Politecnico di Milano

Design and implementation of mobile applications

Swift Ios

Betbook

Authors:
Alessandro Cimbelli
Matteo Daverio

Professor:
Luciano Baresi

February 9, 2016

Contents

1 Requirements

The following lines describe the system requirements:

1. *Bet Coverage*: User can calculate the "coverage" of a bet that has already been made. The app will provide two modalities:
   - *Offline*: Where the user already knows what to perform to cover his bet, therefore needing only the "calculation" task to be done by the app.
   - *Online*: In this case, the app is going to automatically cover the previous bet made by the user.

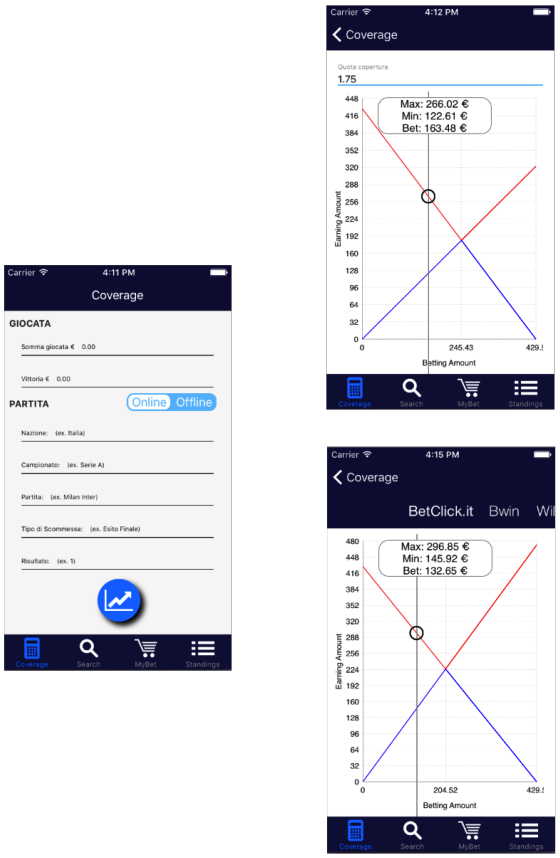


Figure 1: File Coverage UI

2. *Search*: The user can see the scheduled matches for a specific league and, if available, he can also see the odds for a specific match. The user will therefore have the possibility to add, remove, and modify matches from his personal bet.

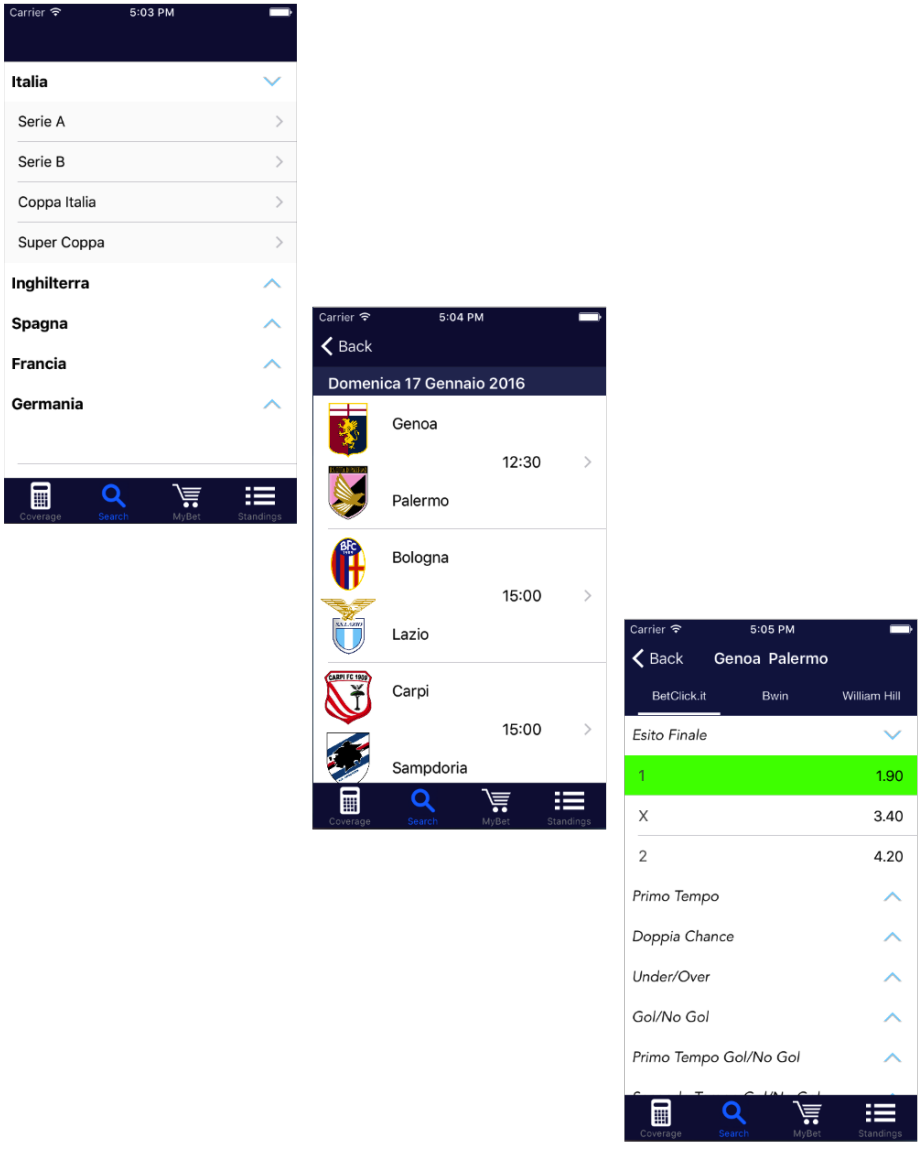


Figure 2: File Search UI

3. *Bet Management*: The user can modify some parameters of his personal bet such as the bet amount and the bonus coefficient. He can also delete matches or add them.

Figure 3: File Bet Management UI

4. *Standings View*: The user can see the rankings of the top leagues along with the last results of each team.

Figure 4: File Standings View UI

2 Design Pattern

The design pattern used to develop the application is the popular MVC that stands for *Model View Controller*. The main concept of this pattern is that you must divide the objects of your program into 3 camps:

- *Model*: That represents what the application is, and its data (but not how is displayed)
- *Controller*: That is in charge to specify how the model is presented to the user (UI Logic)
- *View*: That is the Controller's minions and represents to UI

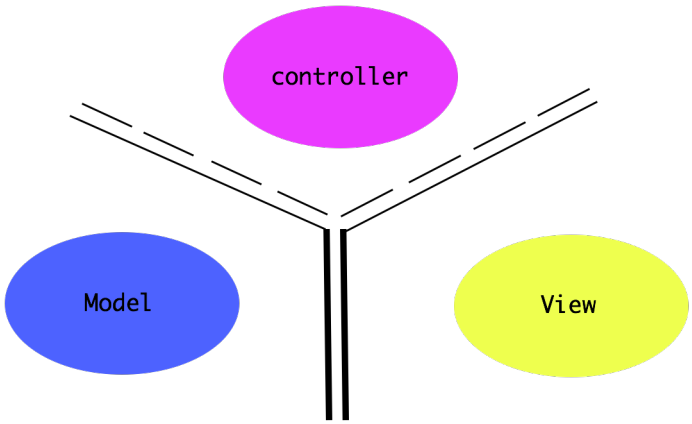


Figure 5: File MVC

This pattern provides an incredible scalability that could allow us to include new features in the future without having integration problems. Therfore it simplifies the task of debugging the application. We can in first approximation imagine that every screen of our app is ruled by an *ad hoc* controller. In this way we are going to introduce the possibility to reuse some part of this app in the future. This pattern also helped us in the subdivision of the amount of work for each component of the team. In some cases, we have root controllers that control other, more simple controllers. This separation of the objects of our app is going to be underlined by the next classes diagrams.

3 Class Diagrams

3.1 Coverage

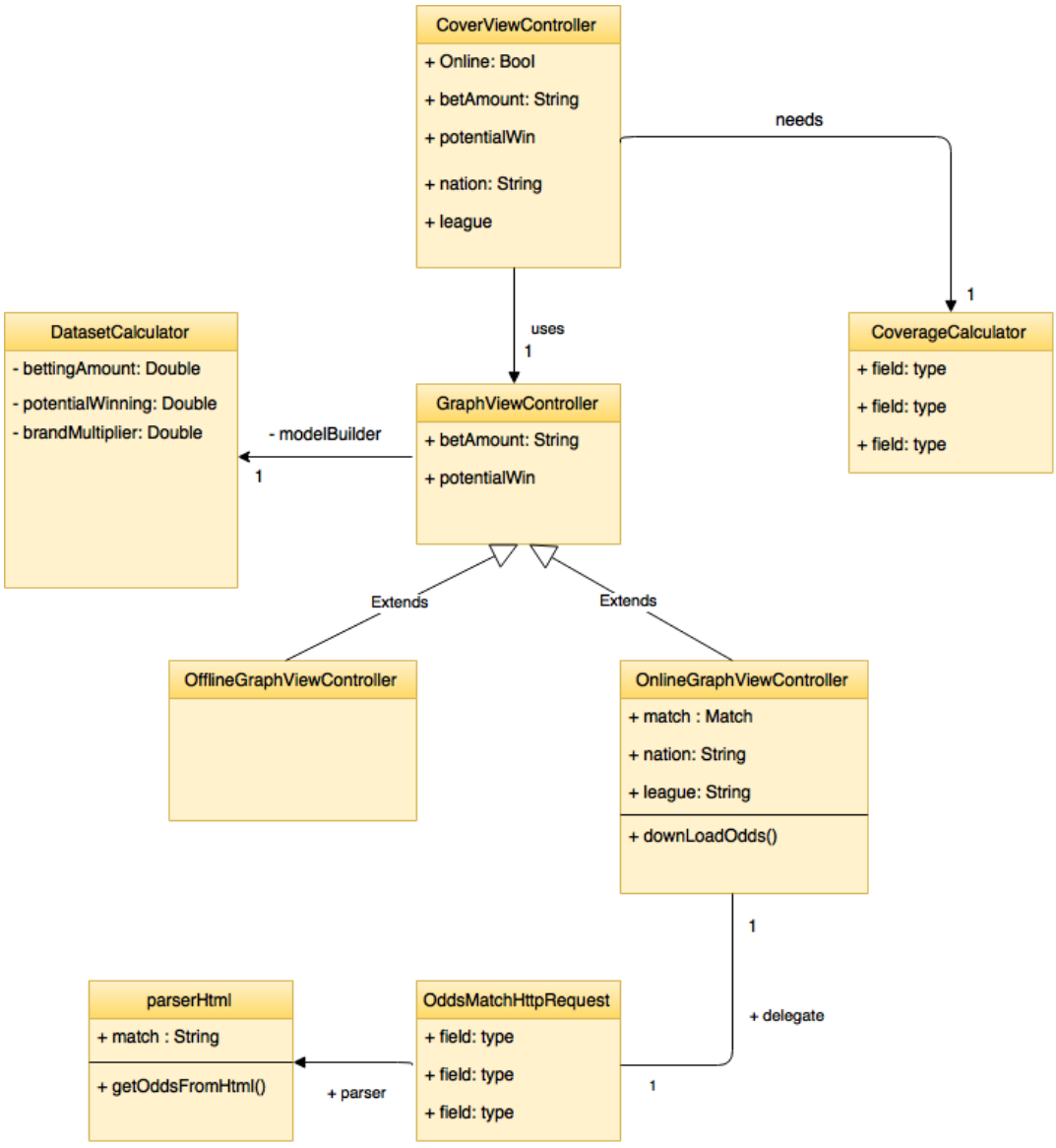


Figure 6: File Coverage Class Diagram

3.2 MatchList

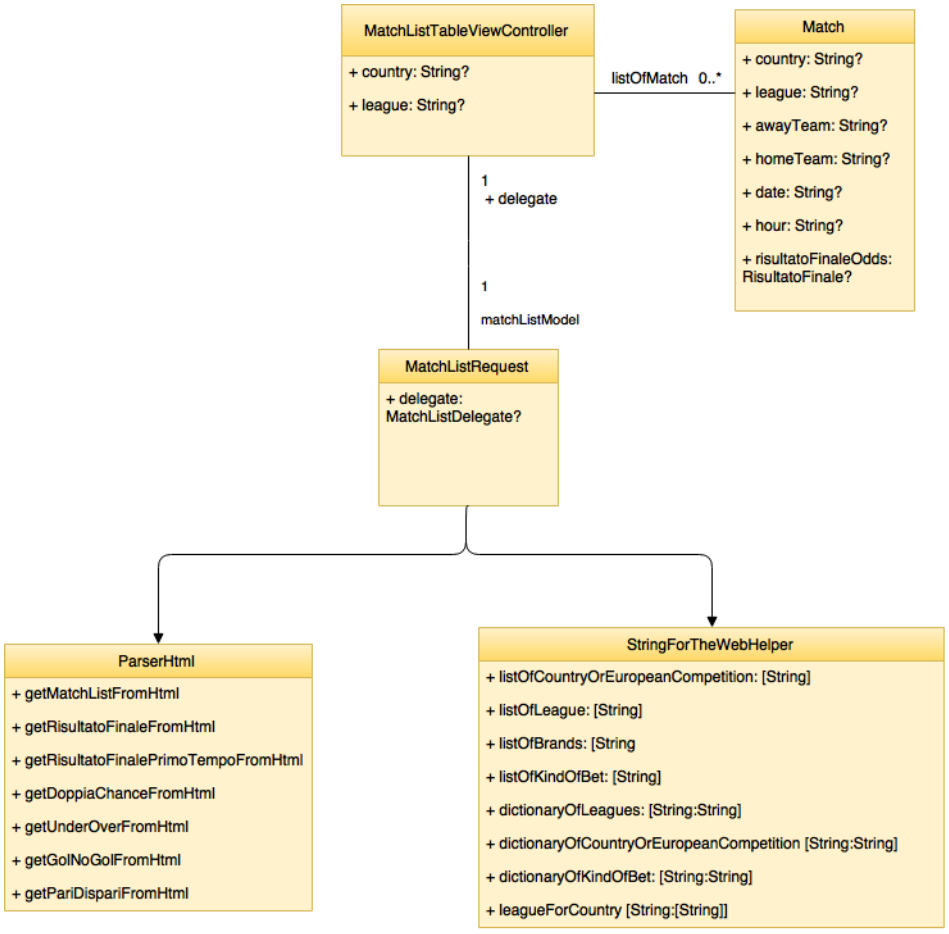


Figure 7: File MatchList Class Diagram

3.3 LeaderBoard

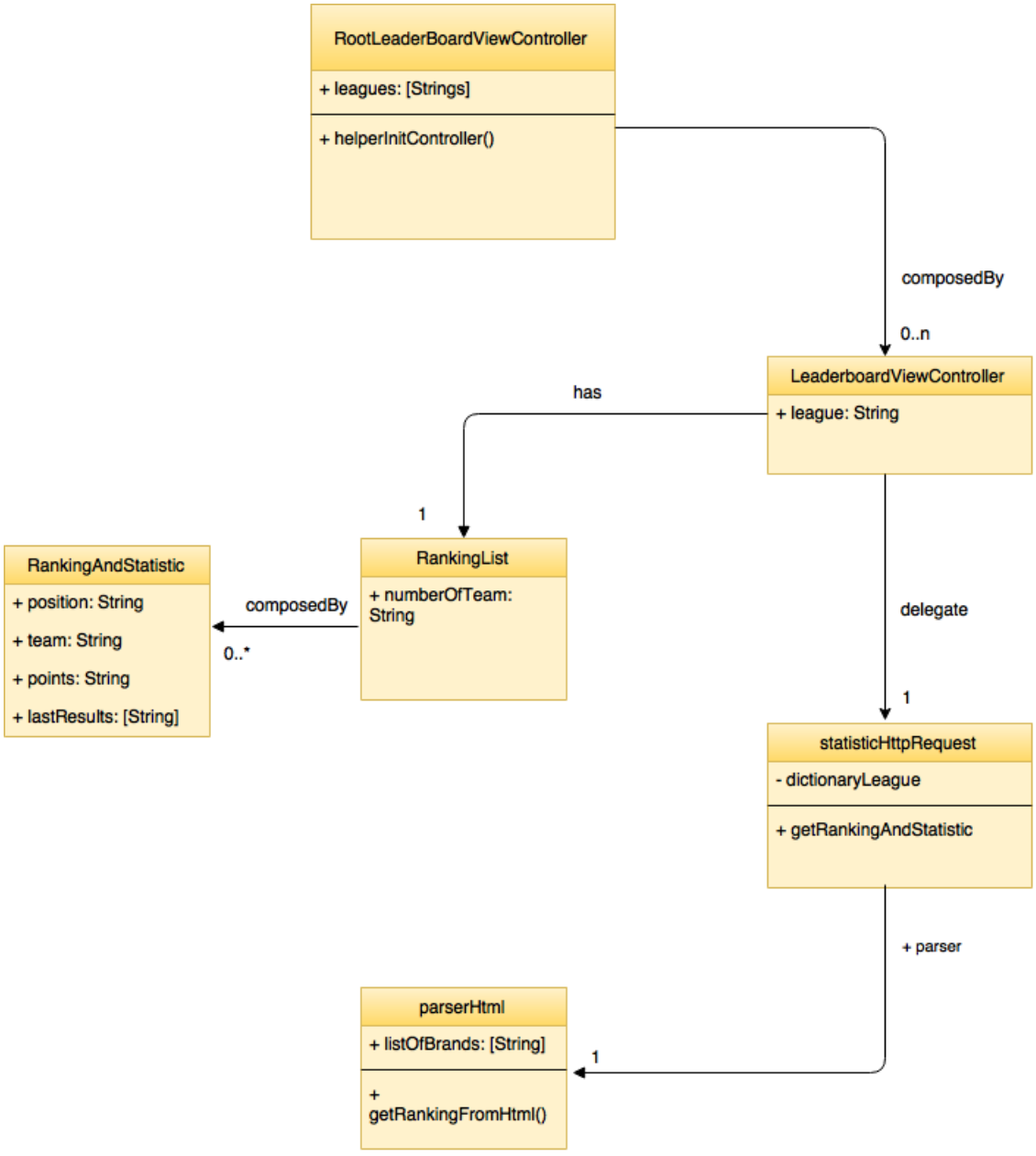


Figure 8: File LeaderBoard Class Diagram

4 Sub Systems

We separate our systems into four sub-systems:

- BetCoverage Sub-System
  - Online Coverage Sub-System
  - Offline Coverage Sub-System
- Evets Viewer Sub-System
  - Betting quotes viewer Sub-System
  - Betting pool management Sub-System
- Betting pool viewer Sub-System
- Leaderboards viewer Sub-System

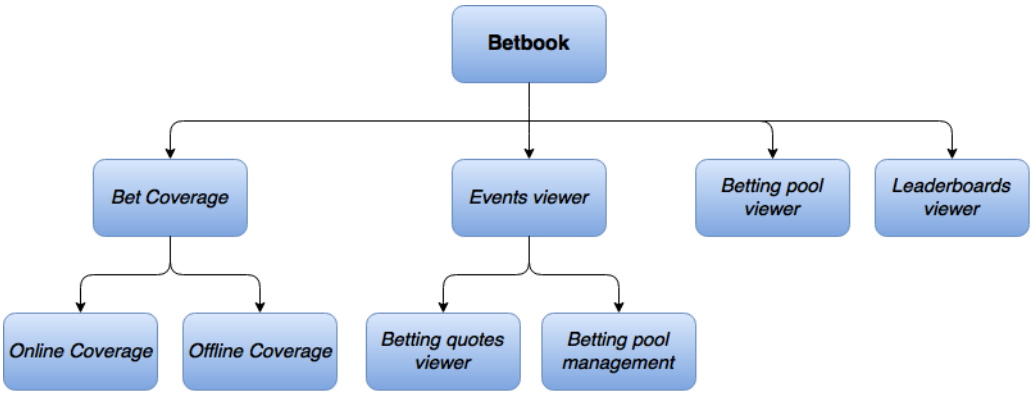


Figure 9: File BetBook's SubSystems

5 Connectivity

To exploit all the features of the app is required to be connected to the internet. In absence of connection only some features can be used. According to what we said above we can classify our app as *Partially Connected*. With no connection only the following features will be available:

- Offline Coverage of a bet: with its graph and details
- Visualization of the personal bet previously composed
- Visualization of rankings and statistics already loaded

While in presence of connection will be also possible to completely use the app, taking advantage of the data retrived by the *External Services*:

- Online Coverage of a bet: with the online available odds
- Visualization of scheduled matches
- Visualization of available odds for a match
- Addition of matches to the personal bet
- Refresh and download of rankings and statistics

In case of absence of connection or lost of connection durning the use of the app, an alert message will be shown to the user.

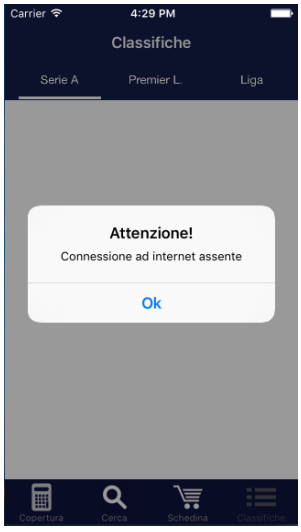


Figure 10: File No Connection Alert

6 Persistence

In our app, we want to make our betting pool persistent. In fact, we want to give the possibility to the user to create it in a moment, and re-open and/or modify it later. To do this we need to store all the data in a persistent way, to prevent a loss of data due to app's termination.

6.1 Data type

In order to do complete this task, we need to first define the data type we want to store. The most important things we need are:

- HomeTeam: name of the host team
- AwayTeam: name of the guest team
- Date: date of the match
- Hour: hour of the match
- League: league or cup in which the match takes place
- Country: country of the league or cup
- KindOfBet: kind of bet choosen (Final result, Gol/NoGol, ...)
- Bet: bet choosen in relation of the kind of bet
- BetValue: bet outcome given by the brand
- Brand: betting entity on which it is decided to bet

All these items are saved as a string, because it is the way we decide to obtain all the data from our parser.

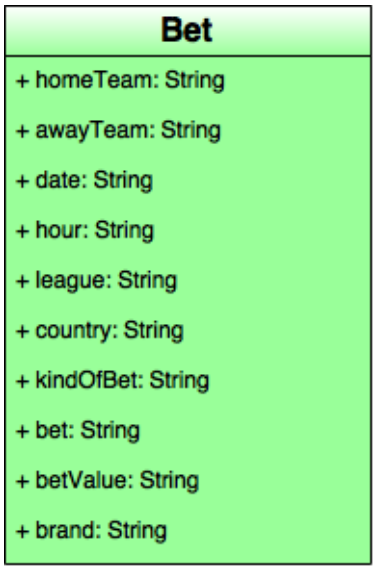


Figure 11: File Persistence

6.2 Implementation choice

In this section we try to discuss about some possible way to save our data. We know that, for each instance, we need very little space, because we save only few strings, with very low space consumption. We first think about the possibility in using databases like MySQL, but we operate only on very small amount of data, because not only our instances are very small, but a soccer betting pool is very often populated with only a small amount of matches.

It is easy to see that it is preferred the usage of a mechanism that is much faster and can easily operate in very small amount of data. To do so, we use NSUserDefaults, because can be used very easy, and perform very well on our data. We save in our persistence all the modifications done by the users, either if he adds or deletes some matches.

7 Use Case

7.1 Coverage

The system will provide the user with the possibility to perform online or offline coverage. The user will need to go to the coverage page, insert all the data, and choose which coverage he wants to perform. If the data inserted is correct, the user will see the chart, otherwise the system requires the user to insert correct data.

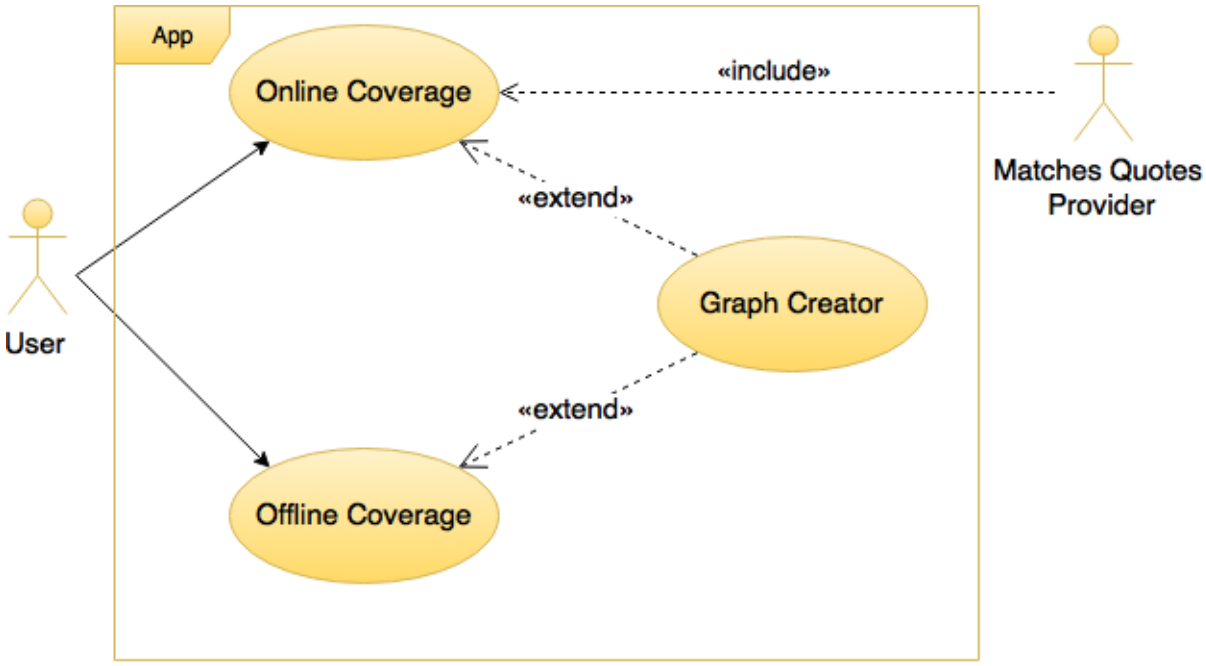


Figure 12: File Coverage

7.2 QuoteViewer

The system will provide the user with the possibility to see the entire event with their quotes for each brand, or to manage a soccer pool where all the events selected by the user are displayed. The user can go to event viewer page searching for a specific match, and looking for a specific betting quotation with all the different brands. Otherwise he can search for his soccer pool where he can see all the quotation he added and how much money he can win.

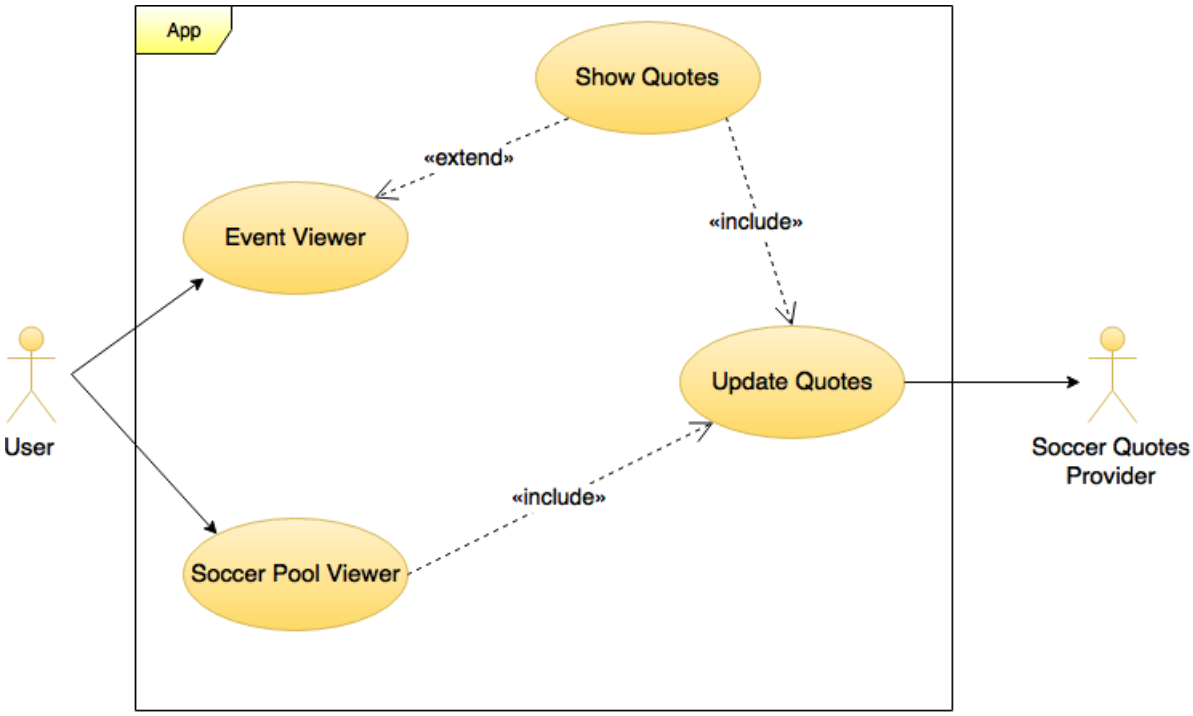


Figure 13: File QuoteViewer

8 Multithreading

The app requires to be designed to support the multithreading. This is because the presence of a user interface and the interaction with external services make it impossible to follow a different track. In the following pages, we are going to show an instance of this problem and how we solved it to guarantee an interactive user interface.

8.1 The MatchList Multithreading Point of View

In this scenario, once the user has selected the league of which he wants to see the scheduled matches, the app has to perform an external request to get the data without blocking the user interface.

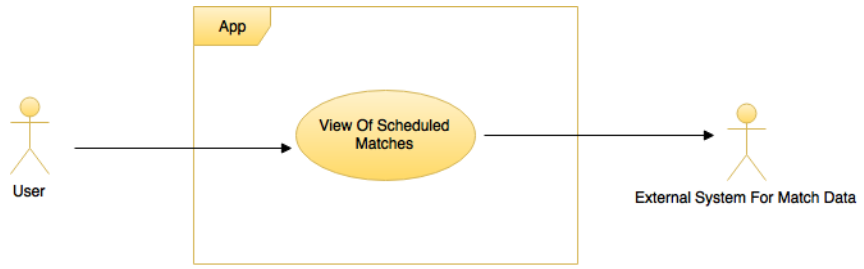


Figure 14: File The MatchList Multithreading Point of View

We designed the app to dispatch in an asynchronous way the *slow request* to the external service. In this way, the user interface doesn't result in it being frozen and when the data are available, the app logic is in charge to update the user interface. The controller responsible to display the list of scheduled matches will delegate to a specific class the task of retrieving the data from the web. Then, through a specific protocol, the *http request class*, when available, will give back the result of the external request to the calling class. In the following images, we are going to show an example of what explained above.

We can see that who wants to use the MatchRequestClass has to implement the protocol MatchListDelegate. This is needed because when the data is back, the class must have the possibility to comunicate the result of the previous request.

```
protocol MatchListDelegate {
    //Gives back to the delegate the list of match scheduled
    func setMatchList(matchList: [Match]?)
}

class MatchListRequest {

    //Property
    var delegate: MatchListDelegate?
```

Figure 15: File protocol

Here we can see an example of the protocol implementation. It should be noticed that all the operation that are going to update the UI must be dispatched on the main queue. In fact it's never OK to modify the Ui from the background thread.

```
func setMatchList(matchList: [Match]?){

    //Questa istruzione serve per mettere questa istruzione in esecuzione sulla coda principale! Cosi la grafica viene aggiornata in maniera sicura senza problemi
    dispatch_async(dispatch_get_main_queue()){ () -> Void in
        if( matchList == nil){

            self.view.userInteractionEnabled = false

            let title = "Match non disponibili"
            let message = "Non sono al momento previsti incontri per questa competizione"
            let okText = "Ok"

            let alert = UIAlertController(title: title, message: message, preferredStyle: UIAlertControllerStyle.Alert)

            let okayButton = UIAlertAction(title: okText, style: UIAlertActionStyle.Cancel, handler: { (UIAlertAction) -> Void in
                self.spinner.stopAnimating()
            })

            alert.addAction(okayButton)
            self.presentViewController(alert, animated: true, completion: nil)
        }else{
            let organizedMatches = self.organizeForDate(matchList!)

            if(organizedMatches.count == 0){
                self.setMatchList(nil)
                return
            }

            self.matches = organizedMatches
            self.spinner.stopAnimating()
            self.refreshControl?.endRefreshing()
            self.tableView.reloadData()
        }
    }

}
```

Figure 16: File protocolImplementation

It is also very important that at the beginning, the class that wants to use the http requester class sets itself as the *delegate*.

```
class MatchListTableViewController: UITableViewController,MatchListDelegate {

    let spinner = UIActivityIndicatorView(frame: CGRectMake(0,0,100,100))

    //My model
    let matchListModel = MatchListRequest()

    var country: String?
    var league: String?

    var matches = [[Match]]()

    override func viewDidLoad() {
        super.viewDidLoad()

        //Importantissimo
        matchListModel.delegate = self
```

Figure 17: File MatchList Delegate

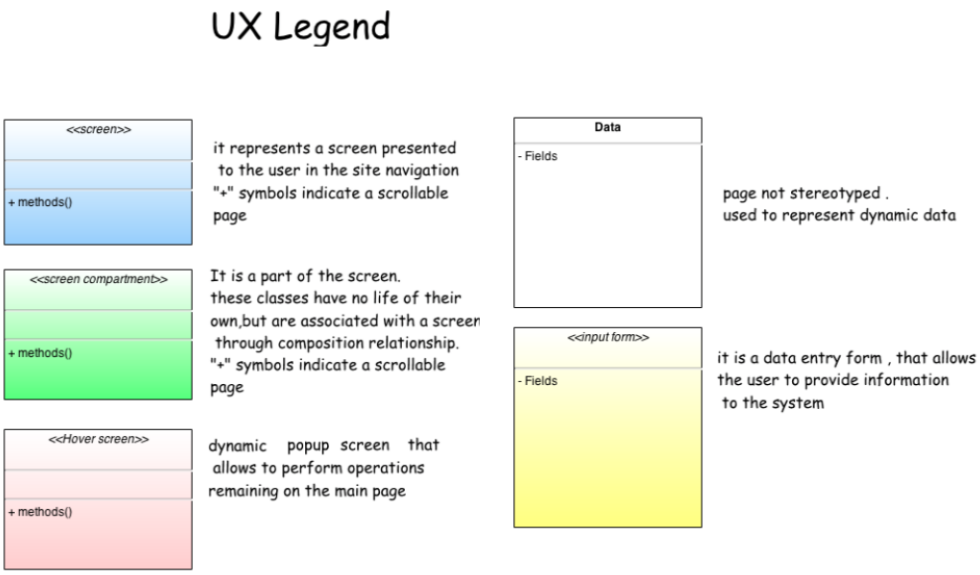
In the last picture we can see how the request arriving from tha matchlist controller has dispatched in an asynchronous way.

```
//This function ask to the model to look for the odds available on the web according to the parameters setted before like country and league
func getMatchList(uiCountryOrEuropeanCompetition: String, uiLeague: String ){

    let selectedCountryOrEuropeanCompetitionForTheWeb = stringHelper.selectedCountryOrEuropeanCompetition(uiCountryOrEuropeanCompetition)!

    let selectedLeagueForTheWeb = stringHelper.selectedLeague(uiLeague)!

    //At this point we should arleady have setted all the parameters to build the path, in the oppisite case we want to crash
    let path = basePath + slash + selectedCountryOrEuropeanCompetitionForTheWeb + slash + selectedLeagueForTheWeb

    //We create an url for the http request
    let url = NSURL(string: path)

    var matchList: [Match]?

    let session = NSURLSession.sharedSession()

    //Imposta il task
    //Le operazioni svolte dal task vanno su un altro thread!!! Quando faccio resume (lo avvio) e il task è eseguito a parte. Il task gira in maniera asincrona.
    let task = session.dataTaskWithURL(url!) { (data: NSData?, respons: NSURLResponse?, error: NSError?) -> Void in

        if(data != nil){

        if let doc = Kanna.HTML(html: data!, encoding: NSUTF8StringEncoding) {
            matchList = self.parserHtml.getMatchListFromHtml(doc,country: uiCountryOrEuropeanCompetition, league: uiLeague)
            }
        }
        if self.delegate != nil{

            let list = matchList

            self.delegate?.setMatchList(list)

        }
    }

    //serve per avviare il task
    task.resume()
}
}
```

Figure 18: File Dispatch of getMatchList

9 UX Diagrams

In this section, we want to describe the User Experience (UX) given by our system to its users. Every diagram will be presented with a brief description of the feature that is described and the main components that are involved in the graph. We added a legend for the kind of components that can be found in the graphs to help their interpretation.



9.1 Bet Coverage

This UX diagram represents how the user interacts with the system to look for betting coverage. He will insert all the details about its bet, either online or offline, knowing via chart the optimal betting amount, with maximum and minimum win.

Figure 19: File Bet Coverage

9.2 MatchList and Odds Menus

This UX diagram represents how the user interacts with the system to look for matches and the see the odds in detail offered by the bookmakers. During the check of the odds, he can also add to his personal bet the match that he is visioning.

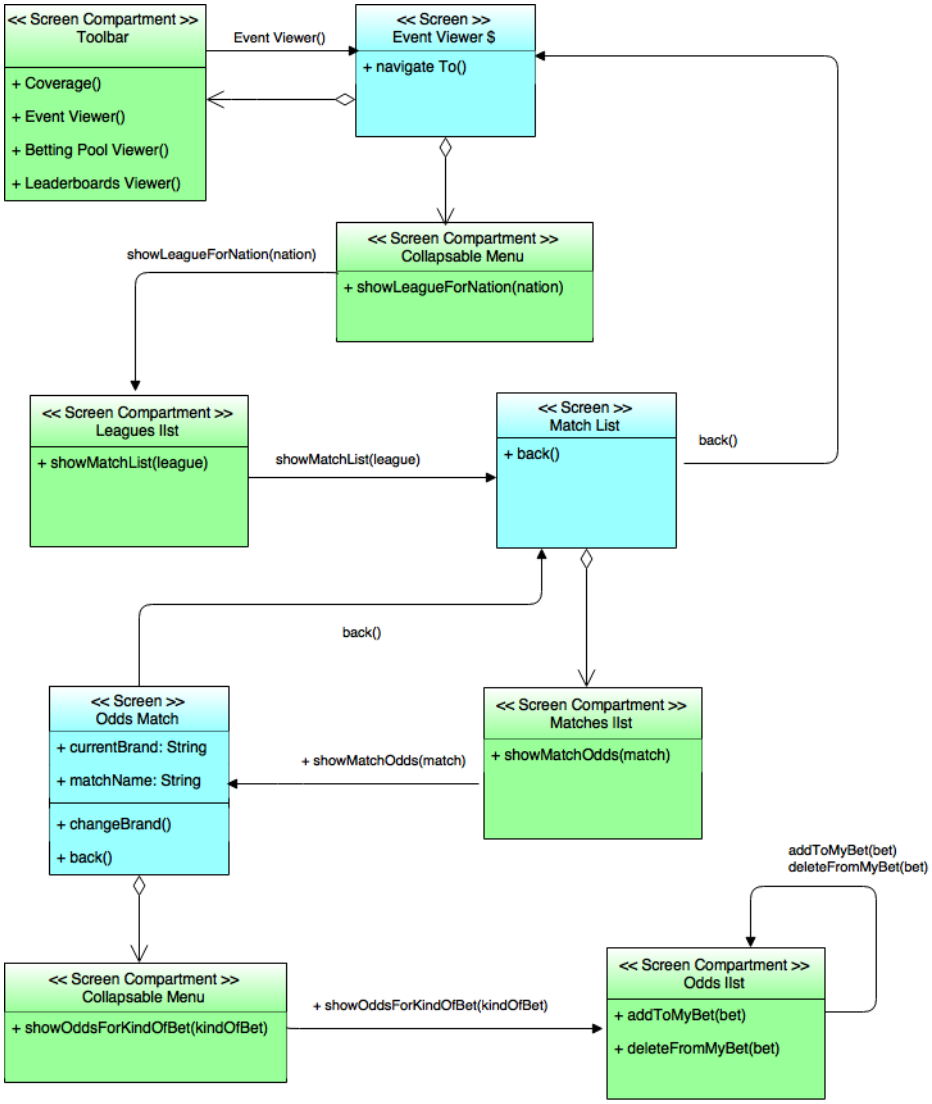


Figure 20: File MatchList and Odds Menus

9.3 Create my bet

This Ux represents how the user interacts with the system to create a bet. He can add new matches and modify the bet amount and the bonus percentage linked to the bet. The bet engine will display the computation connected to the potential win and the multiplier coefficient.

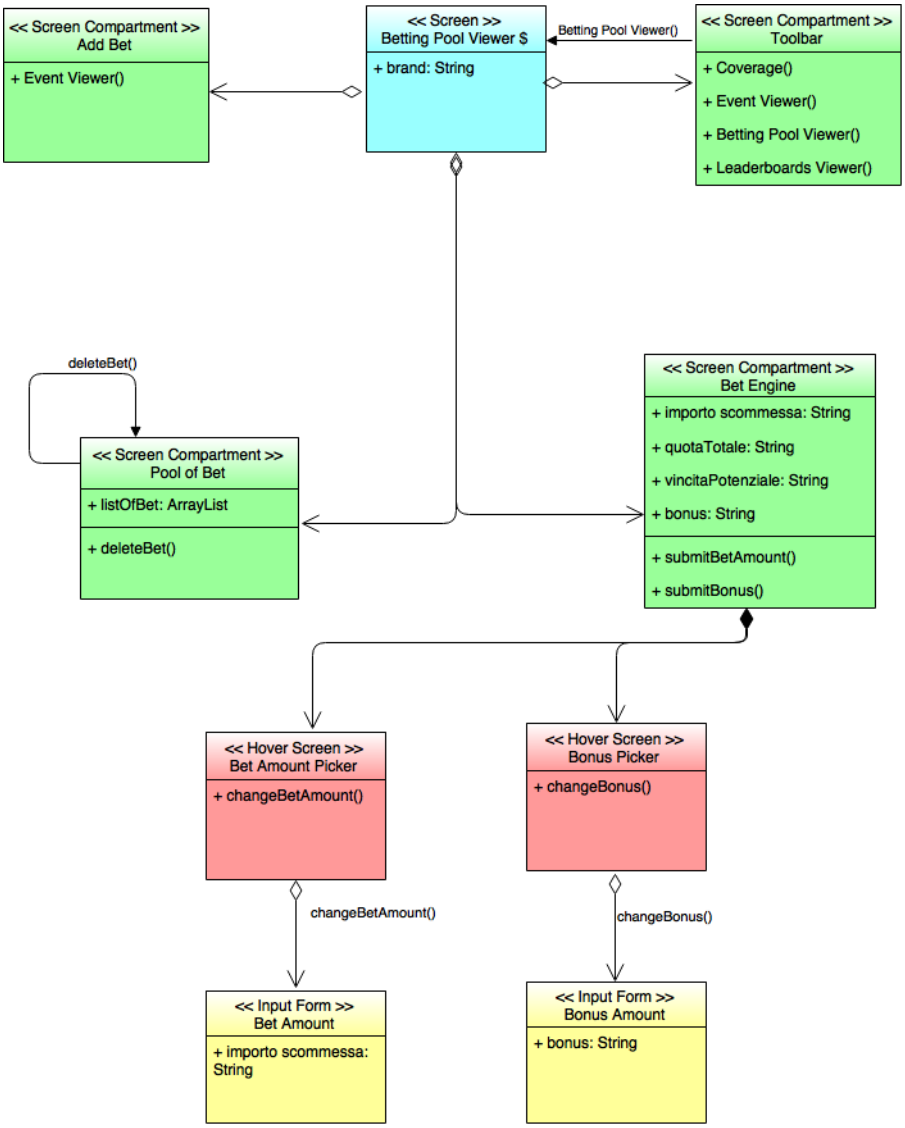


Figure 21: File Create my bet

9.4 View Leaderboard

This Ux represents how the user interacts with the system to look for leaderbords of the most important soccer championships, also looking for details abount score and previous performances of each team.

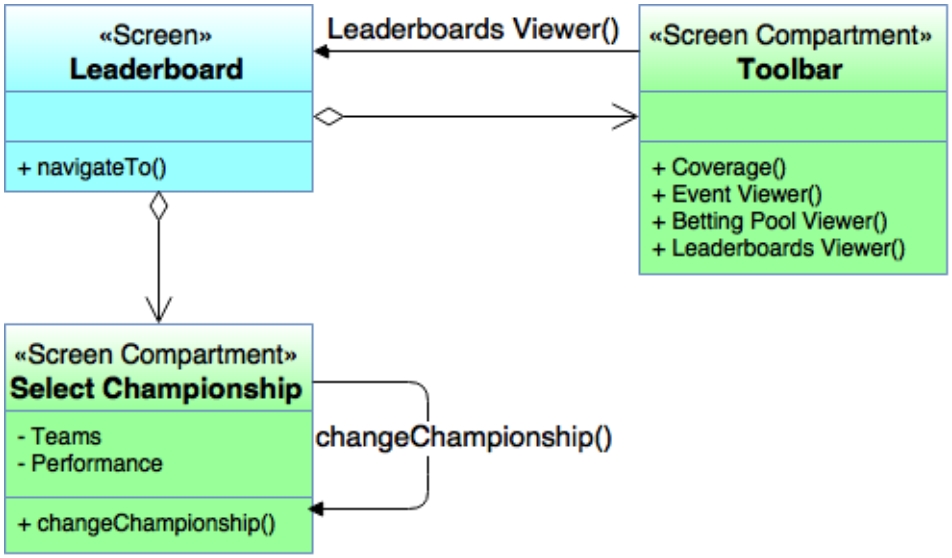
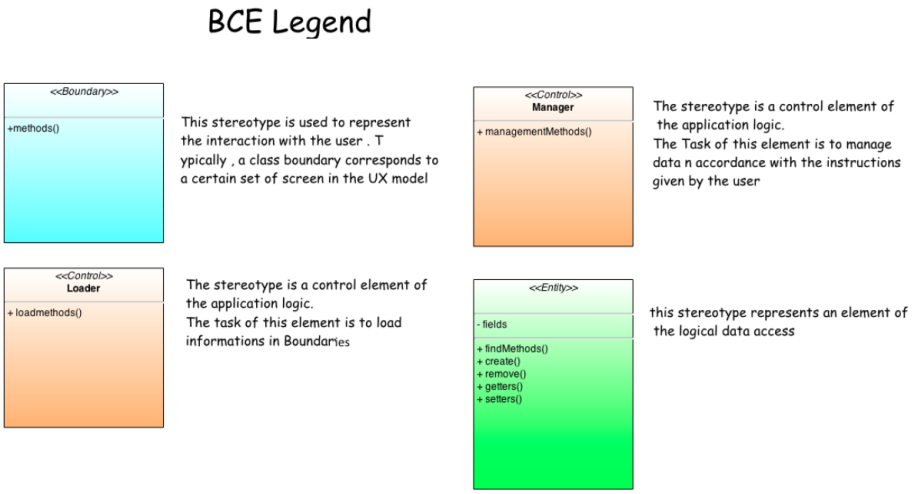


Figure 22: File View Leaderboard

10 BCE Diagrams

We decided to give a further scheme of BetBook using the Boundary-Control-Entity pattern, because it is very close to the Model-View-Controller pattern and the UMl defines a standard way to represent it. All methods of the screens are written into the appropriate boundaries (maybe some names changed a bit only to make them more understandable in this context). At the bottom of the page we inserted a legend trying to make the diagrams easier to understand.



10.1 Bet Coverage

The BCE diagram is characterized by three boundary "Coverage", "Online Graph" and "Offline Graph" which represent the interface with the users. The objective of the boundaries is to represent the functionalities that its user can exploit in Coverage page.

In the following lines will be described a concise description of the controllers involved into the diagram:

- GraphValueManager: this controller manages the verification of the data submitted by users. In case of valid input values, it send them to "Online Graph" or "Offline Graph", displaying that page.
- GraphLoader: this controller is responsible of the general management of the graph pages. It will create a graph using the data received by the "Coverage" page.

(see next page)

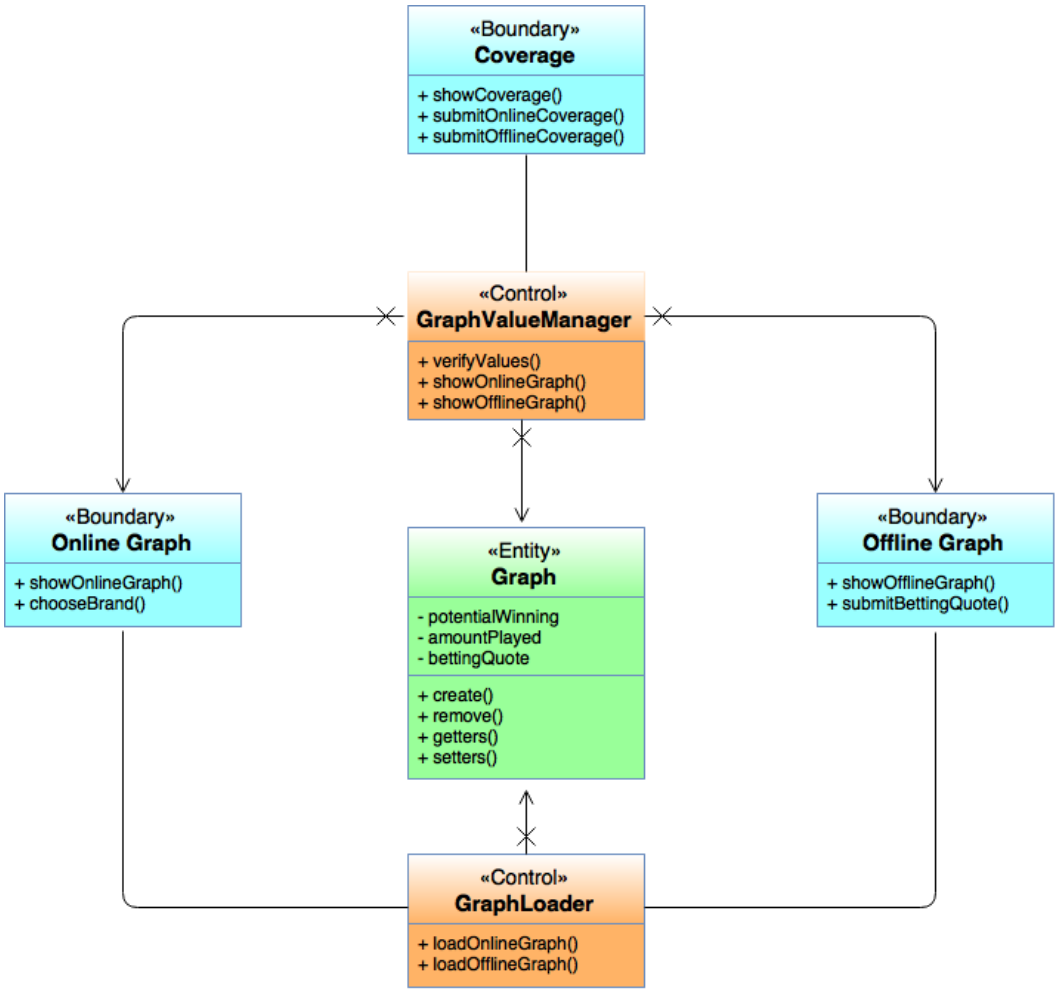


Figure 23: File Bet Coverage

10.2 View Match's Odds

The BCE diagram is characterized by three boundaries "Nation Menu", "League Matches" and "Odds Match" that represent the interface that the system provides to user to view the scheduled matches for a league and the odds for a specific match.

In the followings lines we're going to briefly describe the meaning of the controllers that are involved into the diagram:

- MenuLoader: this controller is responsible of the general management of the Nation Menu screen. It will load all the data associated to the menu and the leagues for each nation.
- Match List Loader: this controller is responsible to provide the list of the scheduled matches for a selected league.
- Odds Loader: when a user selects a match, this controller load the relative odds.

(see next page)

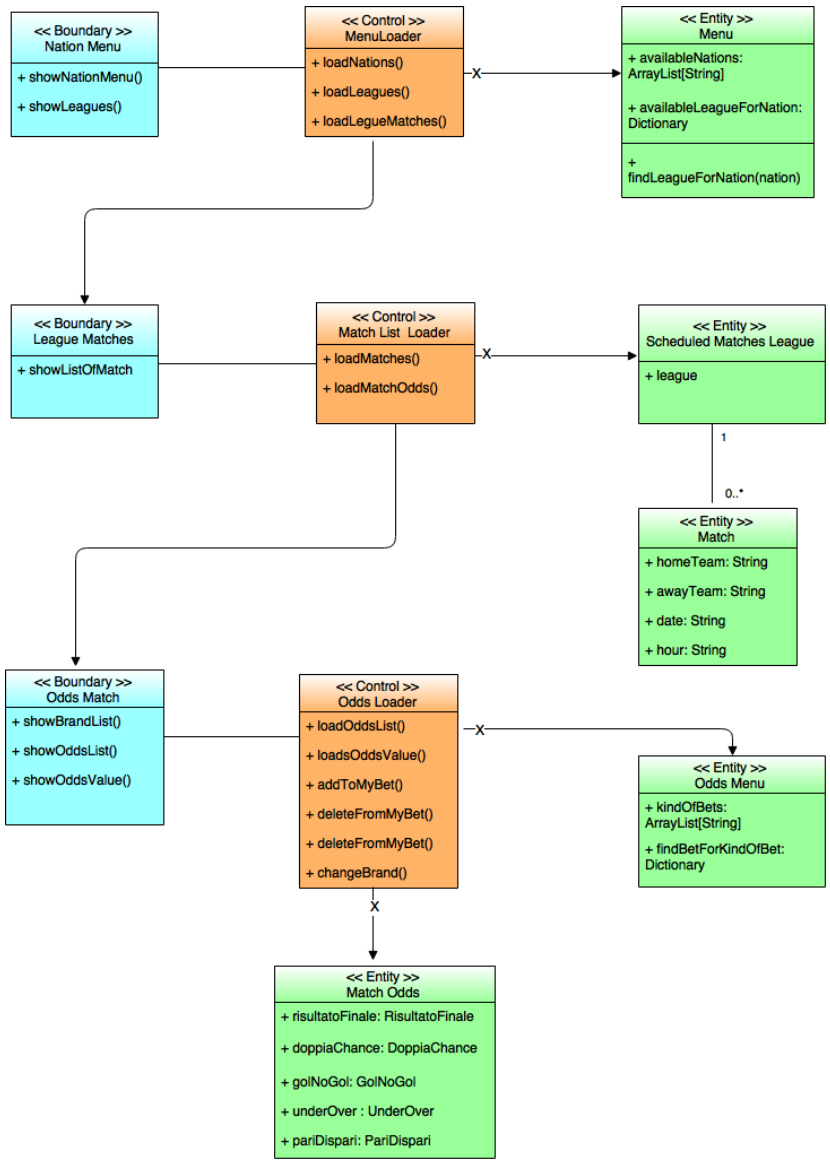


Figure 24: File View Match's Odds

10.3 View My Bet

This bce is caracterize by two boundaries that represent:

- MyBet: The app interface that shows the list of bet that the user added to his personal bet.
- MyBet Summary: The app interface that shows the parameters of the current bet like the betting amount or the totalmultiplier coefficient.

The two boundaries are managed by the respective controller.

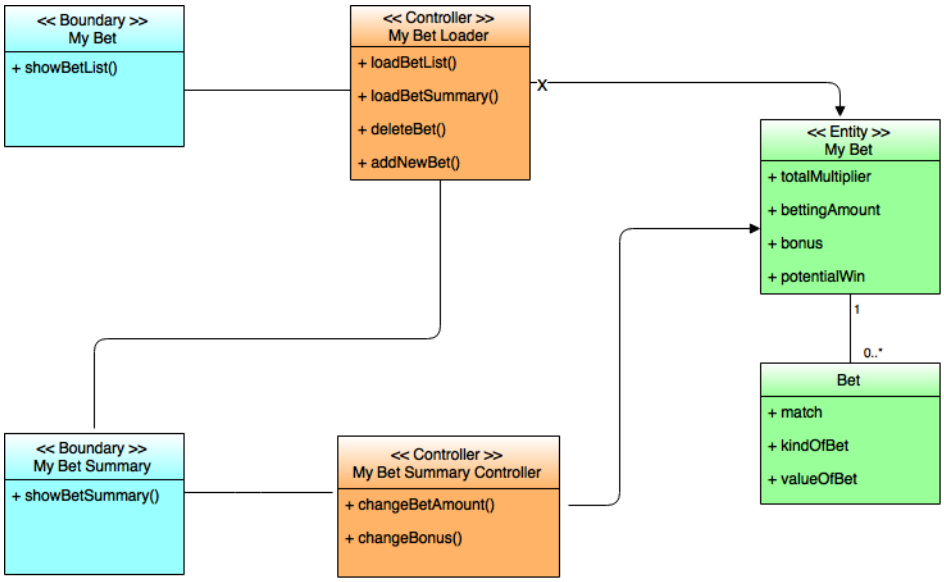


Figure 25: File View My Bet

10.4 View of Leaderboards

This BCE diagram is characterized by a single boundary "Leaderboard", which represent the interface with users. Its objective is to show the leaderboards of the main championships.

There is also a single controller "LeaderboardManager": this controller manages the selection of the championship, creating its leaderboard, and displays it.

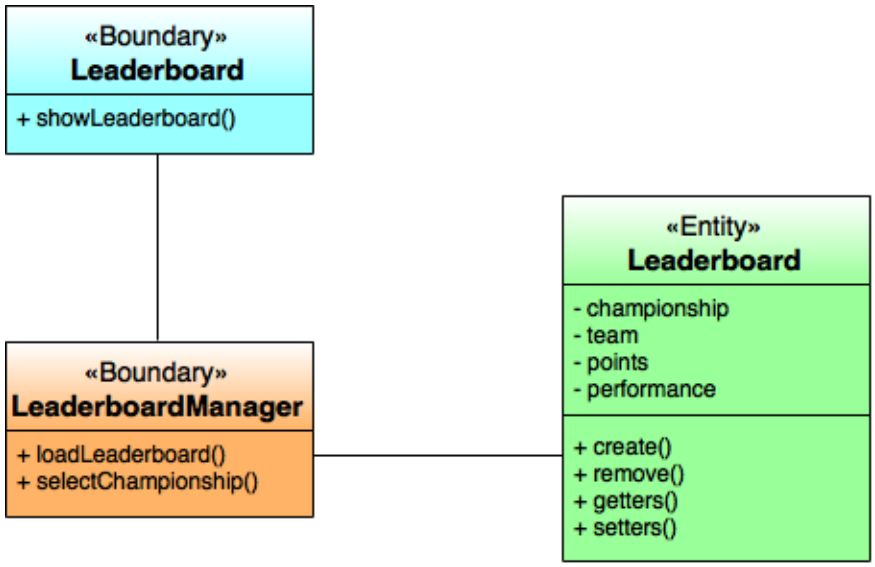


Figure 26: File View of Leaderboards

11 Sequence Diagram

We provided some sequence diagram to let the reader better understand BCE diagrams described above.

11.1 Match List Navigation

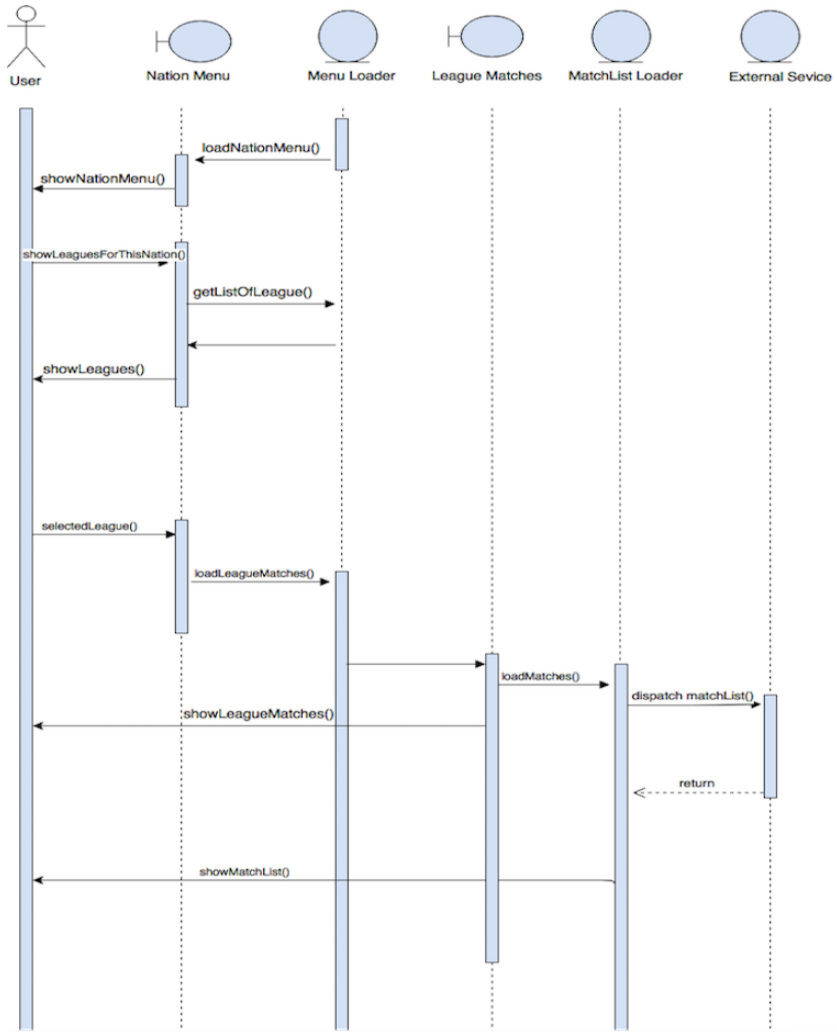


Figure 27: File Match List Navigation

11.2 Bet Coverage

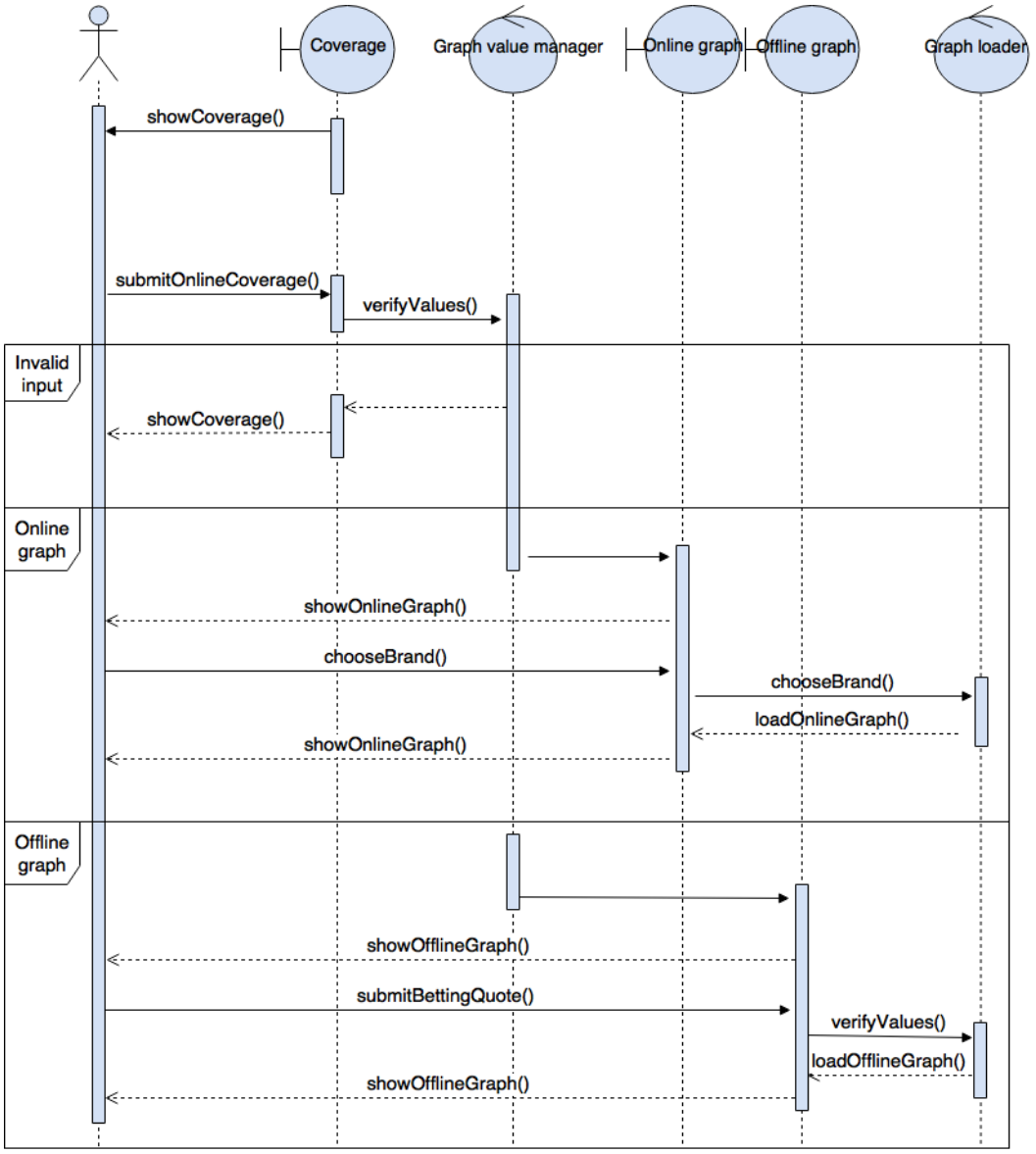


Figure 28: File Bet Coverage

11.3 View Match's Odds

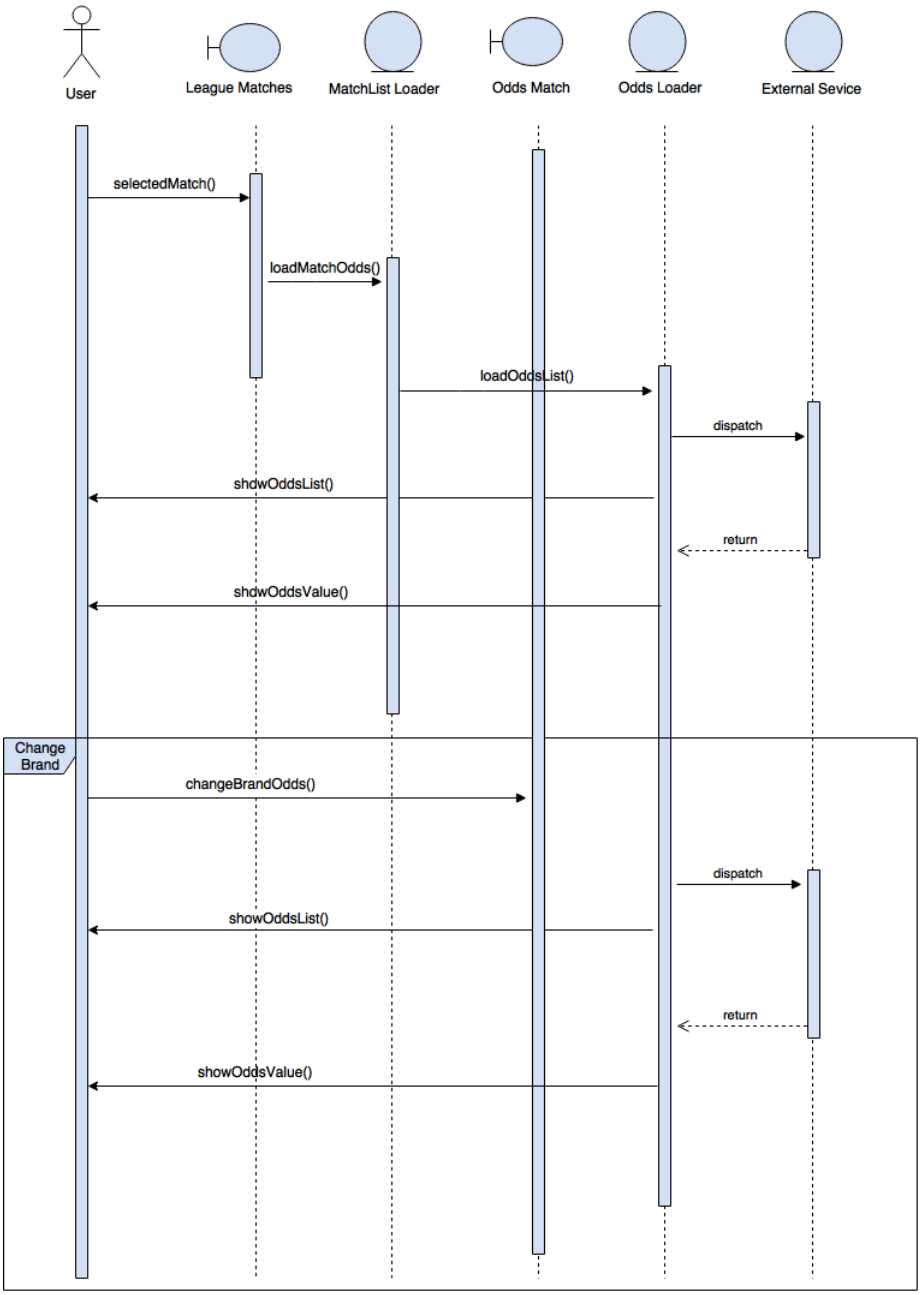


Figure 29: File View Match's Odds

11.4 My Bet Management

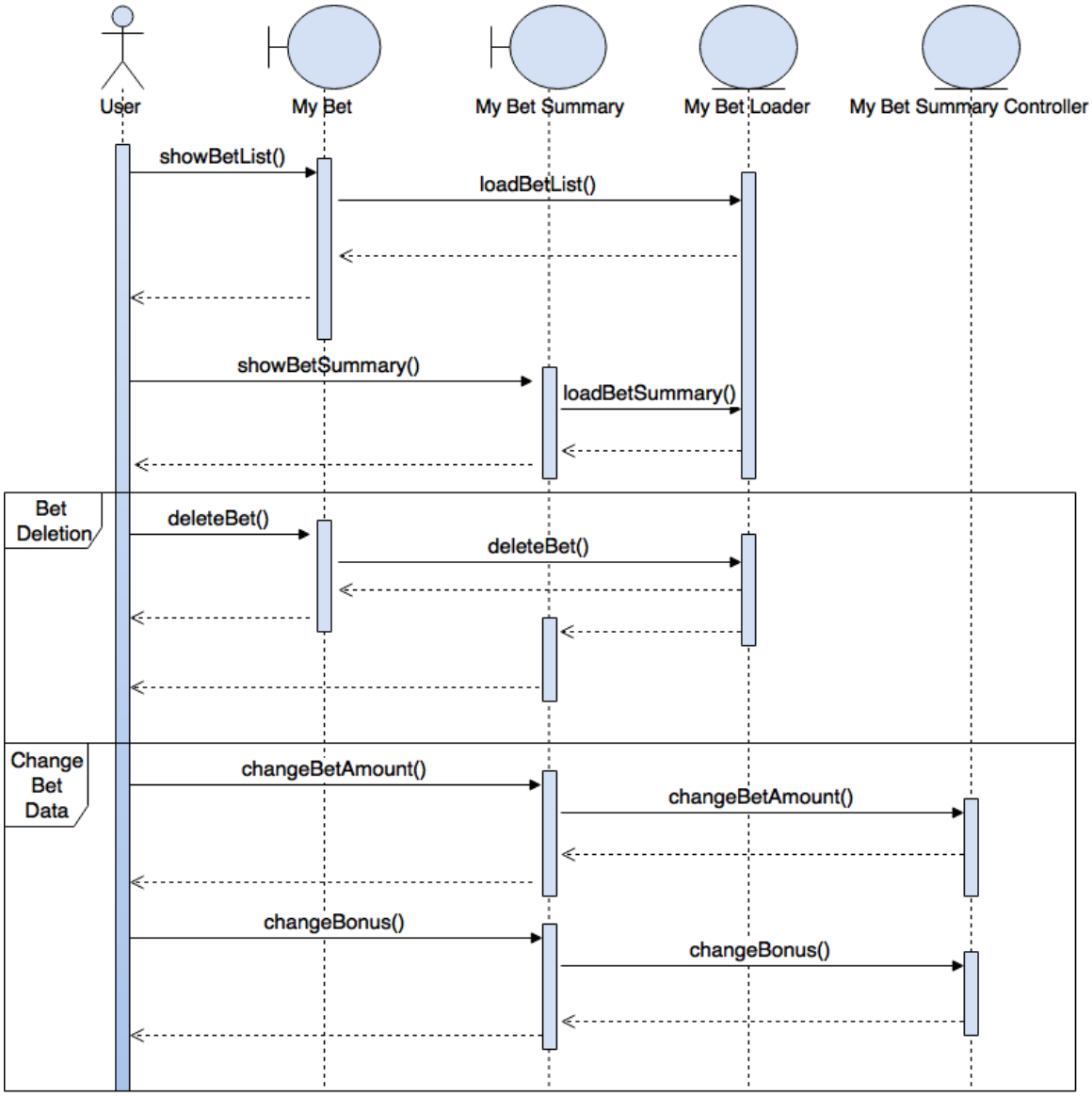


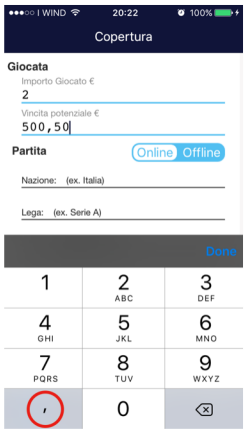
Figure 30: File My Bet Management

12 Testing

In this section, we try to describe some of the most important and relevant test we do on our app.

12.1 Test on device

We tested our app on real devices, to see how it works. As a result, we discovered an important issue on the decimal pad. The problem was that in relation to the country of the smartphone's owner, it gives the possibility to insert a comma or a dot when user insert a decimal number. We fixed that to handle either comma and dot, in the same way.



12.2 Test on connectivity

To see how our app works offline we tested it. To do so, we tried to start it without internet connection and tried to disable it while we were using the app. We fixed all the issues; our app displays a pop-up alerting the user that there is no internet connection, and the app can be used only offline, using the offline coverage, or looking to the betting pool already created.

12.3 Test on simulator

On the simulator, we tried to test our app on every possible iPhone layout, to makes it running with a correct layout, without cutting off parts of the view and without leaving some blank spaces. As a result, we understood that we needed in some scenes a scrollView, because on older devices, the screen has lower height, so we made them scrollable to correctly display everything.

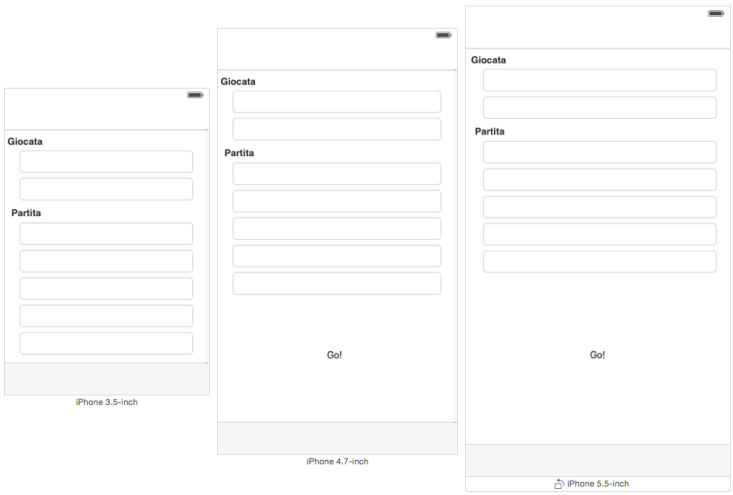


Figure 31: File Layout

For the specific case of a small screen like the iphone 4s, this is the screenshot of the bottom of the scrollview provided by the simulator

Figure 32: File ScrollView

12.4 Test on logic

We tested the app's logic with a white-box approach, checking the flow of our input into our logic.

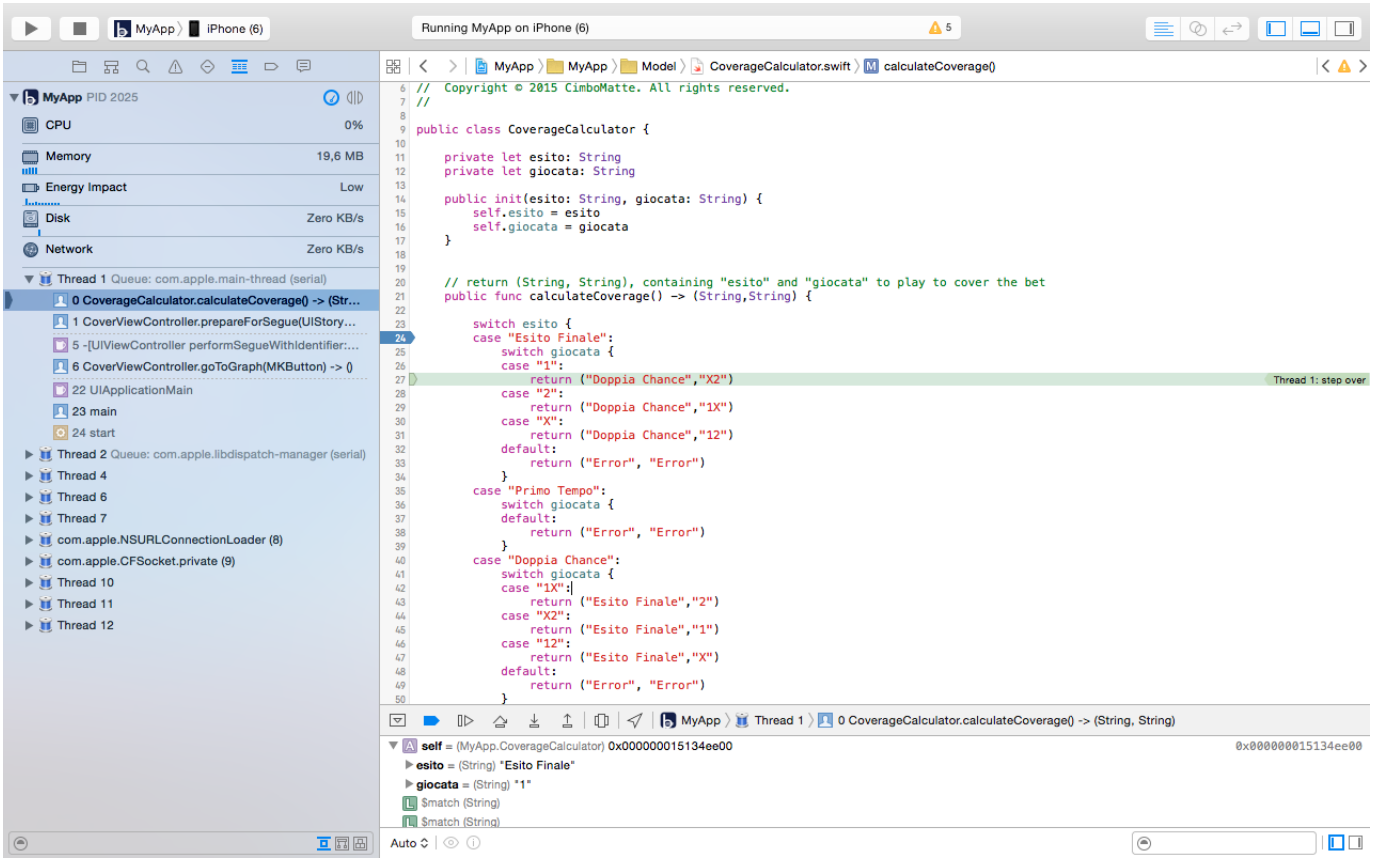


Figure 33: File White Box

13 Final Consideration

As we did not know the swift development environment, we decided to hold us to a higher level of abstraction in order to describe the functionalities of the system. In the specific we know that:

- the UX diagrams represent the screens used by the user to interface with the system
- the BCE diagrams provide a division of the levels of the system
- the sequence diagrams show the interaction between the system component in relation to the different functionalities