatch3.Golfers.Add(CM);
        context.SaveChanges();
    }

    private Player AddNewGolfer(Match m, string name, int handi, string mobile,
Gender gender, Title title = Title.Player)
    {
        var g = new Player() { FullName = name, Handicap = handi, Mobile = mobile,
Gender = gender, Position = title };
        Context.Golfers.Add(g);
        Context.SaveChanges();
        m.Golfers.Add(g);
        return (g);
    }

    private Player AddNewGolfer2(string name, int handi, string mobile, Gender
gender, string username = "", Title title = Title.Player)
    {
        var g2 = new Player() { FullName = name, Handicap = handi, Mobile = mobile,
Gender = gender, Username = username, Position = title };
        Context.Golfers.Add(g2);
        Context.SaveChanges();
        return (g2);
    }

    private Match AddNewMatch(string name, DateTime date, int courseID, Golfer
MatchCreator, MatchType matchType = MatchType.Strokeplay)
    {
        // work for each match type
        Match m = null;
        switch (matchType)
        {
            case MatchType.Strokeplay:
                m = new Strokeplay() { MatchName = name, DateOfMatch = date, CourseID
= courseID, MatchType = matchType, MatchCreator = MatchCreator };
                break;
            case MatchType.Matchplay:
                m = new Matchplay() { MatchName = name, DateOfMatch = date, CourseID
= courseID, MatchType = matchType, MatchCreator = MatchCreator };
                break;
            case MatchType.Stableford:
                m = new Stableford() { MatchName = name, DateOfMatch = date, CourseID
= courseID, MatchType = matchType, MatchCreator = MatchCreator };
                break;
            default:
                break;
        }
        Context.Matches.Add(m);
        Context.SaveChanges();
        return (m);
    }

```

```

    }

    private StrokeplayScores AddScoreStrokePlay(Match Match, Hole hole, int ScoreA,
int ScoreB, int ScoreC, int ScoreD)
    {
        var s = new StrokeplayScores { Hole = hole, GolferARawScore = ScoreA,
GolferBRawScore = ScoreB, GolferCRawScore = ScoreC, GolferDRawScore = ScoreD };
        Context.StrokeplayScores.Add(s);
        Context.SaveChanges();
        Match.HoleScores.Add(s);
        return (s);
    }

    private Course AddNewCourse(string CourseName, string Location, string
WebsiteLink, string ShortParagraph, string address, int par, int yardage, string Phone)
    {
        var c = new Course() { CourseName = CourseName, Location = Location,
WebsiteLink = WebsiteLink, CourseDescription = ShortParagraph, Address = address, Par =
par, Yardage = yardage, PhoneNumber = Phone };
        Context.Courses.Add(c);
        return (c);
    }

    private Hole AddNewHole(Course c, int HoleNumeber, int par, int distance, int
strokeindex)
    {
        var h = new Hole() { HoleNumber = HoleNumeber, Par = par, WhiteYards =
distance, StrokeIndex = strokeindex };
        Context.Holes.Add(h);
        Context.SaveChanges();
        c.Holes.Add(h);
        return (h);
    }

    private Hole AddNewHole2(Course c, int HoleNumeber, string name, int whiteyards,
int yellowyards, int par, int strokeindex, int ladiesredyards, int redpar, int
redstrokeindex)
    {
        var h = new Hole() { HoleNumber = HoleNumeber, Name = name, WhiteYards =
whiteyards, YellowYards = yellowyards, Par = par, StrokeIndex = strokeindex,
LadiesRedYards = ladiesredyards, RedPar = redpar, RedStrokeIndex = redstrokeindex };
        Context.Holes.Add(h);
        Context.SaveChanges();
        c.Holes.Add(h);
        return (h);
    }

    private ClubManager AddNewClubManager(string name, int handi, string mobile,
Gender gender, Course course, string username = "", bool withinmatch = true, Title title
= Title.ClubManager)
    {
        var g3 = new ClubManager() { FullName = name, Handicap = handi, Mobile =
mobile, Gender = gender, course = course, Username = username, Position = title };
        Context.Golfers.Add(g3);
        Context.SaveChanges();
        return (g3);
    }

    private void AddFriend(Golfer golfer1, Golfer golfer2)
    {
        golfer1.Friends.Add(golfer2);
    }

```

```

        Context.SaveChanges();
    }
    private Course EditCourse(Course course, Golfer golfer)
    {
        var c = course;
        c.ClubManager = golfer;
        Context.SaveChanges();
        return c;
    }

    private Facility AddNewFacility(Facilities Facility)
    {
        var F = new Facility() { facility = Facility, Name = Facility.ToString() };
        Context.Facilities.Add(F);
        Context.SaveChanges();
        return F;
    }
    private Group AddNewGroup(string GroupName, Golfer golfer)
    {
        var g = new Group() { Name = GroupName, GroupOwner = golfer };
        Context.Groups.Add(g);
        Context.SaveChanges();
        return g;
    }
}
}

```

Server

NakedObjectsRunsetting

```

namespace NakedObjects.GolfRecord {
    public class NakedObjectsRunSettings
    {

        public static string RestRoot
        {
            get { return ""; }
        }

        private static string[] ModelNamespaces
        {
            get
            {
                return new string[] { "GolfRecord.Model" }; //Add top-level namespace(s)
that cover the domain model
            }
        }

        private static Type[] Types
        {
            get
            {
                return new Type[] {
                    typeof(Stableford),
                    typeof(Strokeplay),
                    typeof(Matchplay),
                    typeof(StrokeplayScores),
                    typeof(StablefordScores),
                }
            }
        }
    }
}

```



```

        typeof(MatchPlayHoleScore),
        typeof(ClubManager),
        typeof(Player),
        typeof(MatchInvitation),
        typeof(FriendInvitation),
        typeof(GroupInvitation),
        typeof(RequestToJoin),
        typeof(PlayerMessage),
        typeof(GroupMessage)
        //You need only register here any domain model types that cannot be
        //'discovered' by the framework when it 'walks the graph' from the
methods
        //defined on services registered below
    };
}
}

private static Type[] Services
{
    get
    {
        return new Type[] {
            typeof(GolferServices),
            typeof(MatchServices),
            typeof(CourseServices),
            typeof(HoleServices),
            typeof(FacilityServices)
        };
    }
}

public static ReflectorConfiguration ReflectorConfig()
{
    return new ReflectorConfiguration(Types, Services, ModelNamespaces,
MainMenus);
}

public static EntityObjectStoreConfiguration EntityObjectStoreConfig()
{
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new
GolfRecordDbContext("NakedObjectsGolfRecord"));
    return config;
}

public static IMenu[] MainMenus(IMenuFactory factory)
{
    return new IMenu[] {
        factory.NewMenu<GolferServices>(true, "Golfers"),
        factory.NewMenu<MatchServices>(true, "Matches"),
        factory.NewMenu<CourseServices>(true, "Courses")
    };
}

public static IAuthorizationConfiguration AuthorizationConfig()
{
    var config = new AuthorizationConfiguration<DefaultAuthorizer>();
    // config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");
    // config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");
}

```

```

        config.AddTypeAuthorizer<Strokeplay, StrokePlayAuthoriser>();
        config.AddTypeAuthorizer<Stableford, StablefordAuthoriser>();
        config.AddTypeAuthorizer<Matchplay, MatchplayAuthoriser>();
        config.AddTypeAuthorizer<Player, PlayerAuthoriser>();
        config.AddTypeAuthorizer<ClubManager, ClubManagerAuthoriser>();
        config.AddTypeAuthorizer<Hole, HoleAuthoriser>();
        config.AddTypeAuthorizer<Match, MatchAuthoriser>();
        config.AddTypeAuthorizer<Group, GroupAuthoriser>();
        config.AddTypeAuthorizer<Course, CourseAuthoriser>();
        config.AddTypeAuthorizer<Invitation, InvitationAuthoriser>();
        config.AddTypeAuthorizer<Message, MessageAuthoriser>();
        return config;
    }
}

Model
Club manger
namespace GolfRecord.Model
{
    public class ClubManager:Golfer
    {
        [Optionally]
        public virtual Course course { get; set; }
    }
}

Club manager Authoriser
namespace GolfRecord.Model
{
    public class ClubManagerAuthoriser : ITypeAuthorizer<ClubManager>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, ClubManager manager, string
memberName)
        {
            if ((manager.Username == principal.Identity.Name) & (memberName == "Course")
                | (memberName == "Username")
                | (memberName == "Position"))
            {
                return false;
            }
            else if (manager.Username == principal.Identity.Name)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public bool IsVisible(IPrincipal principal, ClubManager manager, string
memberName)

```

```

{
    if (memberName == "AddMatch")
    {
        return false;
    }
    else
    if ((GolferServices.Me().FullName == null) & (memberName == "SendMessage"))
    {
        return false;
    }
    else if ((manager != GolferServices.Me()) & ((memberName == "PrivateAccount"
        | (memberName == "CreateNewGroup")
        | (memberName == "CreateNewMatch"))))
    {
        return false;
    }
    else if ((manager.Friends.Contains(GolferServices.Me())) & (memberName ==
"AddFriend"))
    {
        return false;
    }
    else if ((manager.PrivateAccount == true) & (memberName == "Mobile")
        | (memberName == "Username"))
    {
        return false;
    }
    else if (memberName == "AddMatchHistory")
    {
        return false;
    }
    else if (((manager.Friends.Count == 0) & (memberName == "Friends"))
        | ((manager.Groups.Count == 0) & (memberName == "Groups"))
        | ((manager.Invites.Count == 0) & (memberName == "Invites"))
        | ((manager.Messages.Count == 0) & (memberName == "Messages"))
        | ((manager.MyMatches.Count == 0) & (memberName == "MyMatches")))
    {
        return false;
    }
    else if ((manager.Username == principal.Identity.Name) & (memberName ==
"SendMessage"))
    {
        return false;
    }
    else if ((manager.Username == principal.Identity.Name) &
        ((manager.Invites.Count == 0) & ((memberName == "AcceptFriendship")
        | (memberName == "AcceptGroup")
        | (memberName == "AcceptMatch")
        | (memberName == "DeclineInvite")
        | (memberName ==
"AcceptGroupMember"))
        | ((manager.Messages.Count == 0) & (memberName == "DeleteMessage")))
        | ((manager.MyMatches.Count == 0) & (memberName == "MyMatches")))
    {
        return false;
    }
    else if ((manager.Username != principal.Identity.Name) & (memberName ==
"AcceptFriendship"))

```

```

"AcceptGroup")
"AcceptMatch")
"DeclineInvite")
"DeleteMessage")
"AcceptGroupMember"))))
    {
        return false;
    }
    else
    {
        return true;
    }
}
}
}

```

Course

namespace GolfRecord.Model

```

{
    public class Course
    {
        public FacilityServices FacilityServices { set; protected get; }

        [NakedObjectsIgnore]
        public virtual int Id { get; set; }

        [Title]
        public virtual string CourseName { get; set; }

        public virtual string Location { get; set; }

        public virtual string Address { get; set; }

        [Optionally]
        public virtual int Par { get; set; }

        public virtual Golfer ClubManager { get; set; }

        private ICollection<Hole> _Holes = new List<Hole>();
        [Hidden(WhenTo.UntilPersisted)]
        public virtual ICollection<Hole> Holes
        {
            get
            {
                return _Holes;
            }
            set
            {
                _Holes = value;
            }
        }

        [Optionally]
    }
}

```

```
    public virtual string CourseDescription { get; set; } //So people can look at the
course and see whats its li
```

```
[Optionally]
public virtual string WebsiteLink { get; set; } //link to the course website.
```

```
[Optionally]
public virtual int Yardage { get; set; }
```

```
public virtual string PhoneNumber { get; set; }
public string ValidatePhoneNumber(string PhoneNumber)
{
    if ((PhoneNumber.ElementAt(0) == '+') & (PhoneNumber.Length == 13))
    {
        return null;
    }
    else if ((PhoneNumber.ElementAt(0) == '0') & (PhoneNumber.Length == 11))
    {
        return null;
    }
    else
    {
        return ("Incorrect length of phone number please check");
    }
}
```

```
public virtual FileAttachment PhotoOfCourse
{
    get
    {
        if (AttContent == null) return null;
        return new FileAttachment(AttContent, AttName, AttMime);
    }
}
```

```
[NakedObjectsIgnore]
public virtual byte[] AttContent { get; set; }
```

```
[NakedObjectsIgnore]
public virtual string AttName { get; set; }
```

```
[NakedObjectsIgnore]
public virtual string AttMime { get; set; }
```

```
public void AddOrChangeAttachment(FileAttachment newAttachment)
{
    AttContent = newAttachment.GetResourceAsByteArray();
    AttName = newAttachment.Name;
    AttMime = newAttachment.MimeType;
}
```

```
#region Facilities (collection)
private ICollection<Facility> _Facilities = new List<Facility>();
```

```
public virtual ICollection<Facility> Facilities
{
    get
    {
```

```

        return _Facilities;
    }
    set
    {
        _Facilities = value;
    }
}
public void AddFacility(Facility facility)
{
    Facilities.Add(facility);
}
public IQueryable<Facility> AutoCompleteAddFacility([MinLength(2)] string
matching)
{
    return FacilityServices.AllFacilities().Where(f => f.Name.Contains(matching));
}
#endregion
}
}

```

CourseAuthoriser

namespace GolfRecord.Model

```

{
    public class CourseAuthoriser : ITypeAuthorizer<Course>
    {
        public GolferServices GolferServices { get; protected set; }
        public bool IsEditable(IPrincipal principal, Course target, string memberName)
        {
            if (GolferServices.Me() == target.ClubManager)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public bool IsVisible(IPrincipal principal, Course target, string memberName)
        {
            if ((GolferServices.Me() == target.ClubManager) & ((memberName ==
"AddFacility") | (memberName == "AddOrChangeAttachment")))
            {
                return true;
            }
            else if ((memberName == "AddFacility") | (memberName ==
"AddOrChangeAttachment"))
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
}
}

```

CourseServices

namespace GolfRecord.Model

```
{
    public class CourseServices
    {
        public IDomainObjectContainer Container { set; protected get; }

        public Course AddNewCourse()
        {
            return Container.NewTransientInstance<Course>();
        }

        public IQueryable<Course> BrowseCourses()
        {
            return Container.Instances<Course>();
        }
        [NakedObjectsIgnore]
        public virtual int HoleID { get; set; }
    }
}
```

DefaultAuthoriser

namespace GolfRecord.Model

```
{
    public class DefaultAuthorizer : ITypeAuthorizer<object>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, object target, string memberName)
        {
            if ((GolferServices.IsPlayer() == true) & // needs to be changed with
                (
                    (memberName == "Position")
                    | (memberName == "Username")
                    | (typeof(Course).IsAssignableFrom(target.GetType()))
                    | (typeof(Hole).IsAssignableFrom(target.GetType()))
                )
            )
            {
                return false;
            } //nothing is uneditable to club manager except other golfers and winner.
            else if (memberName == "Winner")
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        public bool IsVisible(IPrincipal principal, object target, string memberName)
        {
            if ((memberName == "AddNewCourse") | (memberName == "AddMatchHistory"))
            {
                return false;
            }
        }
    }
}
```

```

        else if ((memberName == "CreateNewMatch") & (GolferServices.Me().FullName ==
null))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}

```

Enums

namespace GolfRecord.Model

```

{
    public class Enums
    {
        public enum MatchType { Strokeplay = 1, Matchplay = 2, Stableford = 3 }
        public enum Gender { Male = 1, Female = 2, Other = 3 }
        public enum Facilities { ClubHouse = 1, ATM = 2, FreeWifi = 3, Restaurant = 4,
ChangingRooms = 5, ProShop = 6, WeddingVenue = 7, ConferenceRoom = 8, Hotel = 9 }
        public enum Title { Player = 1, ClubManager = 2, SystemManager = 3,
UnregisteredGolfer = 4 }
        public enum InviteType { MatchInvite = 1, FriendInvite = 2, GroupInvite = 3,
RequestToJoin = 4 }
    }
}

```

Facility

namespace GolfRecord.Model

```

{
    public class Facility
    {
        [NakedObjectsIgnore]
        public virtual int ID { get; set; }

        [Title]
        public virtual Facilities facility { get; set; }

        [NakedObjectsIgnore]
        public virtual string Name { get; set; }
    }
}

```

FacilityAuthoriser

namespace GolfRecord.Model

```

{
    public class FacilityServices
    {
        public IDomainObjectContainer Container { set; protected get; }
        public IQueryable<Facility> AllFacilites()
        {
            return Container.Instances<Facility>();
        }
    }
}

```



```

    }
}

FriendInvitation
namespace GolfRecord.Model
{
    public class FriendInvitation: Invitation
    {
    }
}

Golfer
namespace GolfRecord.Model
{
    public class Golfer
    {
        //All persisted properties on a domain object must be 'virtual'
        public IDomainObjectContainer Container { set; protected get; }

        public GolferServices GolferServices { set; protected get; }

        public CourseServices CourseServices { set; protected get; }

        public MatchServices MatchServices { set; protected get; }

        [NakedObjectsIgnore]//Indicates that this property will never be seen in the UI
        public virtual int Id { get; set; }

        [Title]
        [MemberOrder(1)]//This property will be used for the object's title at the top of
the view and in a link
        public virtual string FullName { get; set; }

        [MemberOrder(2)] //this property is not neccessary
        public virtual int Handicap { get; set; }

        [MemberOrder(3)][Optionally]
        public virtual string Mobile { get; set; }
        public string ValidateMobile(string Mobile)
        {
            if ((Mobile.ElementAt(0) == '+') & (Mobile.Length == 13))
            {
                return null;
            }
            else if ((Mobile.ElementAt(0) == '0') & (Mobile.Length == 11))
            {
                return null;
            }
            else
            {
                return ("Incorrect length of mobile number please check");
            }
        }

        [Optionally]

```

```

    public virtual Gender Gender { get; set; }

    public virtual Title Position { get; set; }

    public virtual string Username { get; set; }

    public virtual bool PrivateAccount { get; set; }

    public Match CreateNewMatch(MatchType matchtype)
    {
        var m = MatchServices.CreateNewMatch(matchtype);
        return m;
    }

    #region Friends (collection)
    private ICollection<Golfer> _Friends = new List<Golfer>();

    public virtual ICollection<Golfer> Friends
    {
        get
        {
            return _Friends;
        }
        set
        {
            _Friends = value;
        }
    }

    public void AddFriend(Golfer golfer)
    {
        var invite = Container.NewTransientInstance<FriendInvitation>();
        invite.Sender = GolferServices.Me();
        invite.Receiver = golfer;
        invite.inviteType = InviteType.FriendInvite;
        Container.Persist(ref invite);
        golfer.Invites.Add(invite);
    }

    [PageSize(3)]
    public IQueryable<Golfer> AutoCompleteAddFriend([MinLength(2)] string
matching)
    {
        return GolferServices.AllGolfers().Where(g =>
(g.FullName.Contains(matching)));
        //& (g.Friends.Contains(GolferServices.Me()))));
    }
    #endregion

    #region MyMatches (collection)
    private ICollection<Match> _MyMatches = new List<Match>();

    public virtual ICollection<Match> MyMatches
    {
        get
        {
            return _MyMatches;
        }
    }

```

```

        set
        {
            _MyMatches = value;
        }
    }
    public void AddMatch(Match match)
    {
        MyMatches.Add(match);
    }
#endregion

#region Groups
public Group CreateNewGroup()
{
    var group = Container.NewTransientInstance<Group>();
    group.GroupOwner = GolferServices.Me();
    group.Name = (GolferServices.Me().FullName + "'s group");
    Container.Persist(ref group);
    GolferServices.Me().Groups.Add(group);
    return group;
}

private ICollection<Group> _Groups = new List<Group>();

public virtual ICollection<Group> Groups
{
    get
    {
        return _Groups;
    }
    set
    {
        _Groups = value;
    }
}
#endregion

#region Invites
private ICollection<Invitation> _Invites = new List<Invitation>();

public virtual ICollection<Invitation> Invites
{
    get
    {
        return _Invites;
    }
    set
    {
        _Invites = value;
    }
}

public void DeclineInvite(Invitation invite)
{
    Container.DisposeInstance(invite);
}

```

```

public void AcceptFriendship(FriendInvitation invite)
{
    if (invite.Sender.Friends.Contains(invite.Receiver))
    {
        Container.DisposeInstance(invite);
    }
    else
    {
        invite.Sender.Friends.Add(invite.Receiver);
        invite.Receiver.Friends.Add(invite.Sender);
        Container.DisposeInstance(invite);
    }
}

public string ValidateAcceptFriendship(Invitation invite)
{
    if (invite.inviteType == InviteType.FriendInvite)
    {
        if (invite.Receiver.Friends.Contains(invite.Sender))
        {
            return ("You are already friends with this person please delete the
invite");
        }
        else
        {
            return null;
        }
    }
    else
    {
        return ("This is not an Friend Invitation");
    }
}

public bool HideAcceptFriendship()
{
    int AmountOfInvites = 0;
    for (int i = 0; i < Invites.Count; i++)
    {
        if (Invites.ElementAt(i).inviteType == InviteType.FriendInvite)
        {
            AmountOfInvites += 1;
        }
    }
    return AmountOfInvites == 0;
}

public void AcceptGroupMember(RequestToJoin invite)
{
    if (invite.group.Members.Contains(invite.Sender))
    {
        Container.DisposeInstance(invite);
    }
    else
    {
        invite.group.Members.Add(invite.Sender);
        Container.DisposeInstance(invite);
    }
}

```

```

    }
    public bool HideAcceptGroupMember()
    {
        int AmountOfInvites = 0;
        for (int i = 0; i < Invites.Count; i++)
        {
            if (Invites.ElementAt(i).inviteType == InviteType.RequestToJoin)
            {
                AmountOfInvites += 1;
            }
        }
        return AmountOfInvites == 0;
    }

    public void AcceptGroup(GroupInvitation invite)
    {
        if (invite.group.Members.Contains(invite.Receiver))
        {
            Container.DisposeInstance(invite);
        }
        else
        {
            invite.group.Members.Add(invite.Receiver);
            Container.DisposeInstance(invite);
        }
    }
    public bool HideAcceptGroup()
    {
        int AmountOfInvites = 0;
        for (int i = 0; i < Invites.Count; i++)
        {
            if (Invites.ElementAt(i).inviteType == InviteType.GroupInvite)
            {
                AmountOfInvites += 1;
            }
        }
        return AmountOfInvites == 0;
    }

    public void AcceptMatch(MatchInvitation invite)
    {
        if ((invite.match.MatchType == MatchType.Matchplay) &
(invite.match.Golfers.Count == 2))
        {
            PlayerMessage msg = Container.NewTransientInstance<PlayerMessage>();
            msg.Content = "This match is already full";
            Container.Persist(ref msg);
            Messages.Add(msg);
        }
        else if (invite.match.Golfers.Count == 4)
        {
            var msg = Container.NewTransientInstance<PlayerMessage>();
            msg.Content = "This match is already full";
            Container.Persist(ref msg);
            Messages.Add(msg);
        }
    }

```

```

        else if (invite.match.Golfers.Contains(invite.Receiver))
        {
            var msg = Container.NewTransientInstance<PlayerMessage>();
            msg.Content = "You are already appart of this match";
            Container.Persist(ref msg);
            Messages.Add(msg);
        }
        {
            invite.match.Golfers.Add(invite.Receiver);
        }

        Container.DisposeInstance(invite);
    }
    public bool HideAcceptMatch()
    {
        int AmountOfInvites = 0;
        for (int i = 0; i < Invites.Count; i++)
        {
            if (Invites.ElementAt(i).inviteType == InviteType.MatchInvite)
            {
                AmountOfInvites += 1;
            }
        }
        return AmountOfInvites == 0;
    }
}
#endregion

#region Messages
private ICollection<PlayerMessage> _Messages = new List<PlayerMessage>();

public virtual ICollection<PlayerMessage> Messages
{
    get
    {
        return _Messages;
    }
    set
    {
        _Messages = value;
    }
}

public PlayerMessage SendMessage()
{
    PlayerMessage m = null;
    m = Container.NewTransientInstance<PlayerMessage>();
    m.Reciever = this;
    m.Sender = GolferServices.Me();
    m.SendersName = m.Sender.FullName;
    m.Content = ("Please Press Edit To Change");
    Container.Persist(ref m);
    m.Reciever.Messages.Add(m);
    return m;
}

```

```

        public void DeleteMessage(PlayerMessage m)
        {
            Container.DisposeInstance(m);
        }
    }
}

GolferServices
namespace GolfRecord.Model
{
    public class GolferServices
    {
        #region Injected Services
        //An implementation of this interface is injected automatically by the framework
        public IDomainObjectContainer Container { set; protected get; }
        #endregion

        public IQueryable<Golfer> AllGolfers()
        {
            return Container.Instances<Golfer>();
        }

        public IQueryable<ClubManager> AllManagers()
        {
            return Container.Instances<ClubManager>();
        }

        public Golfer Me()
        {
            var username = Container.Principal.Identity.Name;
            var user = AllGolfers().Where(g =>
g.Username.ToUpper().Contains(username.ToUpper())).SingleOrDefault();
            if (user == null)
            {
                user = Container.NewTransientInstance<Player>();
                user.Username = Container.Principal.Identity.Name;
                user.Position = Title.Player;
                return user;
            }
            else
            {
                return user;
            }
        }

        [NakedObjectsIgnore]
        public bool IsPlayer()
        {
            if (Me() != null)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}

```

```

    }
}

public IQueryable<Golfer> FindGolferByName(string name)
{
    return AllGolfers().Where(c => c.FullName.ToUpper().Contains(name.ToUpper()));
}

public IQueryable<Group> AllGroups()
{
    return Container.Instances<Group>();
}
}

}

Group
namespace GolfRecord.Model
{
    public class Group
    {
        public IDomainObjectContainer Container { set; protected get; }

        public GolferServices GolferServices { set; protected get; }

        [NakedObjectsIgnore]
        public virtual int Id { get; set; }

        [Title][MemberOrder(1)]
        public virtual string Name { get; set; }

        public virtual Golfer GroupOwner { get; set; }

        #region GroupMembers (collection)
        private ICollection<Golfer> _Members = new List<Golfer>();
        [Hidden(WhenTo.UntilPersisted)]
        public virtual ICollection<Golfer> Members
        {
            get
            {
                return _Members;
            }
            set
            {
                _Members = value;
            }
        }
        [PageSize(3)]
        public IQueryable<Golfer> AutoComplete0AddNewMember([MinLength(3)] string name)
        {
            return GolferServices.AllGolfers().Where(g => g.FullName.Contains(name));
        }

        public void AddNewMember(Golfer golfer)
        {
            var invite = Container.NewTransientInstance<GroupInvitation>();
            invite.group = this;
            invite.Sender = GolferServices.Me();
        }
    }
}

```



```

        invite.Receiver = golfer;
        invite.inviteType = InviteType.GroupInvite;
        Container.Persist(ref invite);
        golfer.Invites.Add(invite);
    }

    public void RequestToJoin()
    {
        var invite = Container.NewTransientInstance<RequestToJoin>();
        invite.group = this;
        invite.Sender = GolferServices.Me();
        invite.Receiver = GroupOwner;
        invite.inviteType = InviteType.RequestToJoin;
        Container.Persist(ref invite);
        GroupOwner.Invites.Add(invite);
    }
    #endregion

    #region GroupMessages
    private ICollection<GroupMessage> _Messages = new List<GroupMessage>();

    public virtual ICollection<GroupMessage> Messages
    {
        get
        {
            return _Messages;
        }
        set
        {
            _Messages = value;
        }
    }

    public GroupMessage SendGroupMessage()
    {
        GroupMessage m = Container.NewTransientInstance<GroupMessage>();
        m.Sender = GolferServices.Me();
        m.SendersName = GolferServices.Me().FullName;
        m.Content = ("Please press edit to enter your response.");
        Container.Persist(ref m);
        Messages.Add(m);
        return m;
    }

    #endregion
}

GroupAuthoriser
namespace GolfRecord.Model
{
    public class GroupAuthoriser : ITypeAuthorizer<Group>
    {
        public GolferServices GolferServices { set; protected get; }
    }
}

```

```

public bool IsEditable(IPrincipal principal, Group target, string memberName)
{
    if (target.GroupOwner == GolferServices.Me())
    {
        return true;
    }
    else
    {
        return false;
    }
}

public bool IsVisible(IPrincipal principal, Group target, string memberName)
{
    if ((GolferServices.Me().FullName == null) & ((memberName == "RequestToJoin")
| (memberName == "Messages" ) |(memberName == "AddNewMember") | (memberName ==
"SendGroupMessage"))))
    {
        return false;
    }
    if (((target.GroupOwner == GolferServices.Me()) |
(target.Members.Contains(GolferServices.Me())) & (memberName == "RequestToJoin"))
    {
        return false;
    }
    else if (target.Members.Contains(GolferServices.Me()) == true)
    {
        return true;
    }
    else if ((target.Members.Contains(GolferServices.Me()) == false) &
((memberName == "Messages") | (memberName == "AddNewMember") | (memberName ==
"SendGroupMessage"))))
    {
        return false;
    }
    else
    {
        return true;
    }
}
}
}

```

GroupInvitation

namespace GolfRecord.Model

```

{
    public class GroupInvitation:Invitation
    {
        public IDomainObjectContainer Container;
        public virtual Group group { get; set; }
    }
}

```

GroupMessage

namespace GolfRecord.Model

```

{
    public class GroupMessage : Message
    {
        public virtual Group Group { get; set; }
    }
}
Hole
namespace GolfRecord.Model
{
    public class Hole
    {
        [NakedObjectsIgnore]
        public virtual int Id { get; set; }

        [Title][MemberOrder(1)]
        public virtual int HoleNumber { get; set; }
        public string ValidateHoleNumber(int h)
        {
            if (h == 0)
            {
                return "Holenumber Must me larger than 0 & not repeated";
            }
            return null;
        }

        [MemberOrder(5)]
        public virtual int Par { get; set; }
        public string ValidatePar(int par)
        {
            if (par == 0)
            {
                return "Par Must me larger than 0";
            }
            return null;
        }

        [MemberOrder(6)]
        public virtual int StrokeIndex { get; set; }
        public string ValidateStrokeIndex(int strokeindex)
        {
            if (strokeindex == 0)
            {
                return "StrokeIndex Must me larger than 0";
            }
            return null;
        }

        [MemberOrder(2)]
        public virtual string Name { get; set; }

        [MemberOrder(3)]
        public virtual int WhiteYards { get; set; }
        public string ValidateWhiteYards(int WhiteYards)
        {
            if (WhiteYards == 0)

```

```

        {
            return "WhiteYards Must me larger than 0";
        }
        return null;
    }

[MemberOrder(4)]
public virtual int YellowYards { get; set; }
public string ValidateYellowYards(int YellowYards)
{
    if (YellowYards == 0)
    {
        return "YellowYards Must me larger than 0";
    }
    return null;
}

[MemberOrder(7)]
public virtual int LadiesRedYards { get; set; }
public string ValidateLadiesRedYards(int RedYards)
{
    if (RedYards == 0)
    {
        return "RedYards Must me larger than 0";
    }
    return null;
}

[MemberOrder(8)]
public virtual int RedPar { get; set; }
public string ValidateRedPar(int par)
{
    if (Par == 0)
    {
        return "Par Must me larger than 0";
    }
    return null;
}

[MemberOrder(9)]
public virtual int RedStrokeIndex { get; set; }
public string ValidateRedStrokeIndex(int RedStrokeIndex)
{
    if (RedStrokeIndex == 0)
    {
        return "StrokeIndex Must me larger than 0";
    }
    return null;
}

public virtual Course Course { get; set; }

public virtual FileAttachment PhotoOfHole
{
    get
    {
        if (AttContent == null) return null;
    }
}

```

```

        return new FileAttachment(AttContent, AttName, AttMime);
    }
}

[NakedObjectsIgnore]
public virtual byte[] AttContent { get; set; }

[NakedObjectsIgnore]
public virtual string AttName { get; set; }

[NakedObjectsIgnore]
public virtual string AttMime { get; set; }

public void AddOrChangeAttachment(FileAttachment newAttachment)
{
    AttContent = newAttachment.GetResourceAsByteArray();
    AttName = newAttachment.Name;
    AttMime = newAttachment.MimeType;
}
}

HoleAuthoriser
namespace GolfRecord.Model
{
    public class HoleAuthorier : ITypeAuthorizer<Hole>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, Hole target, string memberName)
        {
            if (target.Course.ClubManager == GolferServices.Me())
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public bool IsVisible(IPrincipal principal, Hole target, string memberName)
        {
            if ((GolferServices.Me().Position == Enums.Title.ClubManager) & (memberName
== "AddOrChangeAttachment"))
            {
                return true;
            }
            if (((target.LadiesRedYards == 0) & (memberName == "LadiesRedYards"))
| ((target.Name == null) & (memberName == "Name"))
| ((target.RedPar == 0) & (memberName == "RedPar"))
| ((target.RedStrokeIndex == 0) & (memberName == "RedStrokeIndex"))
| ((target.WhiteYards == 0) & (memberName == "WhiteYards"))
| ((target.YellowYards == 0) & (memberName == "YellowYards"))
| (memberName == "AddOrChangeAttachment"))
            {
                return false;
            }
        }
    }
}

```

```

        }
        else
        {
            return true;
        }
    }
}

```

HoleScoreAbstract

```

namespace GolfRecord.Model
{
    public abstract class HoleScoreAbstract
    {
        [NakedObjectsIgnore]
        public virtual int ID { get; set; }

        [NakedObjectsIgnore]
        public virtual int HoleId { get; set; }

        [Title][MemberOrder(1)]
        public virtual Hole Hole { get; set; }
    }
}

```

HoleServices

```

namespace GolfRecord.Model
{
    public class HoleServices
    {
        public IDomainObjectContainer Container { set; protected get; }

        public Hole AddNewHole()
        {
            return Container.NewTransientInstance<Hole>();
        }

        public IQueryable<Hole> ShowHoles()
        {
            return Container.Instances<Hole>();
        }
    }
}

```

Invitation

```

namespace GolfRecord.Model
{
    public class Invitation
    {
        [NakedObjectsIgnore]
        public virtual int Id { get; set; }

        public virtual Golfer Sender { get; set; }

        public virtual Golfer Receiver { get; set; }
    }
}

```

```

        [Title]
        public virtual InviteType inviteType { get; set; }
    }
}

Match
namespace GolfRecord.Model
{
    public class Match
    {
        #region InjectedServices

        public IDomainObjectContainer Container { set; protected get; }

        public CourseServices CourseServices { set; protected get; }

        public GolferServices GolferServices { set; protected get; }

        public MatchServices MatchServices { set; protected get; }

        #endregion
        [NakedObjectsIgnore]
        public virtual int ID { get; set; }

        [Title]
        public virtual string MatchName { get; set; }

        public virtual DateTime DateOfMatch { get; set; }
        public string ValidateDateOfMatch(DateTime d)
        {
            if (( d.IsAfterToday() )| (d.IsToday() ))
            {
                return "Must be Today or after Today";
            }
            return null;
        }

        public virtual bool MatchOver { get; set; }

        [NakedObjectsIgnore]
        public virtual int CourseID { get; set; }

        public virtual Course Course { get; set; }

        public virtual Golfer MatchCreator { get; set; }

        [PageSize(3)]
        public IQueryable<Course> AutoCompleteCourse([MinLength(2)] string matching)
        {
            return CourseServices.BrowseCourses().Where(c =>
c.CourseName.Contains(matching));
        }

        [Title]
        public virtual MatchType MatchType { get; set; }

        [Optionally]
        [Hidden(WhenTo.UntilPersisted)]

```

```

public virtual string DescriptionOfMatch { get; set; }

[Optionally]
[Hidden(WhenTo.UntilPersisted)]
public virtual Golfer Winner { get; set; }

#region Add Golfers
public void sendInvite(Golfer golfer)
{
    if (Golfers.Contains(MatchCreator) == false)
    {
        Golfers.Add(MatchCreator);
        MatchCreator.MyMatches.Add(this);
    }
    var invite = Container.NewTransientInstance<MatchInvitation>();
    invite.match = this;
    invite.Sender = GolferServices.Me();
    invite.Receiver = golfer;
    invite.inviteType = InviteType.MatchInvite;
    Container.Persist(ref invite);
    golfer.Invites.Add(invite);
}

public UnRegisteredGolfer AddUnregisteredGolfer(string name, int handicap, Gender
gender)
{
    var UnregisteredGolfer =
Container.NewTransientInstance<UnRegisteredGolfer>();
    UnregisteredGolfer.FullName = name;
    UnregisteredGolfer.Handicap = handicap;
    UnregisteredGolfer.Gender = gender;
    UnregisteredGolfer.GolferCreator = GolferServices.Me();
    Container.Persist(ref UnregisteredGolfer);
    Golfers.Add(UnregisteredGolfer);
    return UnregisteredGolfer;
}

public bool HideSendInvite()
{
    if (MatchType == MatchType.Matchplay)
    {
        return Golfers.Count > 1;
    }
    else
    {
        return Golfers.Count > 3;
    }
}

public bool HideAddUnregisteredGolfer()
{
    if (MatchType == MatchType.Matchplay)
    {
        return Golfers.Count > 1;
    }
    else
    {
        return Golfers.Count > 3;
    }
}

```



```

    }
}
public bool HideAddScores()
{
    if (HoleScores.Count == Course.Holes.Count)
    {
        return false;
    }
    else
    if (MatchType == MatchType.Matchplay)
    {
        return Golfers.Count != 2;
    }
    else
    {
        return Golfers.Count != 4;
    }
}

[PageSize(3)]
public IQueryable<Golfer> AutoComplete0SendInvite([MinLength(2)] string name)
{
    return GolferServices.AllGolfers().Where(g => g.FullName.Contains(name));
}
#endregion
#region Golfers (collection)
private ICollection<Golfer> _Golfers = new List<Golfer>();
[Hidden(WhenTo.UntilPersisted)]
public virtual ICollection<Golfer> Golfers
{
    get
    {
        return _Golfers;
    }
    set
    {
        _Golfers = value;
    }
}
#endregion

#region HoleScores
private ICollection<HoleScoreAbstract> _HoleScores = new
List<HoleScoreAbstract>();
[Hidden(WhenTo.UntilPersisted)]
public virtual ICollection<HoleScoreAbstract> HoleScores
{
    get
    {
        return _HoleScores;
    }
    set
    {
        _HoleScores = value;
    }
}
public IList<Hole> Choices0AddScores()
{

```

```

        if (HoleScores.Count == 0)
        {
            return Course.Holes.ToList();
        }
        else
        {
            // return Course.Holes.ToList();
            return (from h in Course.Holes
                    from s in HoleScores
                    where h.Id != s.HoleId //a query across two sources.
                    select h).ToList();
        }
    }
}
public Hole DefaultAddScores()
{
    int nextHole = 1;
    if (HoleScores.Count > 0)
    {
        nextHole = HoleScores.Max(hs => hs.Hole.HoleNumber) + 1;
    }
    return Course.Holes.First(h => h.HoleNumber == nextHole);
}

#endregion

public void AddPostMatchDescription(string Description)
{
    DescriptionOfMatch = Description;
}

}

}

MatchAuthoriser
namespace GolfRecord.Model
{
    public class MatchAuthoriser : ITypeAuthorizer<Match>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, Match match, string memberName)
        {
            if ((match.Golfers.Contains(GolferServices.Me())) | (match.MatchCreator ==
GolferServices.Me()))
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public bool IsVisible(IPrincipal principal, Match match, string memberName)

```

```

    {
        if ((memberName == "CreateNewMatch") & !(principal.Identity.Name ==
GolferServices.Me().Username))
        {
            return false;
        }
        else if (match.Golfers.Contains(GolferServices.Me()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

MatchInvitation

namespace GolfRecord.Model

```

{
    public class MatchInvitation : Invitation
    {
        public IDomainObjectContainer Container;

        public virtual Match match { get; set; }
    }
}

```

MatchPlay

namespace GolfRecord.Model

```

{
    public class Matchplay : Match //normally have 2 people playing unless its four ball
better ball teams of 2.
    {

        [NakedObjectsIgnore]
        public virtual int TotalScoreA { get; set; }

        [NakedObjectsIgnore]

        public virtual int TotalScoreB { get; set; }


        public int[] Handicaps = new int[2];
        public int[] DifficultyPerGolfer = new int[2];

        public void AddScores(Hole hole, int ScoreA, int ScoreB)
        {
            var hs = Container.NewTransientInstance<MatchPlayHoleScore>();
            int[] Scores = { ScoreA, ScoreB };
            hs.GolferARawScore = ScoreA;
            hs.GolferBRawScore = ScoreB;
            int[] ModifiedScore = StrokeIndexandHandicapEffectOnScore(hole, Scores);
            ScoreCalculation(hole, ModifiedScore, hs );
            Container.Persist(ref hs);
        }
    }
}

```

```

        HoleScores.Add(hs);
        if (hole.HoleNumber == Course.Holes.Count)
        {
            findWinnerMatchPlay();
        }
    }

    private int[] StrokeIndexandHandicapEffectOnScore(Hole hole, int[] Scores)
    {
        int[] ModifiedScore = new int[4];
        for (int i = 0; i < 2; i++)
        {
            if ((Golfers.ElementAt(i).Gender == Gender.Female) & (hole.RedStrokeIndex
!= 0))
            {
                if (Golfers.ElementAt(i).Handicap >= hole.RedStrokeIndex)
                {
                    if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex + 18))
& (Course.Holes.Count == 18))
                    {
                        ModifiedScore[i] = Scores[i] - 2;
                    }
                    else if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex +
9)) & (Course.Holes.Count == 9))
                    {
                        ModifiedScore[i] = Scores[i] - 2;
                    }

                    else
                    {
                        ModifiedScore[i] = Scores[i] - 1;
                    }
                }
                else
                {
                    ModifiedScore[i] = Scores[i];
                }
            }
            else
            {
                if (Golfers.ElementAt(i).Handicap >= hole.StrokeIndex)
                {
                    if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex + 18)) &
(Course.Holes.Count == 18))
                    {
                        ModifiedScore[i] = Scores[i] - 2;
                    }
                    else if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex +
9)) & (Course.Holes.Count == 9))
                    {
                        ModifiedScore[i] = Scores[i] - 2;
                    }
                    else
                    {
                        ModifiedScore[i] = Scores[i] - 1;
                    }
                }
            }
        }
    }

```

```

        else
        {
            ModifiedScore[i] = Scores[i];
        }
    }

    }
    return ModifiedScore;
}

[NakedObjectsIgnore]
public void ScoreCalculation(Hole hole, int[] Scores, MatchPlayHoleScore hs)
{

    if (Scores[0] < Scores[1])
    {
        hs.HoleWinner = Golfers.ElementAt(0);
        TotalScoreA += 1;
    }
    else if (Scores[1] < Scores[0])
    {
        hs.HoleWinner = Golfers.ElementAt(1);
        TotalScoreB += 1;
    }
    else // if there is a draw dont add a score.
    {
    }
    hs.Hole = hole;
    hs.GolferATotalScore = TotalScoreA;
    hs.GolferBTotalScore = TotalScoreB;
    hs.GolferAActualScore = Scores[0];
    hs.GolferBActualScore = Scores[1];
}

[NakedObjectsIgnore]
public void findWinnerMatchPlay()
{
    if (TotalScoreA > TotalScoreB)
    {
        Winner = Golfers.ElementAt(0);
    }
    else if (TotalScoreB > TotalScoreA)
    {
        Winner = Golfers.ElementAt(1);
    }
    MatchOver = true;
}
}
}

MatchPlayAuthoriser
namespace GolfRecord.Model
{
    public class MatchplayAuthoriser : ITypeAuthorizer<Matchplay>
    {

```

```

    public GolferServices GolferServices { set; protected get; }

    public bool IsEditable(IPrincipal principal, Matchplay match, string memberName)
    {
        if ((memberName == "Winner") | (memberName == "MatchType") | (memberName ==
"MatchCreator") | (memberName == "MatchOver"))
        {
            return false;
        }
        else if ((match.Golfers.Contains(GolferServices.Me())) | (match.MatchCreator
== GolferServices.Me()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public bool IsVisible(IPrincipal principal, Matchplay match, string memberName)
    {
        if ((match.Golfers.Contains(GolferServices.Me()) == false) & ((memberName ==
"AddScores") | (memberName == "DescriptionOfMatch") | (memberName ==
"AddPostMatchDescription")))
        {
            return false;
        }
        else if ((match.Golfers.Contains(GolferServices.Me())) & (match.MatchOver ==
true) & (memberName == "DescriptionOfMatch"))
        {
            return true;
        }
        else if ((match.MatchOver == true) & (memberName == "AddScores"))
        {
            return false;
        }
        else if ((match.MatchOver == false) & ((memberName == "DescriptionOfMatch") |
(memberName == "AddPostMatchDescription")))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}

MatchplayHoleScore
namespace GolfRecord.Model
{
    public class MatchPlayHoleScore:HoleScoreAbstract
    {

        [MemberOrder(2)]

```

```

        public virtual int GolferARawScore { get; set; }

        [MemberOrder(5)]
        public virtual int GolferBRawScore { get; set; }

        [MemberOrder(3)]
        public virtual int GolferAActualScore { get; set; }

        [MemberOrder(6)]
        public virtual int GolferBActualScore { get; set; }

        [MemberOrder(4)]
        public virtual int GolferATotalScore { get; set; }

        [MemberOrder(7)]
        public virtual int GolferBTotalScore { get; set; }

        public virtual Golfer HoleWinner { get; set; }
    }
}

MatchServices
namespace GolfRecord.Model
{
    public class MatchServices
    {
        public GolferServices GolferServices { set; protected get; }
        public IDomainObjectContainer Container { set; protected get; }

        public Match CreateNewMatch(MatchType matchtype)
        {
            Match match = null;
            switch (matchtype)
            {
                case MatchType.Strokeplay:
                    match = Container.NewTransientInstance<Strokeplay>();
                    match.MatchType = MatchType.Strokeplay;
                    break;
                case MatchType.Matchplay:
                    match = Container.NewTransientInstance<Matchplay>();
                    match.MatchType = MatchType.Matchplay;
                    break;
                case MatchType.Stableford:
                    match = Container.NewTransientInstance<Stableford>();
                    match.MatchType = MatchType.Stableford;
                    break;
                default:
                    break;
            }
            match.MatchCreator = GolferServices.Me();
            return match;
        }

        public IQueryable<Match> ShowMatches()
        {
            return Container.Instances<Match>();
        }
    }
}

```

```

    }
}
Message

namespace GolfRecord.Model
{
    public class Message
    {
        public IDomainObjectContainer Container { set; protected get; }

        [NakedObjectsIgnore]
        public virtual int ID { get; set; }

        [Title][MemberOrder(1)]
        public virtual string SendersName { get; set; }

        public virtual Golfer Sender { get; set; }

        public virtual string Content { get; set; }

        [Optionally]
        public virtual FileAttachment Attachment
        {
            get
            {
                if (AttContent == null) return null;
                return new FileAttachment(AttContent, AttName, AttMime);
            }
        }

        [NakedObjectsIgnore]
        public virtual byte[] AttContent { get; set; }

        [NakedObjectsIgnore]
        public virtual string AttName { get; set; }

        [NakedObjectsIgnore]
        public virtual string AttMime { get; set; }

        public void AddOrChangeAttachment(FileAttachment newAttachment)
        {
            AttContent = newAttachment.GetResourceAsByteArray();
            AttName = newAttachment.Name;
            AttMime = newAttachment.MimeType;
        }

    }
}

```

MessageAuthoriser

```

namespace GolfRecord.Model
{
    public class MessageAuthoriser : ITypeAuthorizer<Message>
    {
        public bool IsEditable(IPrincipal principal, Message target, string memberName)
    }
}

```



```

    {
        if (target.Sender.Username == principal.Identity.Name)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public bool IsVisible(IPrincipal principal, Message target, string memberName)
    {
        if ((target.Sender.Username == principal.Identity.Name) & (memberName ==
"RespondToMessage"))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}

}
Player
namespace GolfRecord.Model
{
    public class Player :Golfer
    {
        #region FavouriteCourse
        private ICollection<Course> _Favourites = new List<Course>();
        [Hidden(WhenTo.UntilPersisted)]
        public virtual ICollection<Course> FavouriteCourses
        {
            get
            {
                return _Favourites;
            }
            set
            {
                _Favourites = value;
            }
        }

        public void AddCourseToFavourites(Course course)
        {
            FavouriteCourses.Add(course);
        }

        [PageSize(3)]
        public IQueryable<Course> AutoComplete0AddCourseToFavourites([MinLength(2)]
string matching)
        {
            return CourseServices.BrowseCourses().Where(c =>
c.CourseName.Contains(matching));

```

```

    }
    #endregion
}
}
PlayerAuthoriser
namespace GolfRecord.Model
{
    public class PlayerAuthoriser : ITypeAuthorizer<Player>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, Player player, string memberName)
        {
            if (player.Username == principal.Identity.Name)
            {
                if ((memberName == "Position" ) | (memberName == "Username"))
                {
                    return false;
                }
                return true;
            }
            else
            {
                return false;
            }
        }

        public bool IsVisible(IPrincipal principal, Player player, string memberName)
        {
            if (memberName == "AddMatch")
            {
                return false;
            }
            else
            {
                if ((GolferServices.Me().FullName == null) & ((memberName == "SendMessage")
| (memberName == "AddMatch")))
                {
                    return false;
                }
                else
                {
                    if ((player != GolferServices.Me()) & ((memberName == "PrivateAccount")
| (memberName ==
"CreateNewGroup")
| (memberName ==
"CreateNewMatch"))))
                    {
                        return false;
                    }
                    else if ((player.Username == GolferServices.Me().Username ) &
((player.PrivateAccount == true) & (memberName == "Mobile")
| (memberName == "Username")))
                    {
                        return true;
                    }
                    else if ((player.PrivateAccount == true) & ((memberName == "Mobile")
| (memberName == "Username")))
                    {

```

```

        return false;
    }
    else if (memberName == "AddMatchHistory")
    {
        return false;
    }
    else if (((player.Friends.Count == 0) & (memberName == "Friends"))
        | ((player.Groups.Count == 0) & (memberName == "Groups"))
        | ((player.Invites.Count == 0) & (memberName == "Invites"))
        | ((player.Messages.Count == 0) & (memberName == "Messages"))
        | ((player.MyMatches.Count == 0) & (memberName == "MyMatches")))

    {
        return false;
    }
    else if ((player.Username == principal.Identity.Name) & (memberName ==
"SendMessage"))
    {
        return false;
    }
    else if ((player.Username == principal.Identity.Name) &
        ((player.Invites.Count == 0) & ((memberName == "AcceptFriendship")
            | (memberName == "AcceptGroup")
            | (memberName ==
"AcceptGroupMember")
                | (memberName == "AcceptMatch")
                | (memberName == "DeclineInvite")))
        | ((player.Messages.Count == 0) & (memberName == "DeleteMessage")))
        | ((player.MyMatches.Count == 0) & (memberName == "MyMatches"))
        | ((player.FavouriteCourses.Count == 0) & (memberName ==
"FavouriteCourses"))))
    {
        return false;
    }
    else if ((player.Friends.Contains(GolferServices.Me())) & (memberName ==
"AddFriend"))
    {
        return false;
    }
    else if ((player.Username != principal.Identity.Name) & ((memberName ==
"AcceptFriendship")
        | (memberName ==
"AcceptGroup")
        | (memberName ==
"AcceptGroupMember")
        | (memberName ==
"AcceptMatch")
        | (memberName ==
"DeclineInvite")
        | (memberName ==
"DeleteMessage"))))
    {
        return false;
    }
    else if (player.Username == principal.Identity.Name)
    {
        return true;
    }
}

```

```

        else
        {
            if ((memberName == "AddFriend") | (memberName ==
"AddCourseToFavourites"))
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
}

PlayerMessages
namespace GolfRecord.Model
{
    public class PlayerMessage :Message
    {
        public virtual Golfer Reciever { get; set; }

        public PlayerMessage RespondToMessage()
        {
            PlayerMessage m = Container.NewTransientInstance<PlayerMessage>();
            m.Reciever = Sender;
            m.Sender = Reciever;
            m.SendersName = Reciever.FullName;
            m.Content = ("Please press edit to enter your response.");
            Container.Persist(ref m);
            m.Reciever.Messages.Add(this);
            return m;
        }
    }
}

RequestToJoin
namespace GolfRecord.Model
{
    public class RequestToJoin: Invitation
    {
        public virtual Group group { get; set; }
    }
}

Stableford
namespace GolfRecord.Model
{
    public class Stableford : Match
    {
        [NakedObjectsIgnore]
        public virtual int TotalScoreA { get; set; }

        [NakedObjectsIgnore]

```

```

public virtual int TotalScoreB { get; set; }

[NakedObjectsIgnore]
public virtual int TotalScoreC { get; set; }

[NakedObjectsIgnore]
public virtual int TotalScoreD { get; set; }

public void AddScores(Hole hole, int ScoreA, int ScoreB, int ScoreC, int ScoreD)
{
    var hs = Container.NewTransientInstance<StablefordScores>();
    Container.Persist(ref hs);
    hs.GolferARawScore = ScoreA;
    hs.GolferBRawScore = ScoreB;
    hs.GolferCRawScore = ScoreC;
    hs.GolferDRawScore = ScoreD;
    hs.Hole = hole;
    int[] Pars = StrokeIndexandHandicapEffectonPar(hole);
    int[] Scores = { ScoreA, ScoreB, ScoreC, ScoreD };
    TotalScoreCalculated(hole, Scores, hs, Pars);
    Container.Persist(ref hs);
    HoleScores.Add(hs);
    if (hole.HoleNumber == Course.Holes.Count)
    {
        FindWinner();
    }
}

public string ValidateAddScores(Hole hole, int A, int B, int C, int D)
{
    if ((A <= 0) | (B <= 0) | (C <= 0) | (D <= 0))
    {
        return "A score can not be negative or 0";
    }
    else
    {
        return null;
    }
}

private int[] StrokeIndexandHandicapEffectonPar(Hole hole)
{
    int[] ModifiedPar = new int[4];
    for (int i = 0; i < 4; i++)
    {
        if ((Golfers.ElementAt(i).Gender == Enums.Gender.Female) &
(hole.RedStrokeIndex != 0))
        {
            if (Golfers.ElementAt(i).Handicap >= hole.RedStrokeIndex)
            {
                if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex + 18))
& (Course.Holes.Count == 18))
                {
                    ModifiedPar[i] = hole.RedPar + 2;
                }
                else if ((Golfers.ElementAt(i).Handicap >= (hole.RedStrokeIndex +
9)) & (Course.Holes.Count == 9))
                {
                    ModifiedPar[i] = hole.RedPar + 2;
                }
            }
        }
    }
}

```

```

        else
        {
            ModifiedPar[i] = hole.RedPar + 1;
        }
    }
    else
    {
        ModifiedPar[i] = hole.RedPar;
    }
}
else
{
    if (Golfers.ElementAt(i).Handicap >= hole.StrokeIndex)
    {
        if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex + 18)) &
(Course.Holes.Count == 18))
        {
            ModifiedPar[i] = hole.Par + 2;
        }
        else if ((Golfers.ElementAt(i).Handicap >= (hole.StrokeIndex +
9)) & (Course.Holes.Count == 9))
        {
            ModifiedPar[i] = hole.Par + 2;
        }
        else
        {
            ModifiedPar[i] = hole.Par + 1;
        }
    }
    else
    {
        ModifiedPar[i] = hole.Par;
    }
}
}
return ModifiedPar;
}
private int FindScore(int Score, int Par)
{
    int TotalScore = 0;
    if (Score - Par == 1)
    {
        TotalScore += 1;
    }
    else if (Score - Par == 0)
    {
        TotalScore += 2;
    }
    else if (Score - Par < 0)
    {
        TotalScore += ((Score- Par) - 2)* (-1);
    }
    else
    {
        TotalScore += 0;
    }
}

```

```

        return TotalScore;
    }

    [NakedObjectsIgnore]
    public void TotalScoreCalculated(Hole hole, int[] Scores, StablefordScores hs,
int[] handicaps)
    {
        int[] TotalScore = new int[4];
        for (int i = 0; i < 4; i++)
        {
            TotalScore[i] += FindScore(Scores[i], handicaps[i]);
            Scores[i] = TotalScore[i];
        }
        hs.GolferAActualScore = Scores[0];
        hs.GolferBActualScore = Scores[1];
        hs.GolferCActualScore = Scores[2];
        hs.GolferDActualScore = Scores[3];
        TotalScoreA += Scores[0];
        TotalScoreB += Scores[1];
        TotalScoreC += Scores[2];
        TotalScoreD += Scores[3];
        hs.GolferATotalScore = TotalScoreA;
        hs.GolferBTotalScore = TotalScoreB;
        hs.GolferCTotalScore = TotalScoreC;
        hs.GolferDTotalScore = TotalScoreD;
    }

    [NakedObjectsIgnore]
    public void FindWinner()
    {
        int[] TotalScores = { TotalScoreA, TotalScoreB, TotalScoreC, TotalScoreD };
        for (int i = 0; i < 4; i++)
        {
            if (TotalScores.Max() == TotalScores[i])
            {
                Winner = Golfers.ElementAt(i);
            }
        }
        MatchOver = true;
    }
}
}

StablefordAuthoriser
namespace GolfRecord.Model
{
    public class StablefordAuthoriser : ITypeAuthorizer<Stableford>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, Stableford match, string memberName)
        {
            if ((memberName == "Winner") | (memberName == "MatchType") | (memberName ==
"MatchCreator"))
            {
                return false;
            }
        }
    }
}

```

```

        else if ((match.Golfers.Contains(GolferServices.Me())) | (match.MatchCreator
== GolferServices.Me()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public bool IsVisible(IPrincipal principal, Stableford match, string memberName)
    {
        if ((match.Golfers.Contains(GolferServices.Me()) == false) & ((memberName ==
"AddScores") | (memberName == "DescriptionOfMatch") | (memberName ==
"AddPostMatchDescription"))))
        {
            return false;
        }
        else if ((match.Golfers.Contains(GolferServices.Me())) & (match.Winner !=
null) & (memberName == "DescriptionOfMatch"))
        {
            return true;
        }
        else if ((match.Winner == null) & ((memberName == "DescriptionOfMatch" )|
(memberName == "AddPostMatchDescription"))))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}

```

StablefordScores

```
namespace GolfRecord.Model
```

```

{
    public class StablefordScores : HoleScoreAbstract
    {
        [MemberOrder(2)]
        public virtual int GolferARawScore { get; set; }

        [MemberOrder(5)]
        public virtual int GolferBRawScore { get; set; }

        [MemberOrder(8)]
        public virtual int GolferCRawScore { get; set; }

        [MemberOrder(11)]
        public virtual int GolferDRawScore { get; set; }

        [MemberOrder(3)]
        public virtual int GolferAActualScore { get; set; }

        [MemberOrder(6)]

```



```

        public virtual int GolferBActualScore { get; set; }

        [MemberOrder(9)]
        public virtual int GolferCActualScore { get; set; }

        [MemberOrder(12)]
        public virtual int GolferDActualScore { get; set; }

        [MemberOrder(4)]
        public virtual int GolferATotalScore { get; set; }

        [MemberOrder(7)]
        public virtual int GolferBTotalScore { get; set; }

        [MemberOrder(10)]
        public virtual int GolferCTotalScore { get; set; }

        [MemberOrder(13)]
        public virtual int GolferDTotalScore { get; set; }
    }
}
Strokeplay
namespace GolfRecord.Model
{
    public class Strokeplay : Match
    {
        [NakedObjectsIgnore]
        public virtual int TotalScoreA { get; set; }

        [NakedObjectsIgnore]
        public virtual int TotalScoreB { get; set; }

        [NakedObjectsIgnore]
        public virtual int TotalScoreC { get; set; }

        [NakedObjectsIgnore]
        public virtual int TotalScoreD { get; set; }

        public void AddScores(Hole hole, int ScoreA, int ScoreB, int ScoreC, int ScoreD)
        {
            int[] TotalScores = { TotalScoreA + ScoreA, TotalScoreB + ScoreB,
TotalScoreC+ScoreC,TotalScoreD+ScoreD };
            var hs = Container.NewTransientInstance<StrokeplayScores>();
            hs.Hole = hole;
            hs.GolferARawScore = ScoreA;
            hs.GolferBRawScore = ScoreB;
            hs.GolferCRawScore = ScoreC;
            hs.GolferDRawScore = ScoreD;
            hs.GolferAActualScore = TotalScores[0];
            hs.GolferBActualScore = TotalScores[1];
            hs.GolferCActualScore = TotalScores[2];
            hs.GolferDActualScore = TotalScores[3];
            TotalScoreA = TotalScores[0];
            TotalScoreB = TotalScores[1];
            TotalScoreC = TotalScores[2];

```

```

        TotalScoreD = TotalScores[3];
        Container.Persist(ref hs);
        HoleScores.Add(hs);
        if (hole.HoleNumber == Course.Holes.Count)
        {
            for (int i = 0; i < 4; i++)
            {
                TotalScores[i] -= Golfers.ElementAt(i).Handicap;
            }
            FindWinnerStrokePlay();
        }
    }
    [NakedObjectsIgnore]
    public void FindWinnerStrokePlay()
    {
        List<int> Scores = new List<int>();
        int[] TotalScores = { TotalScoreA, TotalScoreB, TotalScoreC, TotalScoreD };
        for (int i = 0; i < 4; i++)
        {
            Scores.Add(TotalScores[i]);
        }
        for (int i = 0; i < 4; i++)
        {
            if (Scores.Min() == TotalScores[i])
            {
                Winner = Golfers.ElementAt(i);
            }
        }
        MatchOver = true;
    }

    public string ValidateAddScores(Hole hole, int A, int B, int C, int D)
    {
        if ((A <= 0) | (B <= 0) | (C <= 0) | (D <= 0))
        {
            return "A score can not be negative or 0";
        }
        else
        {
            return null;
        }
    }
}

StrokeplayAuthoriser
namespace GolfRecord.Model
{
    public class StrokePlayAuthoriser : ITypeAuthorizer<Strokeplay>
    {
        public GolferServices GolferServices { set; protected get; }

        public bool IsEditable(IPrincipal principal, Strokeplay match, string memberName)
        {
            if ((memberName == "Winner") | (memberName == "MatchType") | (memberName ==
"MatchCreator"))

```

```

        {
            return false;
        }
        else if ((match.Golfers.Contains(GolferServices.Me())) | (match.MatchCreator
== GolferServices.Me()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public bool IsVisible(IPrincipal principal, Strokeplay match, string memberName)
    {
        if ((match.Golfers.Contains(GolferServices.Me()) == false) & ((memberName ==
"AddScores") | (memberName == "DescriptionOfMatch") | (memberName ==
"AddPostMatchDescription")))
        {
            return false;
        }
        else if ((match.Golfers.Contains(GolferServices.Me())) & (match.Winner !=
null) & ((memberName == "DescriptionOfMatch")))
        {
            return true;
        }

        else if ((match.Winner == null) & ((memberName == "DescriptionOfMatch") |
(memberName == "AddPostMatchDescription")))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}

```

StrokeplayScores

namespace GolfRecord.Model

```

{
    public class StrokeplayScores:HoleScoreAbstract
    {
        [MemberOrder(2)]
        public virtual int GolferARawScore { get; set; }

        [MemberOrder(5)]
        public virtual int GolferBRawScore { get; set; }

        [MemberOrder(8)]
        public virtual int GolferCRawScore { get; set; }

        [MemberOrder(11)]
        public virtual int GolferDRawScore { get; set; }

        [MemberOrder(3)]
    }
}

```

```

        public virtual int GolferAActualScore { get; set; }

        [MemberOrder(6)]
        public virtual int GolferBActualScore { get; set; }

        [MemberOrder(9)]
        public virtual int GolferCActualScore { get; set; }

        [MemberOrder(12)]
        public virtual int GolferDActualScore { get; set; }

    }
}

UnregisteredGolfer
namespace GolfRecord.Model
{
    public class UnRegisteredGolfer:Golfer
    {
        public virtual Golfer GolferCreator { get; set; }
    }
}

```