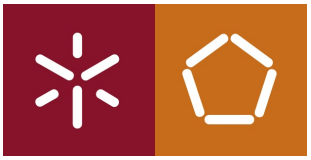


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Arquiteturas de Software 18/19

Plataforma de Apostas BetESS

João Pedro Ferreira Vieira A78468
Simão Paulo Leal Barbosa A77689

10 de Dezembro de 2018

Conteúdo

1 Introdução

O presente documento descreve o processo de desenvolvimento do primeiro trabalho prático da UC de *Arquiteturas de Software*. O problema proposto passa pela criação de dois produtos de *software* de uma aplicação de apostas BetESS: uma primeira versão sem padrões de *software*, e uma segunda versão com a aplicação de padrões de *software*. O objetivo passa por comparar as soluções obtidas e verificar as qualidades e diferenças entre as duas versões, com o intuito de verificar a importância e o porquê de cada vez mais serem utilizados diversos padrões no desenvolvimento de *software* nos dias de hoje.

2 Funções do sistema

Tendo em conta os requisitos funcionais apresentados no enunciado deste projeto prático, conseguimos depreender as funcionalidades/comportamentos que o sistema a desenvolver deve apresentar:

- Um apostador regista-se no sistema com o seu *email*, *password*, *nome* e o *saldo* da sua conta em ESScoins
- Um utilizador do sistema é capaz de fazer autenticação
- Um apostador tem acesso à lista de eventos da BetESS
- Um funcionário tem acesso à lista de eventos da BetESS
- A apresentação dos eventos aos utilizadores deve conter o nome das equipas envolvidas e as *odd's* respetivas
- Um apostador apenas pode apostar num único resultado (1, X ou 2) para um evento, indicando o resultado pretendido e o valor a apostar.
- Um apostador é capaz de importar uma quantia para o seu saldo
- O sistema mantém uma lista das apostas realizadas por cada apostador e este pode consultar as mesmas
- Um funcionário é capaz de criar e modificar eventos.
- Um funcionário é capaz de terminar um dado evento, indicando o resultado do mesmo
- Quando um evento é terminado, deve ser (ou não) atualizado o saldo dos apostadores que apostaram no evento.
- Quando um evento é terminado, os apostadores do mesmo devem ser notificados

Desta forma, um diagrama de *use cases* ajuda a demonstra o que os dois tipos de utilizadores do sistema (apostadores e funcionários) tem acesso no sistema:

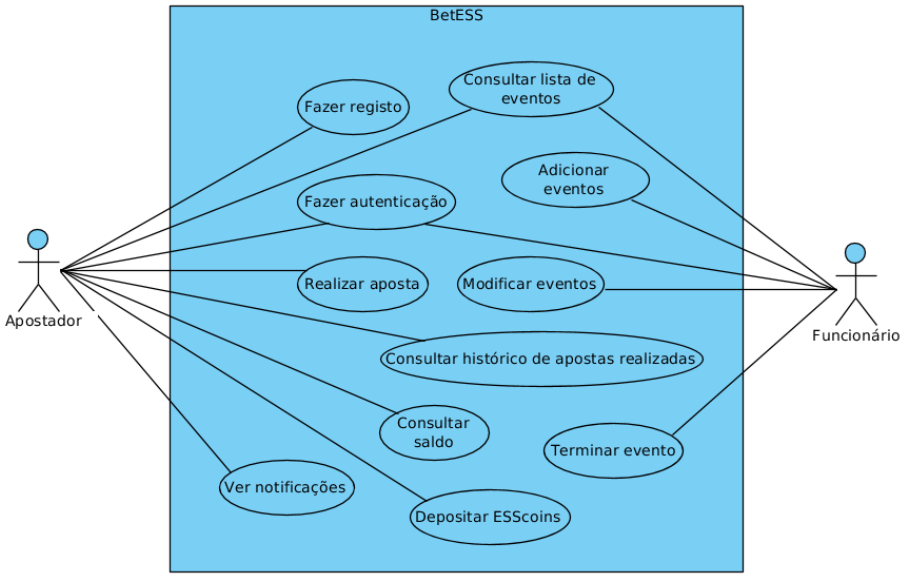


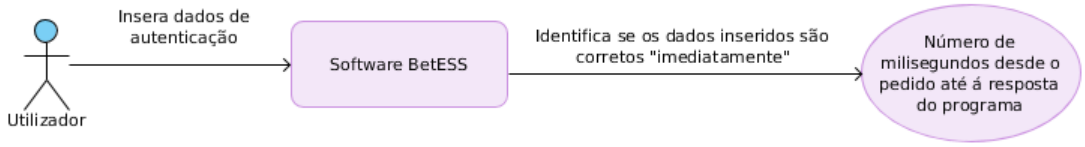
Figura 1: Funcionalidades ao dispor dos utilizadores

3 Versão sem padrões de software

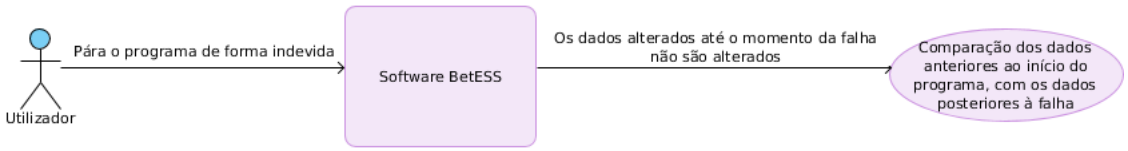
3.1 Atributos de qualidade

Tendo em conta as funcionalidades do sistema, foram por nós levantados os seguintes atributos de qualidade para o *software* a desenvolver:

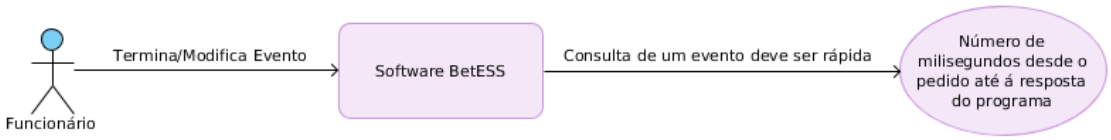
Verificar a autenticação de um utilizador de forma rápida



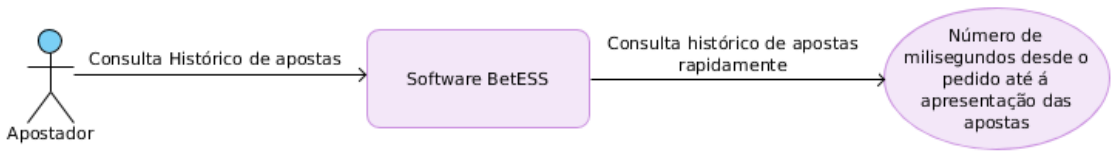
Caso o utilizador pare o programa de forma indevida, os dados alterados até ao momento da falha não são alterados



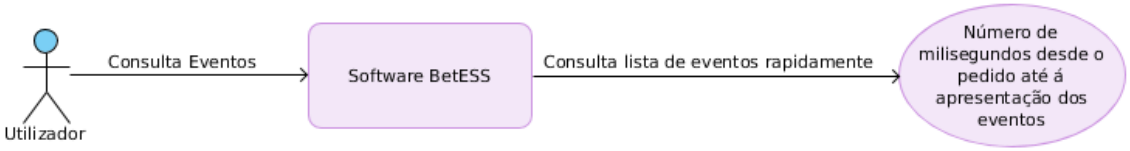
A consulta de um evento quando um funcionário termina ou modifica um evento deve ser rápida



Quando um utilizador pretende consultar o seu historial de apostas, este deve ser obtido rapidamente

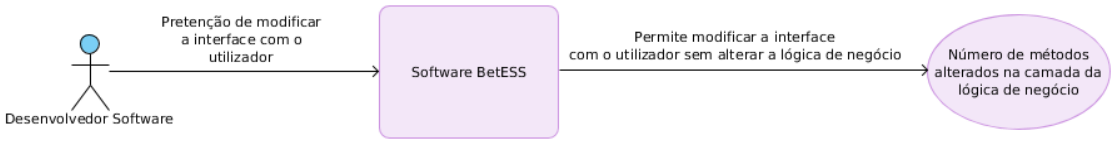


Quando um utilizador (Apostador ou Funcionário) pretende consultar a lista de eventos, esta informação deve ser obtida num curto espaço de tempo

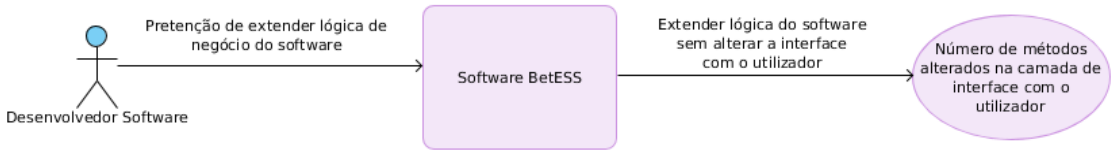


Em caso de necessidade de extensão da interface com o utilizador do produto, o mesmo deve ser conseguido sem alterações à lógica de negócio

Para modificar a interface com o utilizador do sistema, isto deve ser conseguido sem alterações à lógica de negócio

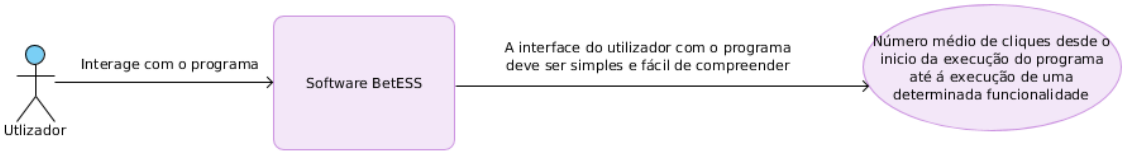


Extensões da parte lógica do *software* devem conseguir ser obtidas sem alterar a interface com o utilizador



Alterações da parte lógica do *software* devem ser conseguidas sem alterar a interface com o utilizador

O sistema deve ser simples e fácil de compreender rapidamente pelos seus utilizadores



3.2 Arquitetura da solução

Por forma a implementar a primeira fase deste enunciado, a arquitetura implementada passou pela utilização da arquitetura *MVC*, na qual temos 3 camadas distintas que pensamos se adequarem a este projeto: Camada Lógica (*Model*), Camada de Apresentação (*View*) e Camada de Negócio (*Controller*).
Desta forma, o diagrama de classes que demonstra o sistema de software desenvolvido nesta etapa do projeto é em baixo apresentado.

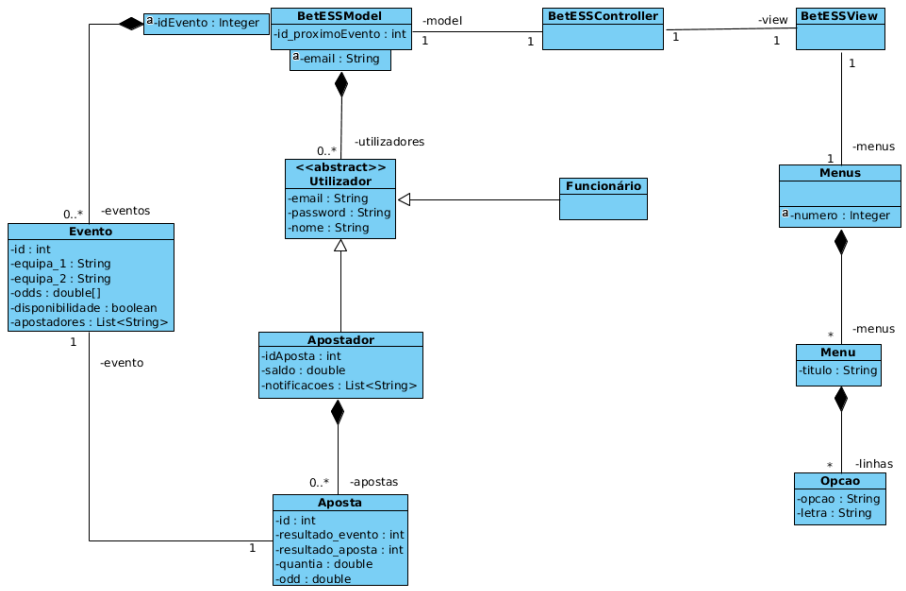


Figura 2: Diagrama de classes da 1ª Fase

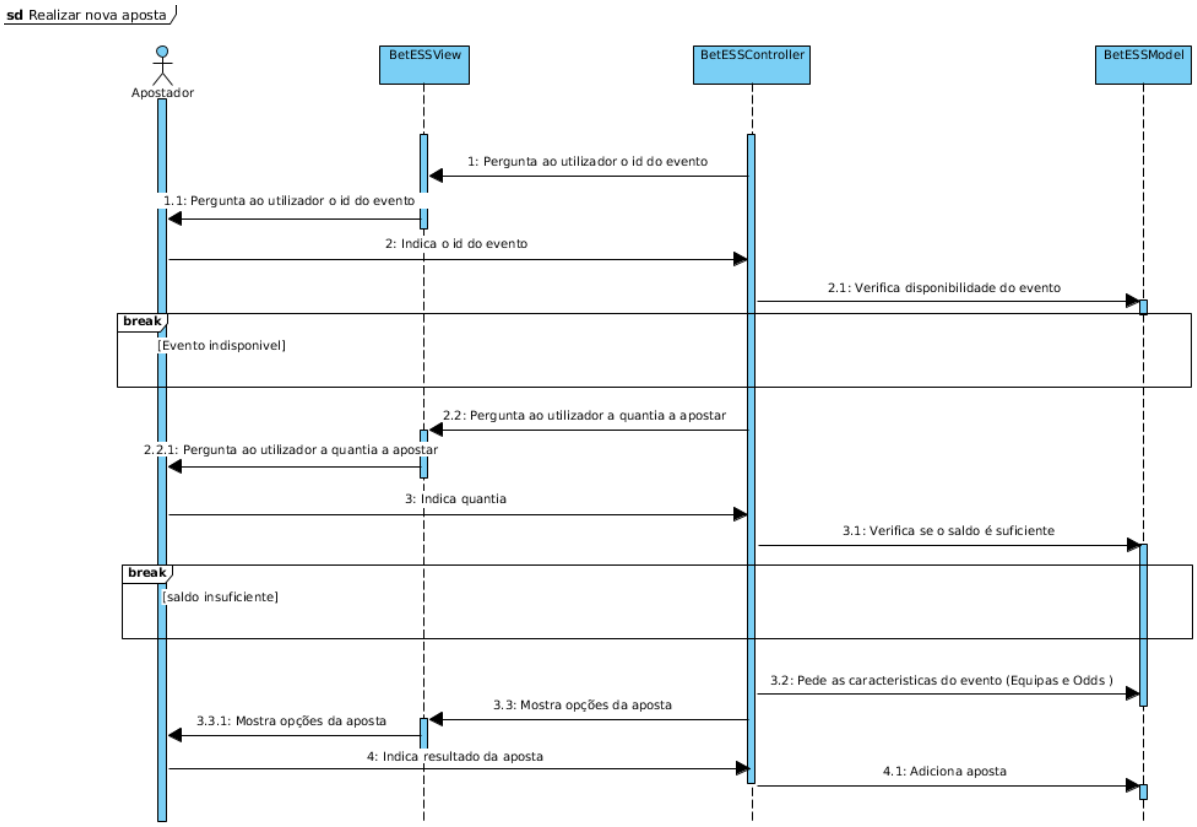
3.3 Implementação

Com a imagem do diagrama de classes em cima apresentada é possível perceber as classes implementadas e os seus componentes:

- **BetESSModel**: tal como referido anteriormente, é a camada lógica na arquitetura implementada, contem a lista de utilizadores do sistema, os eventos e um inteiro `id_proximoEvento` que é usado para definir o *id* dos eventos quando estes são criados.
- **Evento**: é a classe que representa um Evento da BetESS, contem um `id`, o nome das equipas envolvidas, as respetivas `odds` para os 3 possíveis resultados, a `disponibilidade` do mesmo, e a lista dos *emails* dos apostadores que realizaram apostas neste evento (`apostadores`).
- **Utilizador**: é uma classe abstrata utilizada tanto por Apostador como para Funcionário, esta classe faz com que cada utilizador tenha um `email`, uma `password` e um `nome`.
- **Funcionario**: extende a classe Utilizador.
- **Apostador**: extende a classe Utilizador, acrescentando um valor para o saldo da sua conta (`saldo`), um inteiro `idAposta` utilizado para definir o *id* das apostas feitas por este, uma lista de *String's* para as notificações (`notificacoes`) e lista das apostas realizadas (`apostas`).
- **Aposta**: representa uma aposta realizada por um apostador, contem um `id`, dois inteiros, um para representar o resultado apostado pelo apostador e outro que representa o resultado final da aposta, a `quantia` apostada, a `odd` a que a aposta foi realizada, e ainda um objeto *Evento* que representa o evento sobre o qual foi realizada a aposta.
- **BetESSController**: é a camada de negócio do sistema, onde são processadas todas as operações feitas pelos utilizadores.
- **BetESSView**: é a camada de apresentação do sistema, que trata de apresentar os menus e informações que o utilizador deve receber. Conta com a "ajuda"das classes **Menus**, **Menu** e **Opcao** para a representação dos diversos menus da plataforma.

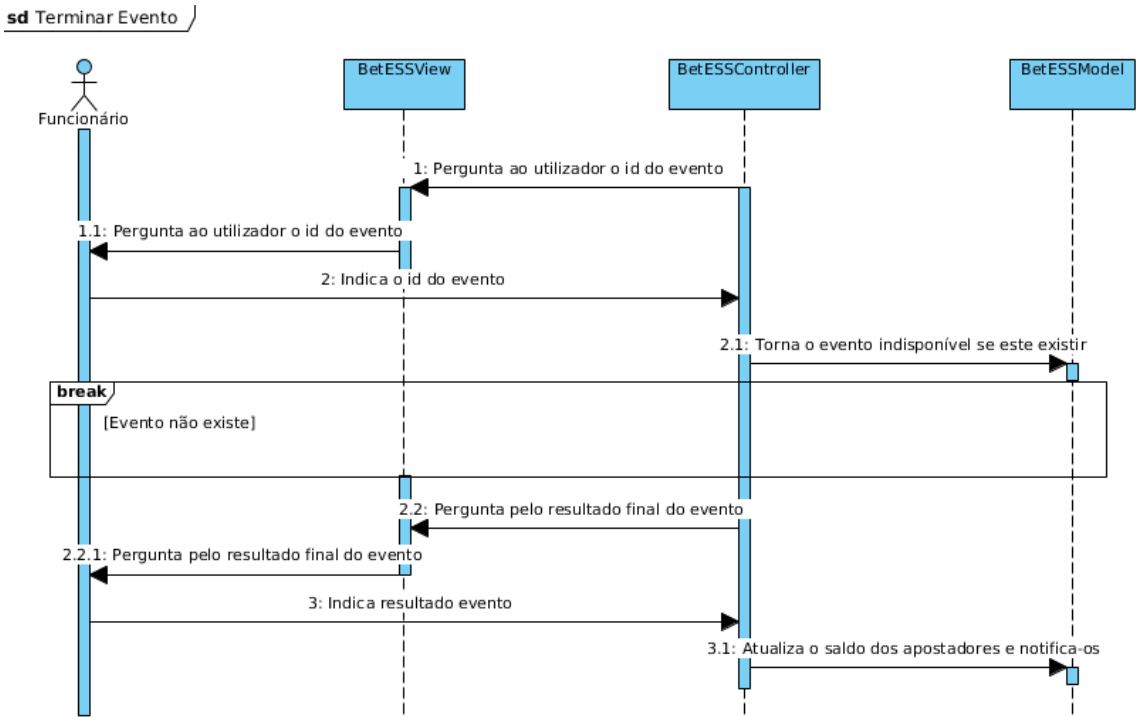
Como forma de demonstrar o procedimento por detrás de algumas operações efetuadas no sistema, são de seguida apresentados dois diagramas de sequência para duas das operações cruciais do sistema.

3.3.1 Apostador realiza aposta



Como se pode perceber através do fluxo que o diagrama de sequência apresenta, esta operação começa com o Apostador a ser questionado pelo *id* do Evento em que pretende apostar. Após a inserção do valor pedido, é verificado se o Evento está disponível (a sua existência e a disponibilidade para realizar aposta no mesmo).
Caso o evento esteja disponível, o Apostador volta a ser questionado, desta vez pela quantia que pretende apostar neste mesmo evento. Depois de indicada a quantia, é verificado se o saldo do mesmo permite realizar uma aposta de tamanha quantia.
Em caso afirmativo, são obtidas e mostradas ao Apostador as opções do Evento (equipas e respetivas *odd's*, assim como o possível ganho em cada resultado). A operação termina com o Apostador a indicar o resultado em que pretende apostar e a mesma aposta a ser registada.

3.3.2 Funcionário termina Evento



O procedimento desta operação inicia-se com o Funcionário a ser questionado pelo *id* do Evento que pretende encerrar. Após a inserção deste valor pedido, a disponibilidade do Evento é colocada a *false* (`private boolean disponibilidade;`). Caso o evento não exista a operação é cancelada.
Em caso de sucesso, o funcionário deve inserir o resultado final do evento depois de pedido pelo programa com a apresentação das 3 possibilidades. Após isto, o evento é dado como encerrado e é atualizado (ou não) o saldo de todos os Apostadores que apostaram neste mesmo Evento, recebendo uma notificação com a informação do mesmo.

4 Versão aplicando padrões de software

4.1 Novas funcionalidades

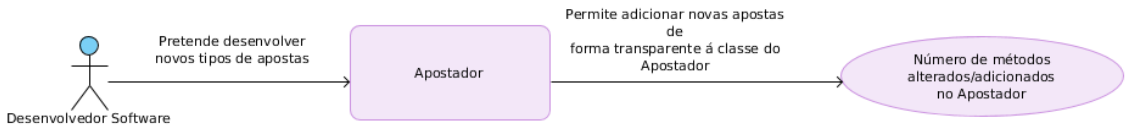
Nesta segunda fase do projeto, por forma a podermos não só implementar, mas também demonstrar o uso de padrões de *software*, adicionamos algumas funcionalidades ao sistema desenvolvido. Desta forma, são adicionadas as seguintes funcionalidades:

- Tanto os apostadores como os funcionários, ao consultar a lista de Eventos da aplicação, devem poder optar por:
  - Ver todas as apostas
  - Consultar os eventos ordenados pelo seu número de apostas
  - Procurar por eventos pela competição dos mesmos
  - Procurar por eventos pelo nome de uma equipa
- Os apostadores podem consultar as suas apostas:
  - Sem critério
  - Ordenadas por ganhos/perdas
  - Ordenadas por possíveis ganhos
  - Ordenados por valor apostado
- Os apostadores podem fazer apostas múltiplas, para além das apostas simples da 1ª fase

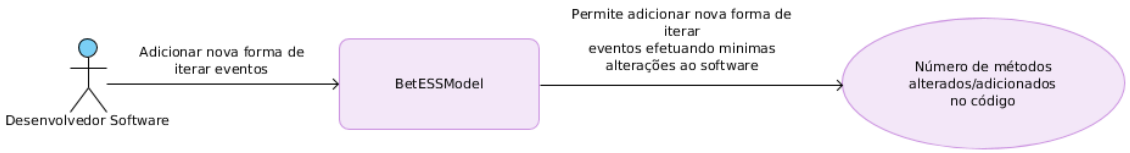
4.2 Atributos de qualidade

Tendo em conta as funcionalidades que o sistema deve implementar, foram levantados os seguintes atributos de qualidade para o *software* a desenvolver:

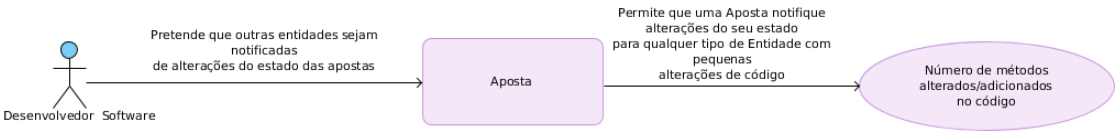
Ser fácil adicionar novos tipos de apostas no futuro.



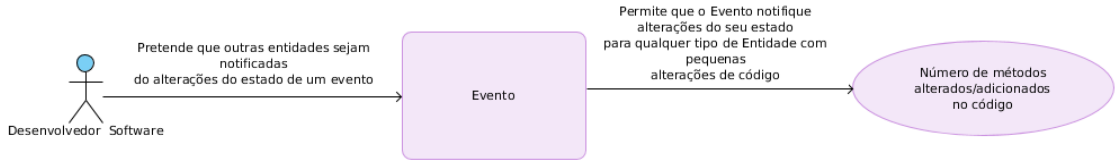
Iterar eventos de diferentes formas, sendo fácil extender no futuro.



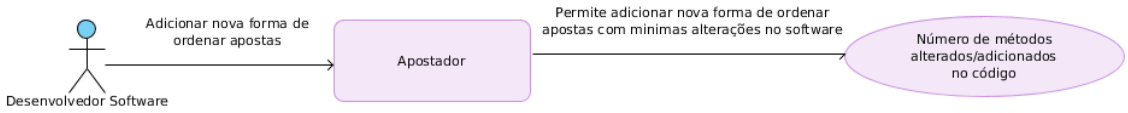
Ser possível entidades serem observadoras de apostas



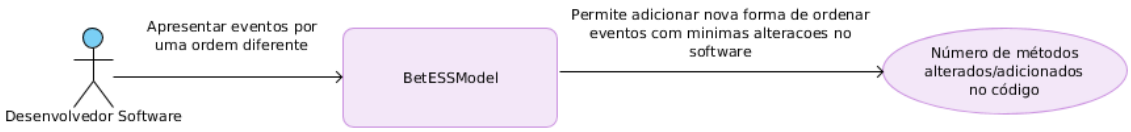
Notificar entidades de alterações do estado de eventos



Ordenar apostas segundo diferentes critérios e ser de fácil extensão/modificabilidade



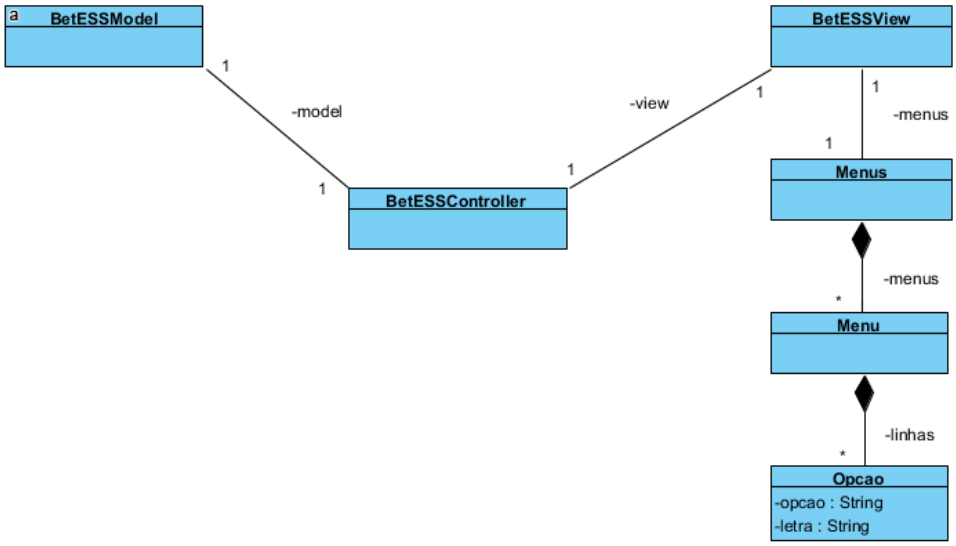
Ordenar eventos segundo diferentes critérios e ser de fácil extensão/modificabilidade



4.3 Arquitetura da solução

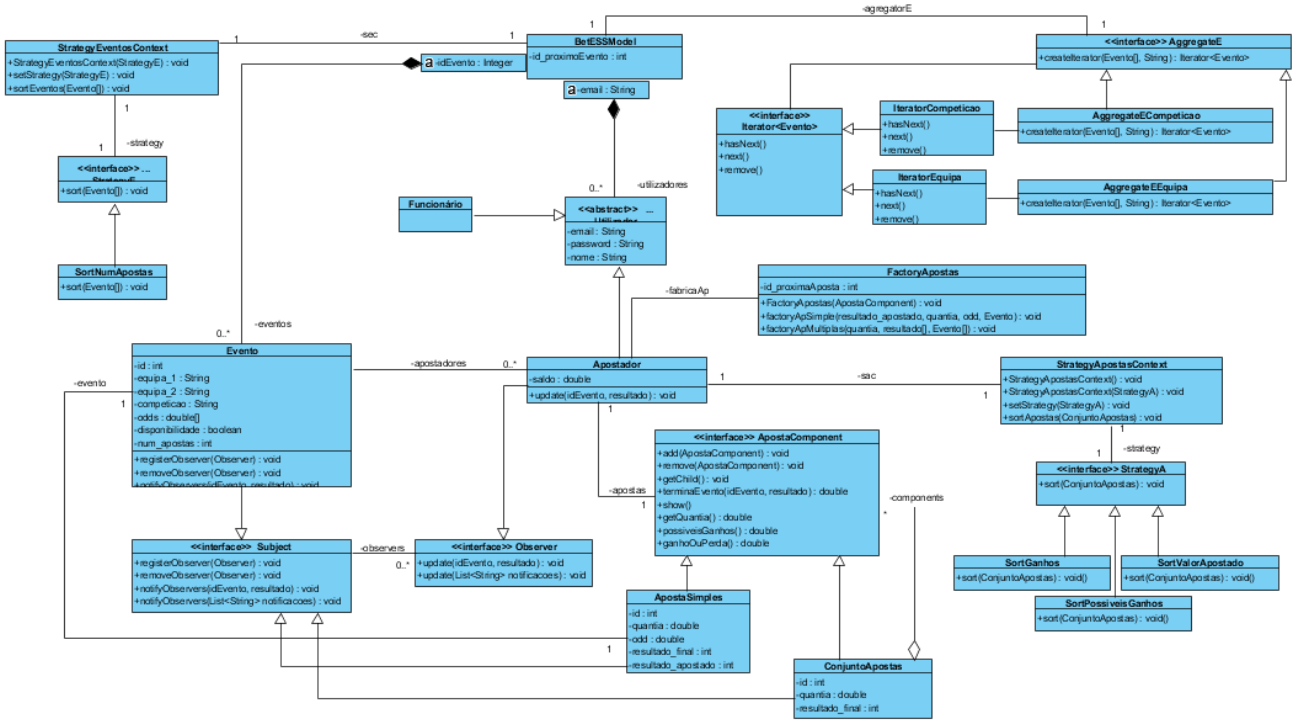
4.3.1 Padrão Arquitetural utilizado

Decidimos utilizar o padrão arquitetural MVC para a nossa arquitetura, pois consideramos ser o mais adequado ao problema e capaz de tornar o *software* desenvolvido mais extensível e modificável. Separamos as classes em 3 camadas distintas: Camada Lógica (*Model*), Camada de Apresentação (*View*) e Camada de Controlo (*Controller*).



As classes **Menus**, **Menu** e **Opcao**, tal como explicado anteriormente na primeira parte deste projeto, são utilizados para a representação dos diversos menus da plataforma desenvolvida.

A arquitetura geral de **BetESSModel** é a seguinte:



Como através desta imagem é difícil perceber todos os padrões utilizados pos nós, os mesmos são expostos e explicados de seguida de forma individual.

4.3.2 Padrões de Desenho utilizados

Utilizamos os seguintes padrões de desenho de *software* na nossa arquitetura, de modo a conseguir implementar os atributos de qualidade definidos e apresentados anteriormente.

- **Observer**

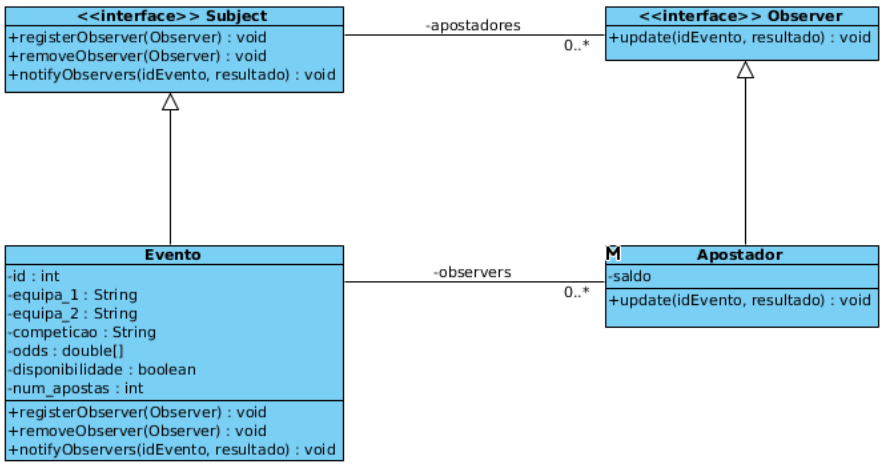


Figura 3: Apostador é um observador de Evento

De forma a conseguirmos melhorar a reusabilidade da nossa arquitetura, optamos por tornar o Evento observável (*Subject*) para qualquer entidade que seja observador (*Observer*) ter a possibilidade de ser notificado de alterações nos eventos, especialmente quando este termina e se sabe o resultado final do mesmo.
No caso concreto do diagrama em cima apresentado, implementamos o padrão *Observer* colocando Evento como sendo observável pelos Apostador(es) do mesmo evento, de forma a estes serem notificados e o seu saldo atualizando quando o mesmo evento é encerrado.
Com o uso destas interfaces, qualquer novo tipo de Utilizador adicionado ao sistema pode ser tornado um observador de Evento, e assim notificado pelo mesmo, bastando implementar a interface *Observer* e definir qual o tipo de *update* a realizar.
Da mesma forma, tornamos as apostas observáveis (*Subject*):

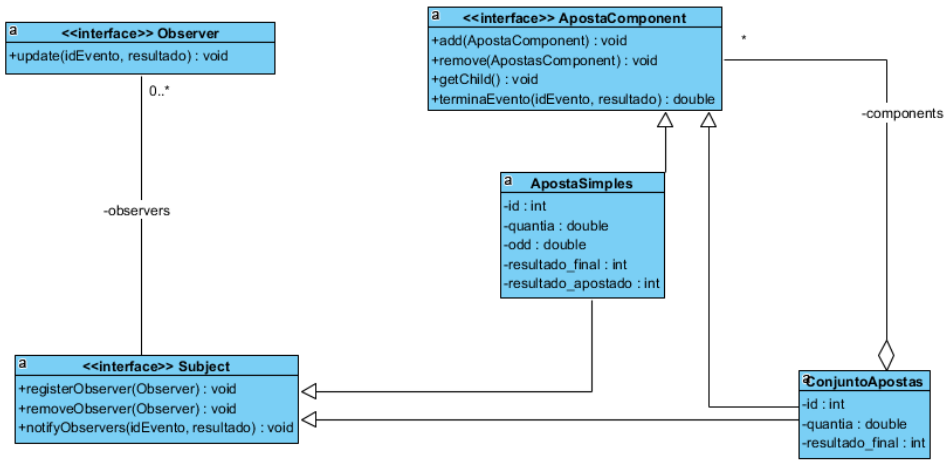
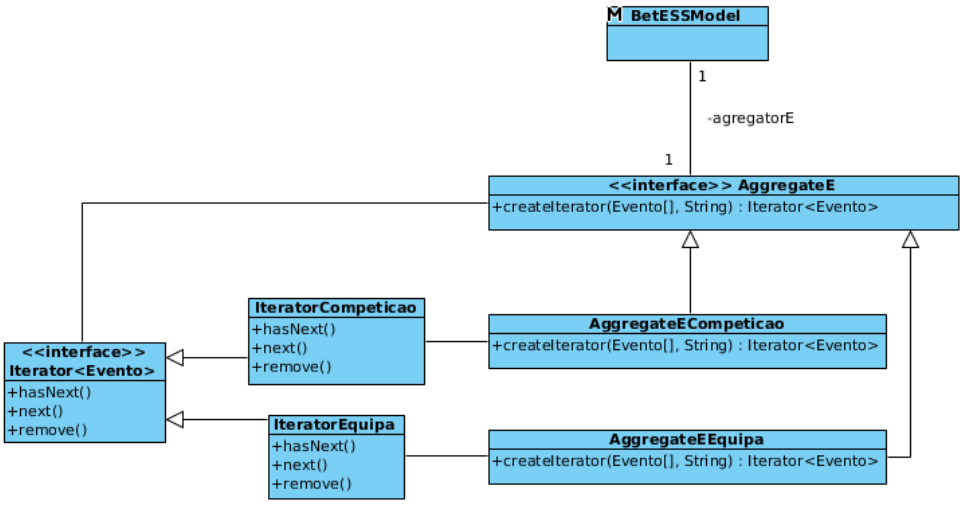


Figura 4: As apostas implementam a interface *Subject*

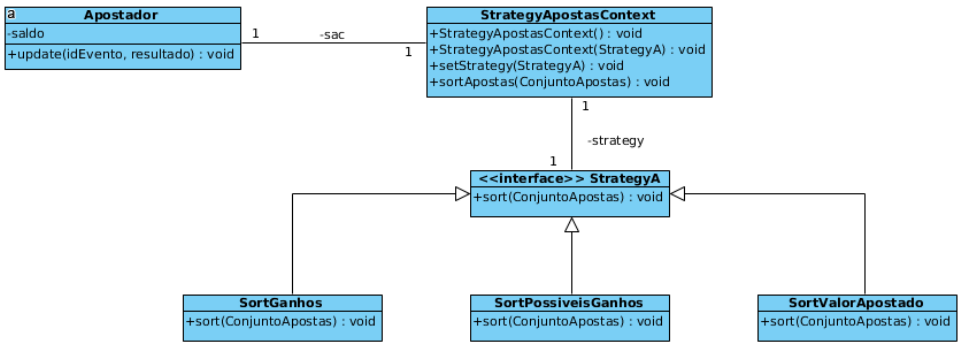
- **Iterator**



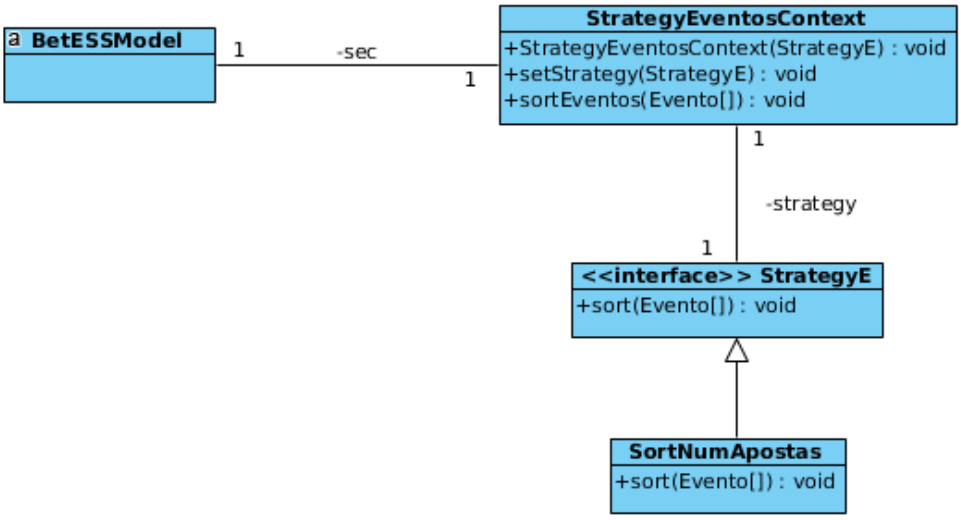
Com o objetivo de aumentar a extensibilidade e modificabilidade do produto desenvolvido, nomeadamente da classe Evento, utilizamos este desenho de padrão para iterar a lista de Eventos de diferentes maneiras e para permitir futuramente adicionar novas formas de a iterar facilmente.
As formas de iterar desenvolvidas são implementadas com o intuito de poder facilitar aos utilizadores a procura

pelos eventos em que desejam apostar.
Decidimos para já desenvolver 2 iteradores, um iterador que encontra eventos de uma determinada equipa (`IteratorEquipa`) dado um termo de pesquisa, e outro iterador que encontra eventos de uma determinada competição (`IteratorCompeticao`) também com um termo de pesquisa fornecido pelo utilizador. Para isso criamos os *Aggregatores*: `AggregateEEquipa` e `AggregateECompeticao`, que são utilizados para criar os iteradores mencionados anteriormente.

- **Strategy**

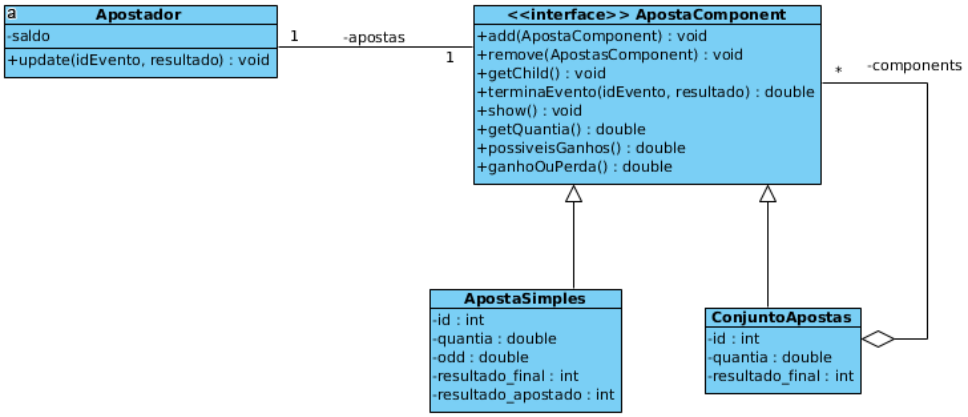


Com o objetivo de aumentar a extensibilidade e modificabilidade da nossa arquitetura, decidimos utilizar este desenho de padrão para tornar possível ordenar as Apostas do Apostador e futuramente adicionar novas formas de ordenação com pouca dificuldade. Assim, os Apostadores poderão visualizar as suas apostas de diversas ordens.
Para isso desenvolvemos uma interface `StrategyA` que tem o método `sort(ConjuntoApostas)`. Definimos várias classes que implementam esta interface, que ordenam as apostas de diferentes formas: pelos ganhos conseguidos nas apostas (`SortGanhos`), pelos possíveis ganhos (`SortPossiveisGanhos`) e pela quantia apostada (`SortValorApostado`). O Apostador tem uma classe nas suas variáveis (`sac`, do tipo `StrategyApostasContext`) para escolher a estratégia a ser utilizada.
Seguindo a mesma lógica e objetivo, desenvolvemos o mesmo padrão para os eventos do sistema, que pertencem ao `BetESSModel`.

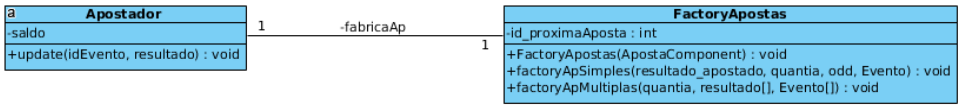


Neste caso, a ordenação oferecida de momento aos utilizadores para ver a lista de Eventos é pelo número de apostas nos mesmos (`SortNumApostas`).

- **Composite**
  Utilizamos o desenho de padrão *Composite* para estruturar as apostas dos apostadores, de forma a melhorar a extensibilidade e modificabilidade das mesmas. Assim se futuramente se pretender desenvolver novos tipos de apostas para o sistema, a aplicação do mesmo será mais simples, pois as operações sobre estas são similares, será sempre necessário terminá-las quando os eventos acabam e também apresentá-las aos utilizadores.
  Desenvolvemos apostas simples (iguais ás apostas da 1ºfase, `ApostaSimples` nesta implementação) e apostas múltiplas (`ConjuntoApostas`). Qualquer operação sobre o nodo inicial implica a execução dessa operação em toda a árvore (importante no terminar de um evento).



- **Factory Method**
  Para o correto desenvolvimento do desenho de padrão apresentado anteriormente, precisámos de uma forma de inicializar as instâncias da classe *ApostaComponent*. Para tal, utilizamos este desenho de padrão (*Factory Method*). Criámos a classe *FactoryApostas* que tem 2 métodos essenciais, um para instanciar apostas do tipo simples e outro para instanciar apostas múltiplas:
  `* public void factoryApSimples(resultado_apostado, quantia, odd, Evento);`
  Recebe como parâmetros o resultado apostado, a quantia apostada, a *odd* a que realiza a aposta e o Evento no qual está a apostar.
  `* public void factoryApMultiplas(quantia, resultado[], Evento[]);`
  Recebe como parâmetros a quantia apostada, a lista de resultados e a lista de Eventos, sendo que `resultado[i]` é o resultado apostado no Evento `Evento[i]`.



4.4 Implementação

4.4.1 Classes Implementadas

Implementamos a modelação anterior nos seguintes *packages*:

- **Main**
  - **BetESS_App** –>Classe **main** do sistema, inicializa o **Model**,**View** e **Controller** e inicia a aplicação
  - **BetESS_Persistency** –>Classe responsável pela persistência dos dados, carrega dados no início e guarda-os no fim da execução
- **Controller - Camada de Negócio**

  - **BetESSController** –>Camada de negócio, conecta a camada da lógica com a camada de visualização

- **Model - Camada Lógica**
  - **BetESSModel** –>Camada lógica, contém eventos, utilizadores, agregadores e estratégia de ordenação de eventos
  - **Apostador** –>Extende **Utilizador**. Contém saldo, as suas apostas, fábrica de apostas, estratégia de ordenação de apostas e as suas notificações. Implementa **Observer**, observa os eventos sobre os quais efetuou apostas para ser notificado de alterações neste.
  - **Evento** –>Implementa **Subject**, para notificar os seus observadores (Apostador) quando termina um evento. Contém lista de observadores e variáveis descritivas do Evento.
  - **Funcionario** –>Classe que extende Utilizador.
  - **Utilizador** –>Classe abstrata. Contém as variáveis *email*, *password* e *nome*.

- **Model-Composite**
  - **ApostaComponent** –>Interface de apostas, **Component** do padrão de desenho **Composite**
  - **ApostaSimples** –>Implementação da uma aposta simples. **Leaf** do padrão Composite e **Subject** do padrão **Observer**
  - **ConjuntoApostas** –>Lista de ApostaComponent , **Composite** do padrão **Composite** e **Subject** do padrão **Observer**. Usada para implementação de apostas múltiplas

- **Model-Factory**
  - **FactoryApostas** –>Classe que instância diferentes tipos de aposta, uso do **Factory Method**

- **Model-Iterator**
  - **AggregateE** –>Interface para criar Iteradores de eventos.
  - **AggregateECompeticao** –>Classe que implementa **AggregateE**.
  - **AggregateEEquipa** –>Classe que implementa **AggregateE**.
  - **IteratorCompeticao** –>Implementa **Iterator<Evento>**
  - **IteratorEquipa** –>Implementa **Iterator<Evento>**

- **Model-Observer**
  - **Observer** –>Interface do observador no padrão **Observer**
  - **Subject** –>Interface de observável no padrão **Subject**

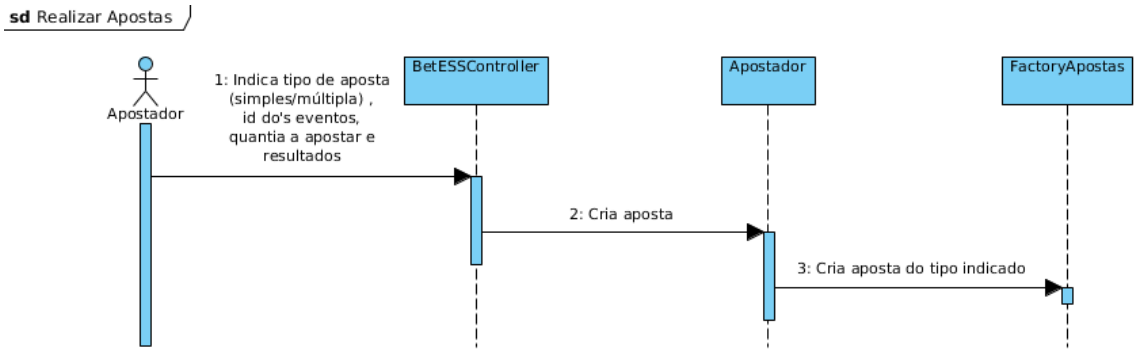
- **Model-Strategy**
  - **SortGanhos** –>Implementa **StrategyA**, utilizado para ordenar apostas
  - **SortNumApostas** –>Implementa **StrategyE**, utilizado para ordenar eventos
  - **SortPossiveisGanhos** –>Implementa **StrategyA**, utilizado para ordenar apostas
  - **SortValorApostado** –>Implementa **StrategyA**, utilizado para ordenar apostas
  - **StrategyE** –>Interface do padrão **Strategy** para eventos
  - **StrategyEventosContext** –>Classe usada para escolher estratégia a utilizar para ordenar eventos
  - **StrategyA** –>Interface do padrão **Strategy** para apostas
  - **StrategyApostasContext** –>Classe usada para escolher estratégia a utilizar para ordenar apostas

- **View - Camada de Visualização**
  - **BetESSView** –>Classe que contém os menus a serem vistos pelos utilizadores e outras funções auxiliares
  - **Menu** –>Contém uma lista de opções (**Opcao**) e um título.
  - **Menus** –>Contém um conjunto de menus (**Menu**).
  - **Opcao** –>Contém duas `String's`, uma é a descrição da opção para o utilizador, a outra é letra que o utilizador precisa de introduzir para selecionar essa opção

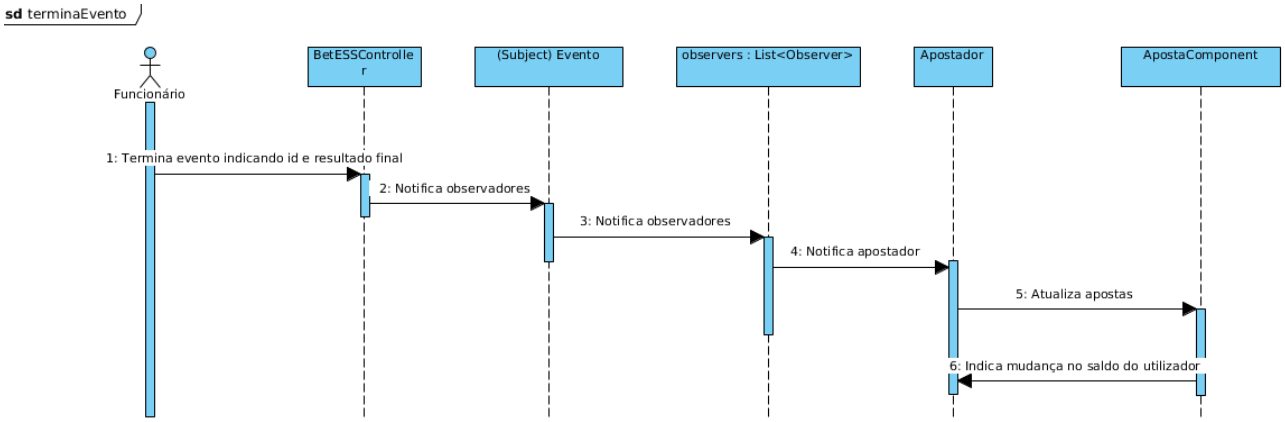
4.4.2 Comportamento do Sistema

Para exemplificar o uso dos padrões em cima apresentados, apresentámos aqui o fluxo de algumas operações realizadas no sistema, realizadas com recurso às classes pertencentes aos padrões implementados.

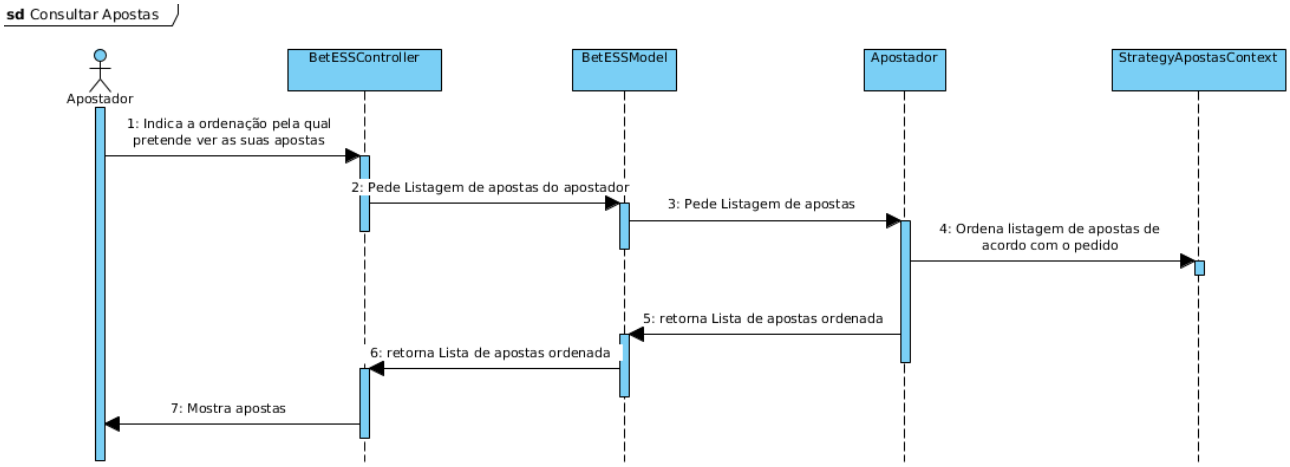
- Apostador realiza aposta



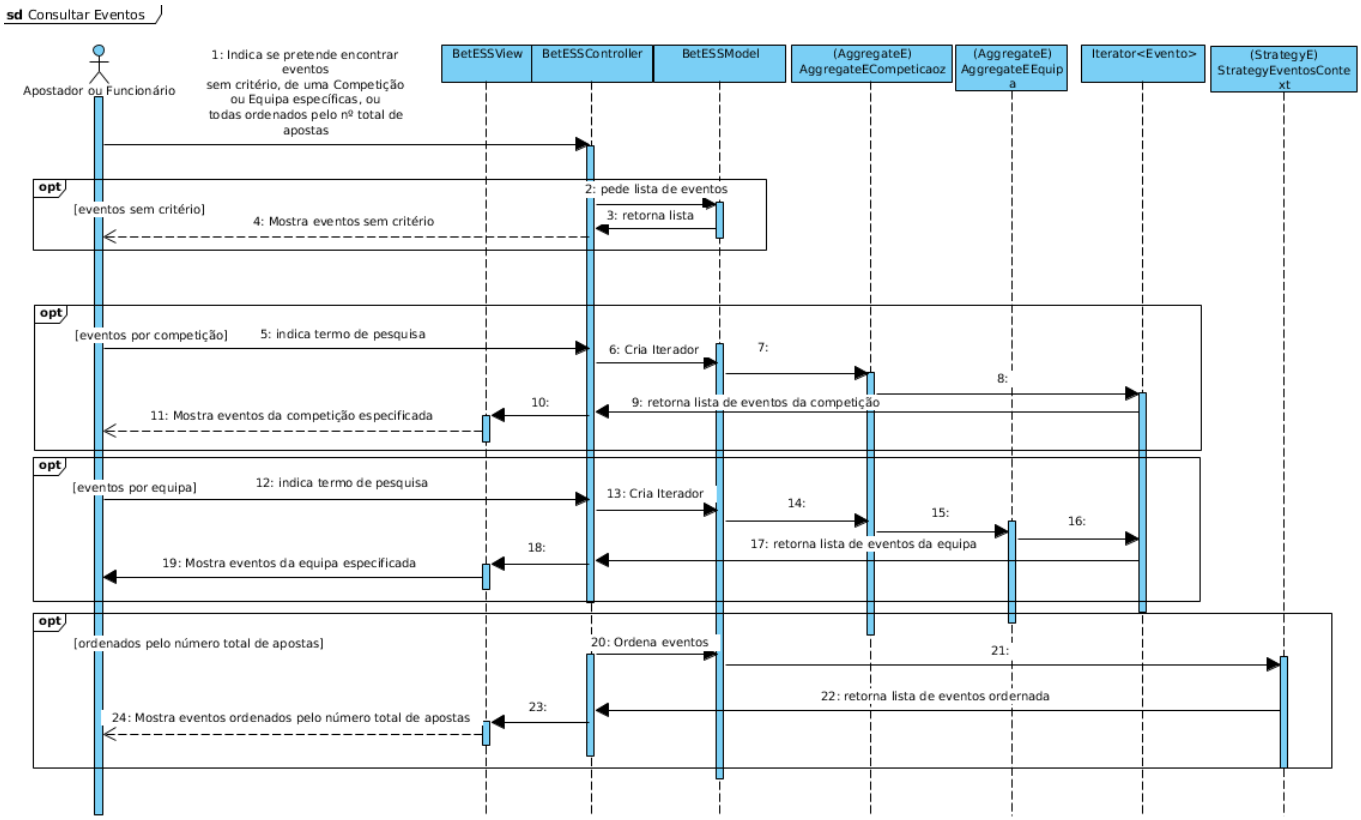
- Funcionário encerra um determinado evento



- Apostador consulta as suas apostas com uma dada ordem



- Apostador ou Funcionário consultam os eventos através das 4 opções oferecidas



4.5 Implementação do novo requisito - *Bookie*

Tendo em conta a arquitetura pensada e implementada por nós à altura, o aparecimento do novo requisito relacionado com os *BookMakers* não nos levou a efetuar muitas modificações no código do sistema.
Como no nosso sistema já existiam os utilizadores do tipo **Funcionario**, estes basicamente eram pensados como tendo funções semelhantes aos *Bookie's*, tendo em conta que os funcionários da BetESS já eram capazes de ver a lista de eventos, criar e modificar eventos, e ainda encerrá-los.
Sendo assim, para implementar a possibilidade de um **Funcionario** receber notificações de uma lista de eventos à sua escolha, as alterações efetuadas foram as seguintes:

- é adicionada no menu do **Funcionario** uma nova opção para poder seguir Eventos, sendo o tratamento dos dados inseridos realizado no *Controller*
- **Funcionario** passa a implementar a interface `Observer` do padrão observador, de forma a poder ser um observador de **Evento**, e passa a ter uma lista de notificações: `private List<String> notificacoes;`
- É adicionado um novo método *update* à interface de observador para ser implementada em **Funcionario**: `public void update(int idEvento, String equipa_1, String equipa_2, double valor);`
- Em **Evento**, é adicionada uma variável `private double balanco;` iniciada a 0, que sempre que um **Apostador** realiza uma aposta simples na mesma é atualizada/incrementada com o valor da quantia apostada, e sempre que um **Apostador** ganha uma aposta, a quantia ganha por ele é decrementada na mesma
- Quando o **Evento** é terminado por um **Funcionario**, é realizada a notificação aos seus observadores (que agora passam a ser do tipo **Apostador** e **Funcionario**), tendo em conta o seu tipo. A notificação com a informação do fim do **Evento** e o seu balanço económico passa a estar disponível para ser lida da próxima vez que o **Funcionario** entrar na sua área, e, depois de lida, é removida da sua lista de notificações

Já para implementar a possibilidade de um **Funcionario** receber notificações de uma lista de apostas à sua escolha, as alterações efetuadas foram as seguintes:

- é adicionada no menu do **Funcionario** uma nova opção para poder seguir Apostas, sendo possível ver a lista de Apostas por utilizador (*email*) e decidir qual ou quais pretende seguir e receber notificações
- é adicionada nas classes **ApostaSimples** e **ConjuntoApostas** uma `String` com o *email* do **Apostador** da mesma para ajudar o processo.
- Como **ApostaSimples** e **ConjuntoApostas** implementam a interface `Observer`, o **Funcionario** passa a fazer parte da lista de observadores das apostas.
- Quando é realizado *update* nos Apostadores quando um Evento termina e se verifica uma alteração de estado de uma dada Aposta, esta passa agora a ser notificada aos observadores da mesma (agora os Funcionários), indo a notificação para a lista de notificações que o **Funcionario** contem (explicada anteriormente), e lida e tratada do mesmo modo que as notificações de Eventos. É utilizado o seguinte método na interface `Observer` para o efeito:
  `public void update(List<String> notificacoes);`

5 Conclusão

Com o concluir do desenvolvimento deste projeto, consideramos que na segunda etapa do mesmo acabamos por desenvolver um sistema consistente, com o uso dos padrões de *software* lecionados nesta UC. Conseguimos ter uma melhor noção de que apesar de ser mais trabalhoso face à primeira fase, a possibilidade de extender e modificar o sistema montado tende a ser menos trabalhoso.
Como possíveis melhorias ao projeto, poderiam ser ainda implementados alguns padrões que poderiam tornar o produto melhor, usar por exemplo o padrão *Composite* em Eventos, na possibilidade de um dia mais tarde serem adicionados diferentes tipos de Eventos ao formato atual. Para além disto, podiamos ter utilizado o padrão *Iterator* para iterar Apostas segundo um dado critério, seguindo a mesma lógica executada no iterador de Eventos. Outro aspeto a melhorar seria a implementação da persistência com recurso a uma Base de Dados, para guardar os dados alterados de imediato, em vez de guardar os dados num ficheiro de dados no final da execução do programa. Desta forma seria possível a alteração de dados em simultâneo por vários utilizadores.