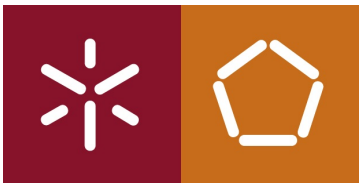

Universidade do Minho
Escola de Engenharia

Arquiteturas de Software
Engenharia de Sistemas de Software

BetESS - Plataforma de Apostas

Autores:	**Número:**
Daniel Maia	A77531
Vitor Peixoto	A79175

19 de Dezembro de 2018

Conteúdo

1 Introdução

Este trabalho prático tem como objetivo a modelação e implementação de uma plataforma de apostas, denominada **BetESS**. Para tal, será feito, numa primeira fase do projeto, o levantamento dos requisitos funcionais do sistema. A partir destes, derivar-se-ão os respetivos requisitos de qualidade e será feita a modelação do sistema recorrendo a diagramas da linguagem UML.

Por fim, será feita a implementação do produto. Serão criadas duas versões do mesmo. Uma, elaborada sem recorrer a padrões de *software* e outra na qual são aplicados os mesmos, derivados dos atributos de qualidades definidos pelos respetivos requisitos. Para cada uma, será apresentada a arquitetura da solução, detalhando as suas vistas de estrutura, bem como a implementação da mesma.

No final, será feito um balanço do trabalho efetuado, dos resultados obtidos e do possível trabalho futuro com a plataforma *BetESS*.

2 Requisitos

2.1 Requisitos funcionais

O levantamento de requisitos é uma das partes essenciais no que toca à inicialização da modelação do *software* a desenvolver. Os requisitos funcionais focam na parte comportamental do sistema, ou seja, como este deverá reagir às interações do utilizador e quais funcionalidades este lhe deverá assegurar.

Analisando o pedido pelo enunciado e tendo em conta a nossa abordagem para este trabalho prático, conseguimos definir e desenvolver os seguintes requisitos:

- O sistema deverá permitir um novo apostador registar no sistema com email, nome, *password* e uma dada quantidade de saldo.
- A unidade monetária utilizada na plataforma será a *ESSCoin*.
- O apostador só poderá interagir com o sistema após ser autenticado com o seu email e *password*.
- O apostador poderá visualizar os eventos ativos a qualquer momento.
- O apostador poderá efetuar uma aposta sobre qualquer evento aberto.
- O apostador só pode efetuar uma aposta por evento.
- O apostador poderá visualizar todas as apostas efetuadas por si.
- O apostador deverá ser notificado sobre o término dos eventos sob os quais tem apostas efetuadas.
- O apostador poderá reverter uma aposta feita por si, mas apenas antes do evento se iniciar, com a devolução da quantia apostada.
- O apostador deverá ter sempre, no mínimo, 5 *ESSCoins* depositados na plataforma, sendo que novos clientes deverão receber como oferta essas 5 *ESSCoins*.
- Um evento é composto por um jogo entre duas equipas da Primeira Liga Portuguesa de Futebol Profissional.
- Um evento tem dois estados possíveis: Aberto ou Fechado.
- Um evento, aquando da sua criação, o seu estado inicial deverá ser aberto e os utilizadores poderão registar imediatamente as suas apostas.
- Um evento, quando declarado fechado, o seu resultado deverá ser introduzido e os prémios automaticamente entregues aos vencedores da aposta.
- Um evento poderá ser apostado de três formas possíveis: vitória para a equipa da casa; empate; vitória para a equipa forasteira.

- Uma aposta envolve a seleção de um dos três resultados possíveis e de uma quantia em valores inteiros em *ESSCoins*.
- Cada equipa só poderá ter um evento ativo.
- O sistema deverá ter um administrador que irá gerir os eventos.
- O administrador só poderá interagir com o sistema após ser autenticado com as suas credenciais próprias.
- O administrador poderá criar novos eventos.
- O administrador poderá fechar um evento, introduzindo os resultados do mesmo.
- O administrador poderá definir as *odds* para o evento criado.

2.2 Requisitos de qualidade

Tendo averiguado as funcionalidades do sistema, procedeu-se à tradução destas nas qualidades necessárias pelas quais se avaliará o sucesso do sistema. Estes requisitos são satisfeitos pela estruturação da arquitetura e pela implementação do código a ser desenvolvido, possivelmente com o auxílio de padrões de *design* já existentes.

Com os requisitos recolhidos, conseguimos derivar então os seguintes atributos de qualidade e os respetivos requisitos:

2.2.1 Desempenho

O desempenho incide no tempo de resposta aos eventos provocados pelos utilizadores do sistema.

O facto de a aplicação ser *offline* permite que acessos múltiplos não existam, facilitando o desempenho da aplicação.

No entanto, pelos requisitos recolhidos, foi derivado um requisito relativamente ao tempo de execução das tarefas, nomeadamente sobre o **tempo das transações**. Exige-se assim que o tempo de qualquer transação (registo de aposta, criação de evento, anulação de aposta, etc.) efetuada por qualquer utilizador (apostador ou administrador) não ultrapassasse um segundo. Esse requisito foi especificado no seguinte esquema:

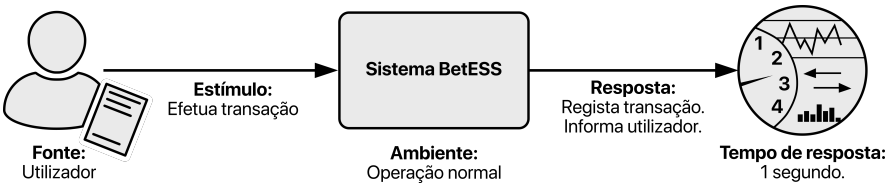

Figura 1: Requisito de qualidade #1.

2.2.2 Modificabilidade

A modificabilidade incide nos custos e consequências que as mudanças aplicadas ao sistema podem trazer. Este atributo de qualidade é fortemente arquitetural.

Relativamente a este atributo de qualidade, foi derivado um requisito. Esse requisito incide na necessidade de o sistema ser capaz de receber **novas funcionalidades** sem grandes dificuldades na implementação do novo código, sendo que essa funcionalidade deve conseguir ser implementada no período máximo de 24 horas.

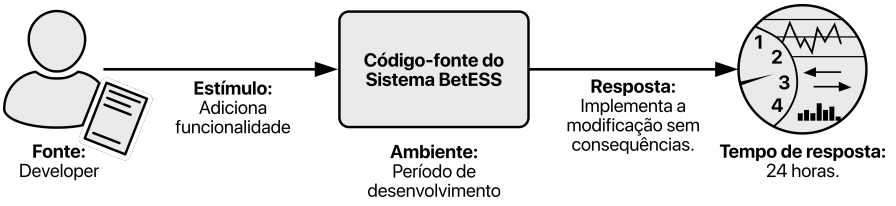

Figura 2: Requisito de qualidade #2.

2.2.3 Facilidade de uso

Os utilizadores desta aplicação serão utilizadores de diversas faixas etárias, logo torna-se necessário que a aplicação tenha uma *interface* gráfica de **fácil uso** e sem elementos desnecessários, facilitando a interação com o utilizador.

O requisito levantado incidiu na necessidade do utilizador familiarizar-se com todas as funcionalidades do sistema num período máximo de 2 horas.

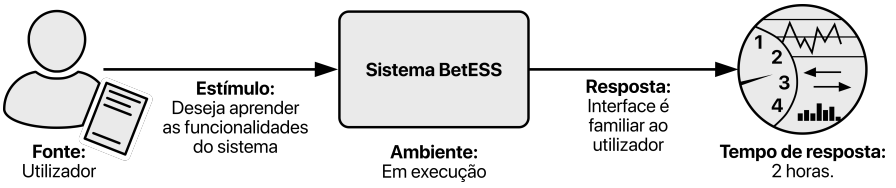

Figura 3: Requisito de qualidade #3.

3 Versão sem padrões de *software*

Inicialmente foi-nos pedido que desenvolvêssemos um *software* que satisfizesse os requisitos descritos no capítulo anterior, sem ter em conta a organização arquitetural do sistema ou a implementação de padrões de *design* que auxiliem no desenvolvimento do projeto.

3.1 Modelação

A modelação assume um papel fulcral na transcrição dos requisitos do sistema para uma solução funcional. Os modelos representam uma vista simplificada do sistema a desenvolver e permitem antecipar erros e documentar as decisões tomadas.

3.1.1 Modelo de Domínio

O modelo de domínio é a base de toda a modelação, pois captura os principais objetos no contexto do problema e os relacionamentos existentes entre eles. Assim, analisando os requisitos do problema, conseguimos desenvolver o seguinte modelo de domínio:

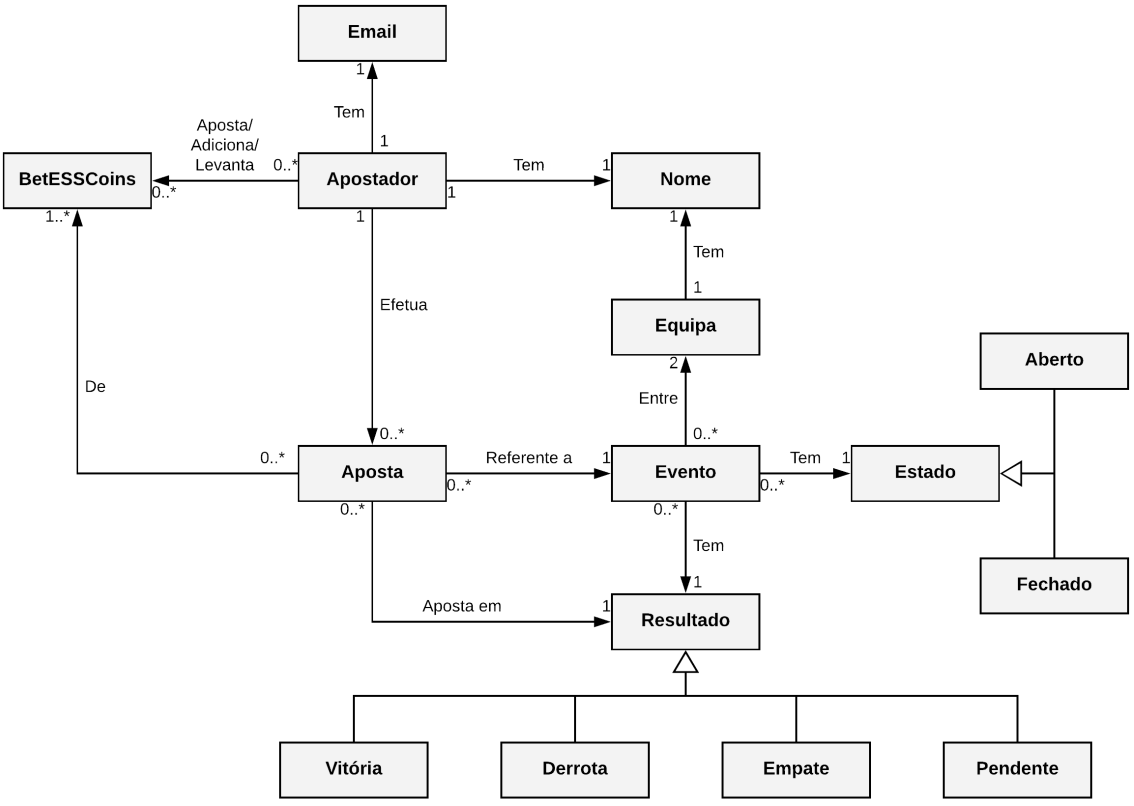

Figura 4: Modelo de domínio.

3.1.2 Diagrama de *Use Case*

Os diagramas de *use case* permitem identificar os atores intervenientes no sistema e as funcionalidades que o sistema deve suportar a esses intervenientes. Pelos requisitos levantados, consegue-se claramente definir dois atores (apostador e administrador) e as seguintes funcionalidades que cada um pode efetuar:

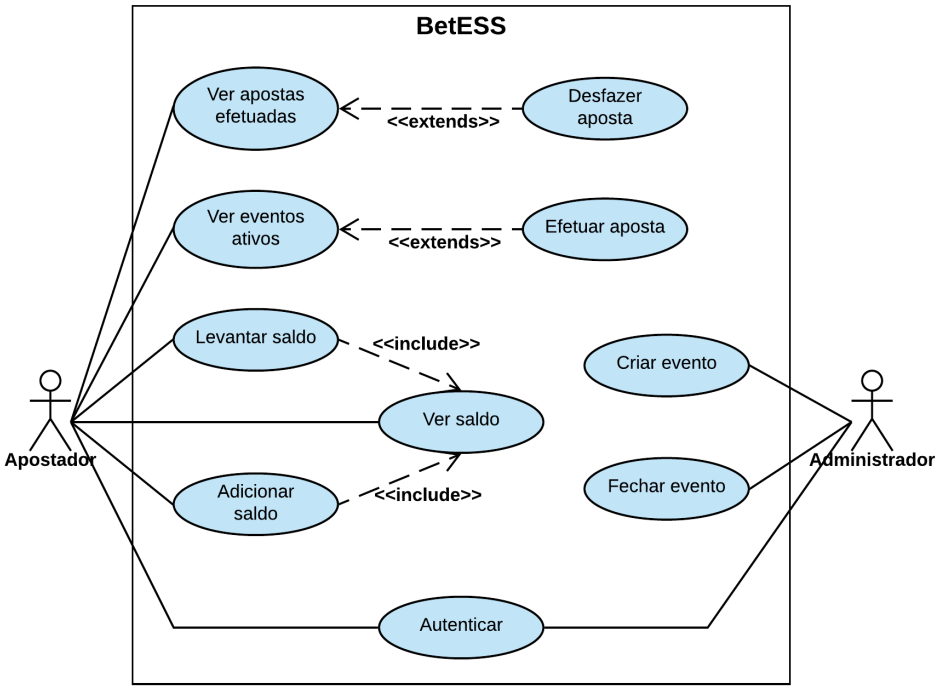

Figura 5: Diagrama de *use case*.

3.1.3 Diagrama de Classe

O diagrama de classe de um sistema é essencial na estruturação de qualquer arquitetura orientada aos objetos. Neste caso, desenvolvemos uma versão sem padrões, pelo que a arquitetura das classes do sistema é simples e com pouca estruturação no que toca à implementação dos métodos necessários para dar resposta aos requisitos levantados.

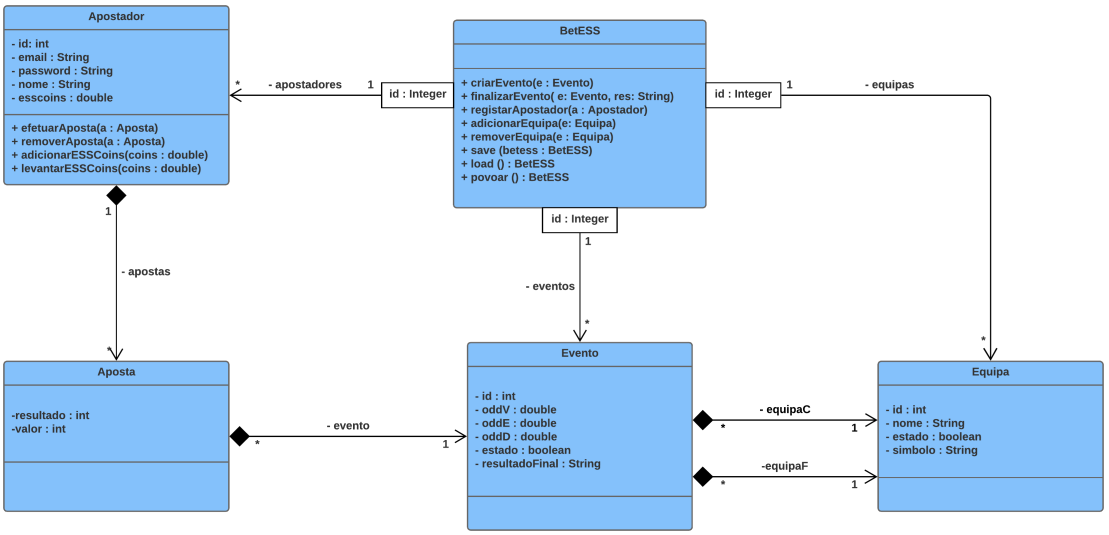

Figura 6: Diagrama de classe.

Esses métodos encontram-se divididos pelas classes onde irão provocar uma alteração de facto, nos seus dados. Temos por exemplo o método `efetuarAposta` na classe `Apostador` que recebe uma instância da classe `Aposta` e a adiciona à lista de apostas desse apostador. Ou na classe `BetESS`, o método `criarEvento` que permite a adição de um novo evento ao `map` de eventos existentes nessa classe.

Estes métodos permitem essencialmente responder corretamente aos requisitos do projeto e são capazes de criar uma versão funcional do *software* pretendido, mas não apresentam uma arquitetura correta ao nível do uso de padrões de *software* existentes que permitam resolver problemas comuns de uma forma mais simples.

3.2 Implementação

A implementação do *software* modelado foi feita em *Java* com recurso à API gráfica *Swing*, desenvolvido com recurso ao IDE *Netbeans*.

A arquitetura da solução baseia-se nas classes descritas no Diagrama de Classe modelado, com adição das classes necessárias para os formulários que compõem a interface gráfica do projeto.

Visto a arquitetura ser muito rústica e sem utilização de padrões de *software*, os métodos que verificam o cumprimento dos requisitos levantados encontram-se nas próprias classes dos formulários, recorrendo depois aos métodos descritos no Diagrama de Classe para efetuar as modificações necessárias à estrutura de dados da aplicação.

Foi também implementada persistência dos dados inseridos e alterados durante a execução do programa através do uso do mecanismo de serialização na classe `BetESS` com os métodos `load` e `save`.

3.3 Resultado final

O resultado final foi uma aplicação funcional, capaz de responder aos requisitos levantados.

Apresentamos a seguir, algumas capturas de ecrã da interface do *software*:

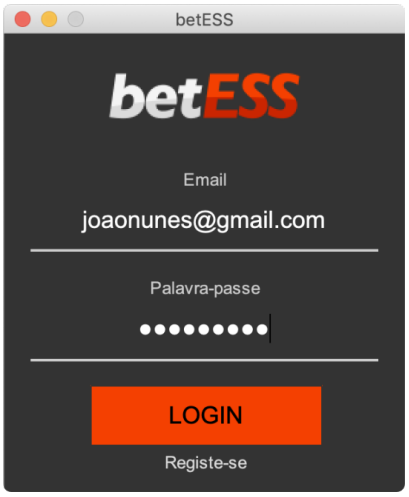

Figura 7: Página de autenticação.

Figura 8: Eventos ativos para apostar.

Figura 9: Área de administrador.

4 Versão com padrões de *software*

4.1 Escolha dos padrões

Os padrões de *design* apresentam uma curva de aprendizagem bastante árdua inicialmente, mas que se revela num investimento vantajoso. Permitem implementar soluções já existentes e testadas no nosso projeto, poupando tempo e esforço. Isto permite com que seja mais fácil entender a solução implementada no projeto, tornando-se também mais fácil de manter.

A implementação de padrões neste projeto tem de ser analisada de acordo com os requisitos de qualidade recolhidos previamente. A sua simples implementação de padrões de *design* por si só já é benéfica, mas é necessário analisar os padrões adequados ao projeto a ser desenvolvido, uma vez que alguns padrões podem prejudicar certos atributos de qualidade específicos (por exemplo, o padrão *Singleton* tem um impacto negativo na expansibilidade do código).

Neste caso em específico, foi-nos pedido para implementar padrões que beneficiassem o **desempenho**, **modificabilidade** e **facilidade de uso** do sistema.

Relativamente à facilidade de uso, este requisito é cumprido tendo em conta o ambiente gráfico agradável atribuído à aplicação, com funcionalidades simples e intuitivas.

Os outros dois requisitos de qualidade foram respeitados recorrendo a padrões de *design*, nomeadamente o *Observer* e a arquitetura *Layered* (por camadas).

4.1.1 Padrão *Observer*

Este padrão deve ser implementado numa arquitetura orientada aos objetos quando um objeto pertencente ao domínio do problema deseja ser notificado sobre a alteração de outro objeto.

Facilmente encontramos um paralelismo com o nosso projeto. De facto, temos um apostador que já recebe notificações sobre o término dos eventos sobre os quais tem apostas efetuadas na versão sem padrões. A aplicação deste padrão poderá ser benéfica no nosso caso, uma vez que permite aplicar uma solução já existente e testada ao nosso problema sem grandes complicações, permitindo reduzir o tempo investido no desenvolvimento do *software* e aumentar a facilidade de compreensão do código.

Para além disso, este padrão consegue ir de encontro a alguns atributos de qualidade estabelecidos pelos seus requisitos, pois este padrão aumenta a reutilização, expansibilidade e simplicidade do código, para além de tornar o sistema mais flexível, na medida em que é fácil adicionar funcionalidades ao adicionar entidades que colaboram com as já existentes. Estas caraterísticas facilitam a modificabilidade do sistema, permitindo cumprir o requisito referente a este atributo de qualidade.

4.1.2 Arquitetura por Camadas

Este padrão arquitetural carateriza-se acima de tudo pela segurança, reutilização e manutenibilidade.

A segurança que a arquitetura por camadas proporciona, deve-se ao facto de os utilizadores terem apenas acesso à camada superior da arquitetura, tipicamente a camada de apresentação. Essa camada não contém nem a informação, nem a lógica do *software*.

A sua reutilização deve-se ao facto de a sua arquitetura por camadas, permitir uma abstração maior dos métodos, sendo mais fácil reutilizar esses métodos noutros projetos.

Por fim, a manutenibilidade deve-se à facilidade de alterar certas porções de código, facilmente sem implicar alterações noutras camadas da aplicação. [2]

Contudo, estas caraterísticas podem custar no desempenho do sistema uma vez que um simples pedido, tem de atravessar todas as camadas. No entanto, no nosso caso, dado o pequeno número de camadas implementado e a inexistência de acessos múltiplos à aplicação, a influência da arquitetura no seu desempenho é mínimo.

4.2 Modelação

Parte da modelação desenvolvida para a versão sem padrões de *design* pode ser usada neste capítulo, designadamente o Modelo de Domínio e o Diagrama de *Use Case*, visto que a arquitetura do sistema em nada afeta quer o problema em causa representado no modelo de domínio, quer as funcionalidades representadas no diagrama de *use case* que este deverá proporcionar aos atores envolvidos.

4.2.1 Diagrama Arquitetural

O padrão arquitetural usado para a nossa solução do problema baseou-se em Camadas (*Layers*). Neste padrão, os componentes do *software* estão organizados em camadas horizontais com tarefas definidas que trocam informações com as camadas adjacentes, mas que podem eventualmente ter ligações a outras camadas também.

A nossa arquitetura vai ser baseada em três camadas bem delineadas:

- **Presentation:** Camada que contém as classes relativas aos formulários *Swing*, onde a informação é disponibilizada para os atores (apostador e administrador) e onde os *inputs* desses atores é recolhido para a execução dos métodos.
- **Business:** Camada responsável pela lógica do programa, onde os métodos responsáveis pela verificação de condições, alteração de variáveis, etc. é efetuado.

- **Persistence:** Camada onde os dados voláteis alterados durante o decorrer do programa são guardados num documento e onde é lido esse documento aquando do início da execução do programa.

Este padrão arquitetural tem diversas vantagens, como já foi explicitado no capítulo anterior onde justificámos o porquê da escolha deste padrão arquitetural. Na imagem abaixo é apresentada a nossa para a arquitetura do *software* a ser desenvolvido:

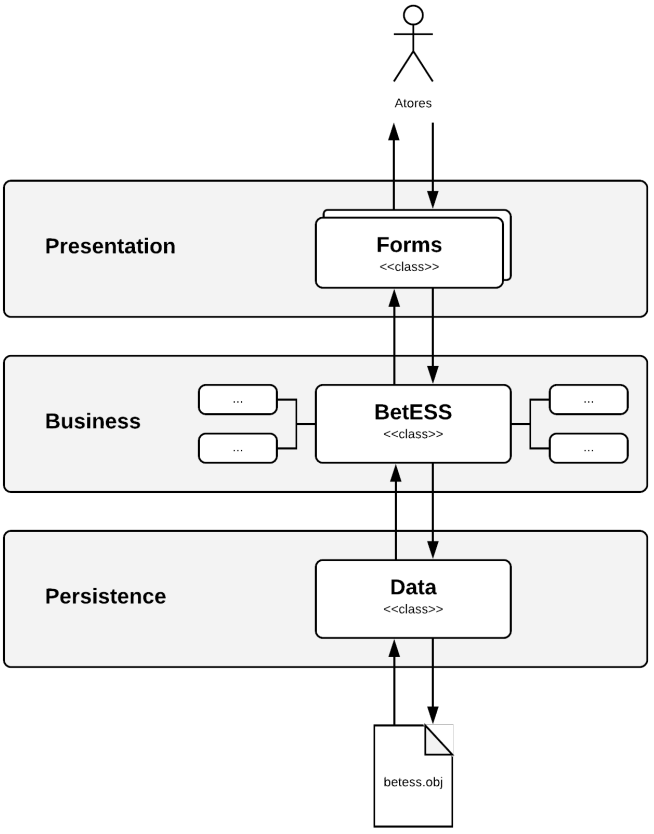

Figura 10: Arquitetura da solução.

4.2.2 Diagrama de Classe

Para a implementação dos padrões de *design*, foi necessário revolucionar por completo a estrutura do sistema no que toca à organização das classes.

Na versão anterior, os métodos responsáveis pelo funcionamento correto do sistema, encontravam-se dispersos nos formulários gráficos onde os botões que acionavam esses métodos se encontravam. Isso vai contra a filosofia da arquitetura por camadas. A função dos formulários é apenas chamar os métodos que se irão encontrar na camada inferior, a camada *Business*, nomeadamente na classe `BetESS`. Esta classe, como se pode ver no diagrama de classe abaixo, é a responsável pela interação entre

as camadas inferior e superior, recebendo todos os métodos invocados pela ação do utilizador nos formulários, tais como `login`, `criarEvento`, `efetuarAposta` e todos os restantes visíveis no diagrama.

Ainda relativamente à implementação da arquitetura por camadas, dada a existência de uma camada de persistência, é necessária a existência de uma classe responsável por receber os pedidos de acesso aos ficheiros de dados por parte da camada de *Business*. Essa classe, denominada `Data` presente na camada *Persistence*, tem métodos responsáveis pela alteração das estruturas de dados às quais está ligada, como os utilizadores, os eventos e as equipas do sistema.

Relativamente à implementação do padrão *Observer*, este foi implementado usando o `Evento` como ***Subject*** e os utilizadores (`User`) interessados (seja ele um *Bookie* ou um Apostador) como ***Observer***. A interface `User` atua então como observador para os eventos, sendo implementada pelas classes `Admin`, `Apostador` e `Bookie`, no entanto, apenas nos interessa notificar as últimas duas. De facto, o evento não atualiza o estado dos utilizadores diretamente, mas notifica-os através do método `notifyUsers` que invoca o método `update` sobre os seus observadores, tornando assim o evento independente de como os utilizadores são notificados. O utilizador só é notificado porém, se for adicionado à lista de observadores de cada evento, utilizando para isso os métodos `addUser` e `removeUser`. Assim que o utilizador recebe uma notificação, cria a `String` de aviso para mostrar na janela e guarda numa lista de notificações, que será "despejada"assim que o utilizador iniciar sessão no sistema.

Tendo concluído o raciocínio para a construção das classes, podemos modelar os conceitos propostos no seguinte diagrama:

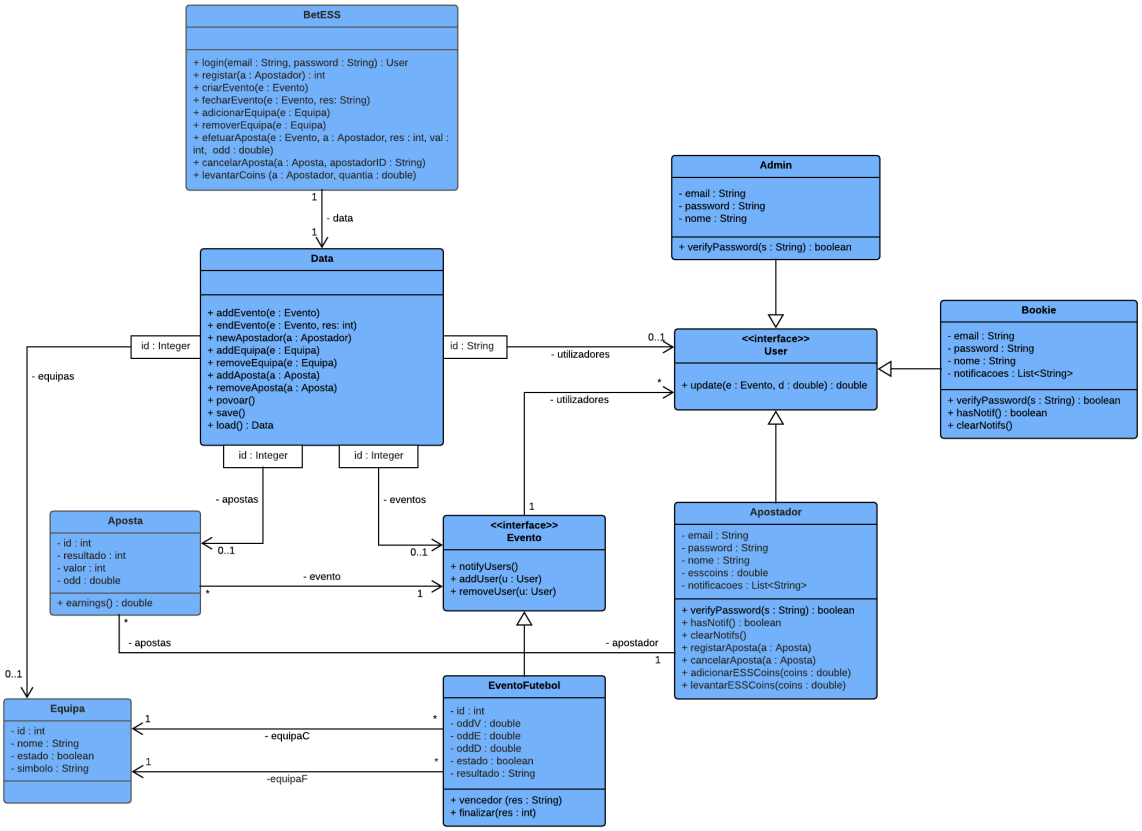

Figura 11: Diagrama de classe.

Este novo diagrama de classes modelado implementa assim todos os padrões escolhidos para responder face aos requisitos de qualidade levantados. Assim, podemos passar à implementação do código modelado, tendo este diagrama como base estrutural, permitindo acelerar o processo do desenvolvimento do *software*.

4.3 Implementação

A implementação do *software* procedeu-se de forma semelhante à versão anterior sem padrões. Usamos o mesmo IDE, com recurso à mesma API gráfica, sendo que a maioria dos formulários existentes foram reutilizados.

As camadas arquiteturais foram divididas em *packages* com o respetivo nome e as classes e interfaces foram implementadas no seu devido *package*. Tal pode ser confirmado na imagem abaixo:

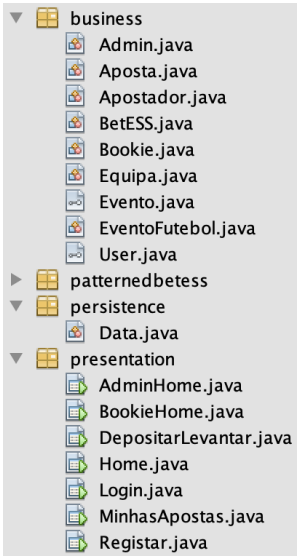

Figura 12: *Packages* e respetivas classes.

Foi também mantida a persistência dos dados implementada na versão anterior.

4.4 Requisito *Bookie*

Perto do final do trabalho prático, foi introduzida uma nova funcionalidade: um novo tipo de utilizador, denominado de *Bookie*. Com este, foi introduzido também um novo conjunto de requisitos funcionais:

- O *Bookie* deverá aceder ao programa apenas após a introdução do respetivo email e password.
- O *Bookie* terá a capacidade de criar novos eventos e definir as suas *odds*.
- O *Bookie* terá a opção de ser ou não notificado do resultado final e lucros gerados por um evento.

Para simplificar a sua implementação, e porque se considerou pouco lógico permitir qualquer utilizador da aplicação registar-se como um *Bookie*, determinou-se que cada *Bookie* tem de ser pré-carregado no ficheiro de persistência através do método `povoarBookies` da classe `Data`.

Para acomodar esta nova funcionalidade, foi necessário implementar uma nova classe responsável por guardar a informação e lógica de negócio para o tipo de utilizador em questão, bem como um menu pelo qual este irá interagir com o sistema.

Por outro lado, foi necessário também ajustar o processo de *login* de modo a que reconheça o utilizador como *Bookie* e o método responsável pela notificação dos resultados dos eventos, de modo a notificar também os *Bookies* interessados.

Resumidamente, a implementação desta nova funcionalidade não se revelou de grande dificuldade uma vez que grande parte do código foi reutilizado. Isto deve-se ao facto de já ter sido recolhido como requisito de qualidade a necessidade a implementação do padrão *Observer* para notificar os apostadores das suas apostas. Assim sendo, o processo de implementação do *Bookie* baseou-se na reutilização do código já existente para o apostador.

4.5 Resultado final

O resultado final revelou-se satisfatório uma vez que foram corretamente aplicados todos os padrões que se revelaram necessários para o cumprimento dos requisitos de qualidade levantados. E a aplicação desses padrões se traduziu num sistema funcional, capaz de responder a todos os requisitos funcionais exigidos. Visualmente, o resultado final é similar ao obtido na versão anterior, com a adição do novo menu para o *Bookie*.

Figura 13: Menu do *Bookie*.

5 Conclusão

Analisando o desenvolvimento deste trabalho de um modo geral, é fácil de entender o porquê da necessidade de implementação de padrões de *design*. Estes são essenciais no processo de desenvolvimento do *software*, visto facilitarem o seu desenvolvimento, mas também aumentarem certas caraterísticas do sistema, tais como a sua reutilização, manutenibilidade, desempenho ou simplicidade.

No nosso caso, os padrões aplicados permitiram ir ao encontro dos requisitos de qualidade levantados. As vantagens do seu uso revelaram-se mesmo durante este projeto, uma vez que o facto de já termos implementado o padrão *Observer* permitiu com que o surgimento de um requisito novo à última da hora fosse implementado sem grande dificuldade, dada a grande capacidade de reutilização fornecida ao código por este padrão.

Comparando diretamente à versão desenvolvida sem padrões aplicados, conseguimos facilmente encontrar as diferenças. A versão inicial é bastante desorganizada, de fácil acesso aos dados do sistema, com pouca capacidade de reutilização de código e de difícil manutenção e compreensibilidade. A versão com padrões, por outro lado, é o oposto: a sua arquitetura é bem estruturada em camadas, o código tem uma capacidade de reutilização maior (tal como foi verificado neste mesmo projeto) e de maior capacidade de compreensão e manutenção.

Concluindo, a implementação da plataforma *BetESS* revelou-se bastante produtiva. Desta forma obteve-se uma base sólida no que toca ao levantamento de requisitos funcionais e de qualidade, que ser irão traduzir em padrões. A aprendizagem obtida na implementação destes padrões também foi essencial para aumentar a capacidade de extensão, reutilização e refinamento do *software* desenvolvido.

Referências

[1] Foutse Khomh, Yann-Gaël Guéhéneuc. *An Empirical Study of Design Patterns and Software Quality*. University of Montreal, 2008.

[2] Nassrin Eftekhari, Morteza Poyan Rad, Hamid Alinejad-Rokny. *Evaluation and Classifying Software Architecture Styles Due to Quality Attributes*. Islamic Azad University, 2011.

[3] André L. Ferreira, João Luís Sobral. *Slides de Arquiteturas de Software*. Universidade do Minho, 2018.