Instituto Superior de Engenharia de Lisboa

Engenharia de Software

Relatório

Projeto – Shopping List

Mestrado em Engenharia Informática e Multimédia

Rodrigo Dias, 45881

Semestre de Inverno, 2021/2022

Índice

Lista de figuras

1 Introdução

Este projeto tem como objetivo aprender e aperfeiçoar técnicas de desenvolvimento de software de maneira a implementá-las tendo em conta boas práticas, de maneira a manter o projeto bem organizado e estruturado, permitindo melhor colaboração e melhor implementação de forma a evitar criar complexidade, pois isso leva o programador a cometer mais erros e a muito tempo desperdiçado. Também permite uma maior facilidade de adaptação de serviços ou adição de funcionalidades no futuro, bem como a manutenção do sistema.

2 Análise de requisitos

2.1 Visão

Esta primeira parte de desenho do projeto tem como objetivo organizar os passos necessários para concretizar as necessidades e características que o projeto deve ter, fazendo uma análise do domínio do problema e criando o documento de visão, que descreve o problema e a solução de forma muito geral. Este documento foi feito explicando as ações que os atores têm sobre o sistema a desenvolver, distinguindo as permissões de cada um e quais as características que o sistema terá, como o ambiente de execução e o comportamento resultante da interação entre os atores e a interface. Este documento está em anexo em formato PDF (Documento de visão.pdf), explicando os aspetos principais da aplicação.

2.2 Especificação de requisitos

O segundo passo é então a especificação dos requisitos que o sistema irá ter de acordo com o ator e o comportamento que deve ter ao reagir às suas interações. É também documentado as necessidades não funcionais, isto é, as restrições e atributos do sistema, e um glossário para todos os que forem realizar este projeto saibam os termos e o contexto do que se está a trabalhar. Este documento também está em anexo em formato PDF (Especificação de requisitos.pdf).

2.2.1 Modelo de casos de utilização

Nesta secção representam-se os atores e as interações que cada um pode realizar sobre o sistema, dando informação resumida para cada caso de utilização, as pré-condições necessárias e os vários cenários que podem decorrer. A ideia é fazer um esboço destas operações e ir aprofundando e pondo em evidência os casos que são comuns a outros. A ideia é detetar onde são necessários os casos de inclusão, de extensão ou de generalização para compactar estes esboços e tornar mais simples a leitura e interpretação dos requisitos.

Os casos de generalização são feitos quando a operação entre dois casos de utilização é a mesma na sua essência, mudando apenas a ação específica realizada para cada caso.

Os de inclusão são quando vários casos de utilização utilizam as mesmas operações, podendo estas ser um caso de utilização separado que é incluído pelos outros.

Os de extensão servem para complementar um caso de utilização já completo, acrescentando funcionalidades, daí este tipo ser opcional.

2.2.2 Especificação suplementar

É aqui que são colocados os requisitos não funcionais como as restrições de ao introduzir parâmetros, os campos não podem ficar vazios ou o tempo que os utilizadores têm para poder ver certos produtos ou listas apagadas é limitado. São requisitos que não influenciam o comportamento do sistema, isto é, não são essenciais estes requisitos para que o sistema funcione.

2.2.3 Glossário

Aqui colocam-se os termos que possam ser utilizados na conceção do projeto para que haja uma linguagem comum entre os colaboradores e os clientes de maneira a evitar qualquer tipo de erros de interpretação.

3 Projeto de arquitetura da aplicação

3.1 Arquitetura lógica

Nesta fase trata-se de aperfeiçoar os casos de utilização especificando a interação entre partes de como funcionará o sistema, ainda sem depender da plataforma de desenvolvimento.

3.1.1 Modelo de domínio

O modelo de domínio tem como objetivo estruturar as entidades do domínio do problema, percebendo quais as que são mais relevantes e a relação entre elas. A Figura 1 mostra o modelo de domínio deste projeto. A partir do documento de visão, fez-se uma análise por conceitos chave que sejam relevantes e retiraram-se as entidades de grupo, lista de compras, produto e assinatura. Ora o grupo possui informação do seu id, nome e dos vários utilizadores que lhe pertencem. A lista de compras possui um id, o seu nome e uma assinatura de quando foi apagada, se alguma vez for. Esta assinatura possui informação de quem fez essa operação e de quando isso aconteceu. O produto tem mais detalhes para além do id e do nome como a marca, a loja onde comprar, detalhes extra, quantidade, uma etiqueta de comprar em desconto ou urgente e assinaturas de quando foi adicionado à lista, quando foi removido e quando foi comprar e por quem.

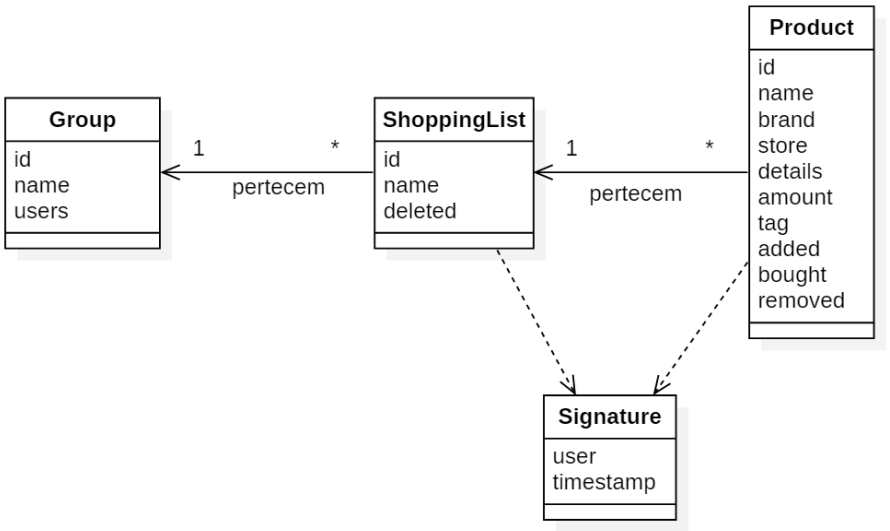

Figura 1 - Modelo de domínio

3.1.2 Realização de casos de utilização

Para a realização de casos de utilização utilizou-se o padrão do tipo *Model-View-ViewModel* (MVVM) pois abstrai muitos componentes e processos, tornando mais fácil e prático o desenvolvimento das interações entre as várias camadas.

Nas figuras seguintes, da realização e dois casos de utilização, observam-se as camadas de repositórios e serviços, que fazem parte do *Model* do padrão. As camadas de vistas, as interfaces com que o ator interage, fazem parte da *View* do padrão. As camadas de vistaModelo são as camadas que tratam de ligar as interações do ator na interface com a camada do modelo e estas correspondem ao *ViewModel* do padrão.

- **Comprar produtos**

Este caso de utilização começa após o ator entrar na janela dos produtos e decide comprar alguns produtos. Assim que entra, a vistaModeloProdutos ativa a *stream* dos produtos da lista de compras em que está, através do servicoProdutos que lida com essa lógica, acedendo ao repositório dos produtos. Esta *stream* mantém a informação atualizada, ou seja, caso algum produto seja adicionado à lista, a *stream* é disparada com a informação atualizada.

Depois o ator pressiona o botão para comprar produtos e transita para a janela de compra. Esta janela de compra apresenta os produtos que se podem comprar, de acordo com a restrição na especificação suplementar. Por isso a vistaModeloCompra chama um método do serviço de produtos para obter esses produtos e o serviço de produtos terá a informação de todos os produtos atualizada devido à *stream* ativada na janela anterior por isso, o método *buyingProducts* acede a essa lista e filtra segundo as restrições necessárias.

Uma vez na janela com os produtos possíveis para comprar, o ator só tem de ativar ou desativar um campo de *checkbox* para indicar que comprou ou cancela a compra do produto específico.

Ao terminar, o ator prime o botão para voltar atrás e regressa à janela dos produtos. Estes passos estão ilustrados no diagrama de interação da Figura 2.

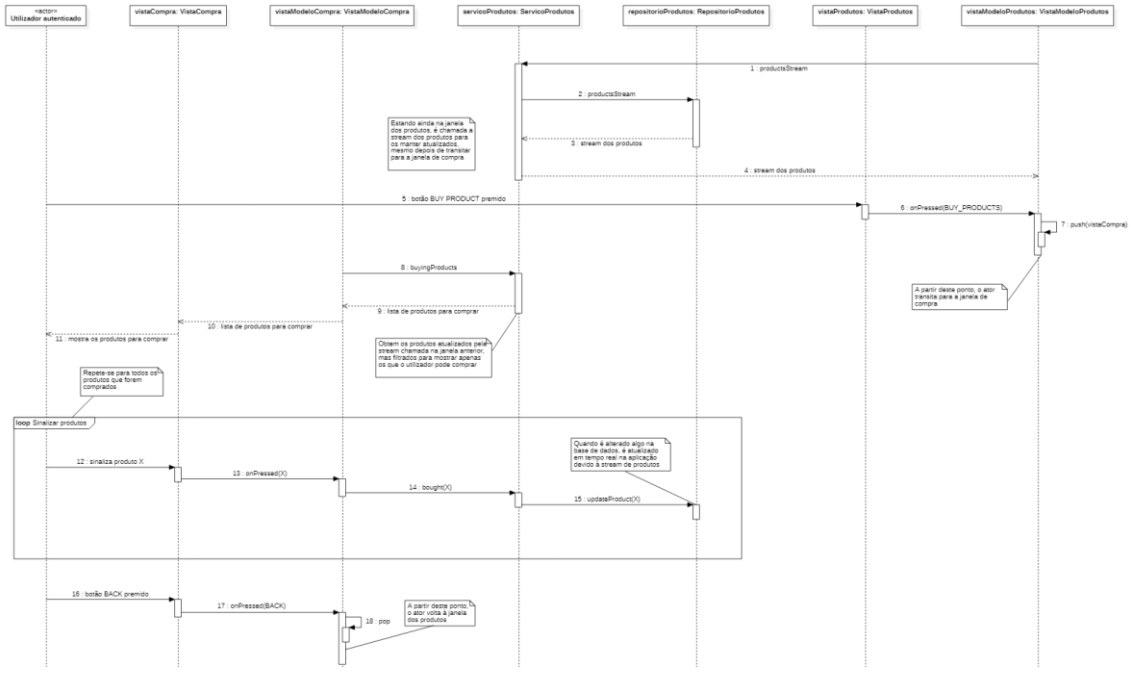

Figura 2 - Diagrama de interação para o caso de utilização comprar produtos

- **Criar grupo**

Este caso de utilização começa quando o ator está na janela dos grupos a que pertence, após se autenticar ou entrar como convidado. O ator começa por premir o botão para criar um grupo, interagindo com a vistaGrupos. Ao clicar, é aberto um painel *popup* e as interações nesse painel são tratadas também na vistaModeloGrupos pois continua-se na mesma janela.

Nesse painel aparece um campo para ser introduzido o nome do grupo que se vai criar. Após o introduzir e confirmar, é feita uma verificação para ficar de acordo com o requisito da especificação suplementar (nome não pode ficar vazio) e caso seja inválido, aparece uma mensagem de aviso a informar o ator, podendo corrigir o erro.

Quando é introduzido um nome válido, é evocado o método, na camada de serviços dos grupos, para criar o grupo, que por sua vez chama o método para adicionar o grupo ao repositório.

As interações com o repositório de dados são síncronas (em todos os casos), apesar de poderem ser assíncronas, porque só após alterar a base de dados é que se continua, pois o que é apresentado no ecrã está atualizado em tempo real com o uso de *snapshots* (ver glossário).

Depois de esperar que a afetação na base de dados termine, o painel é fechado automaticamente, podendo o ator interagir com a janela dos grupos.

O diagrama de interações deste caso está apresentado na Figura 3.

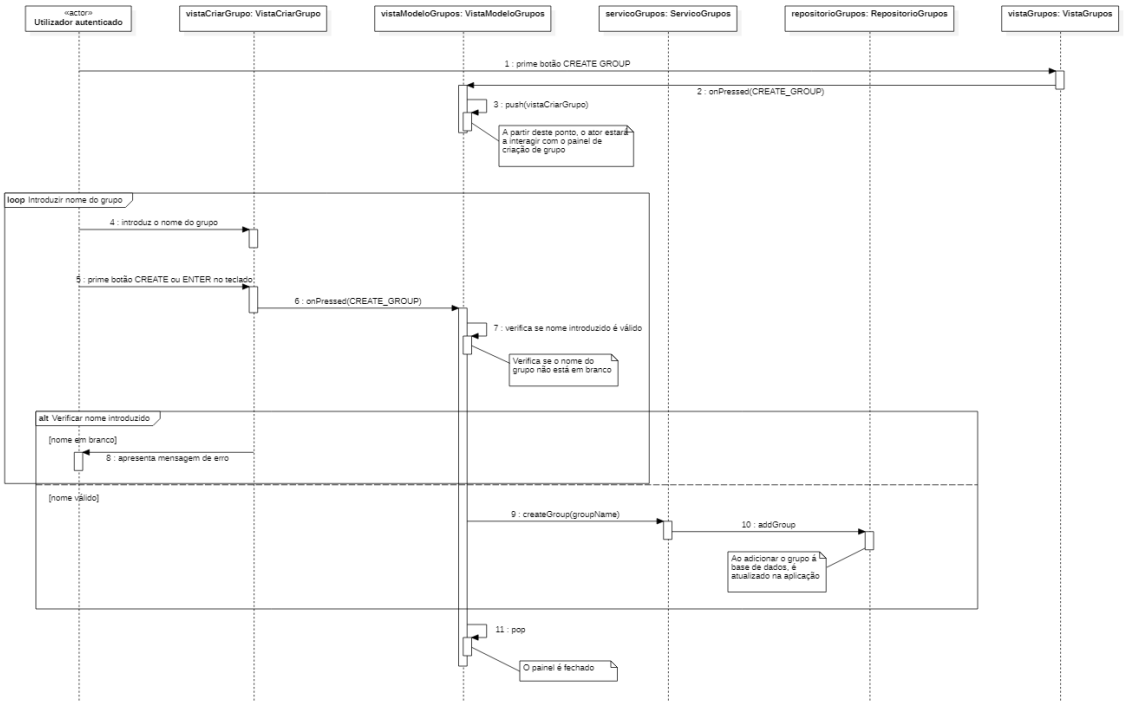

Figura 3 - Diagrama de interação para o caso de utilização criar grupo

3.1.3 Arquitetura de mecanismos

Esta fase serve para obter um esboço dos objetos e entidades que são usados de acordo com o modelo de interação feito no ponto anterior e dos mecanismos entre todas as camadas. O documento da arquitetura está em anexo no formato PDF (Arquitetura.pdf).

- **Comprar produtos**

Para este caso de utilização generalizou-se as vistasModelo pois todos os *viewModels* tem os mesmos métodos de operação sobre as vistas que lhes estiverem associadas. Cada vista tem acesso apenas às entidades e aos serviços, neste caso, apenas dos produtos. A Figura 4 ilustra o diagrama de classes do mecanismo deste caso de utilização.

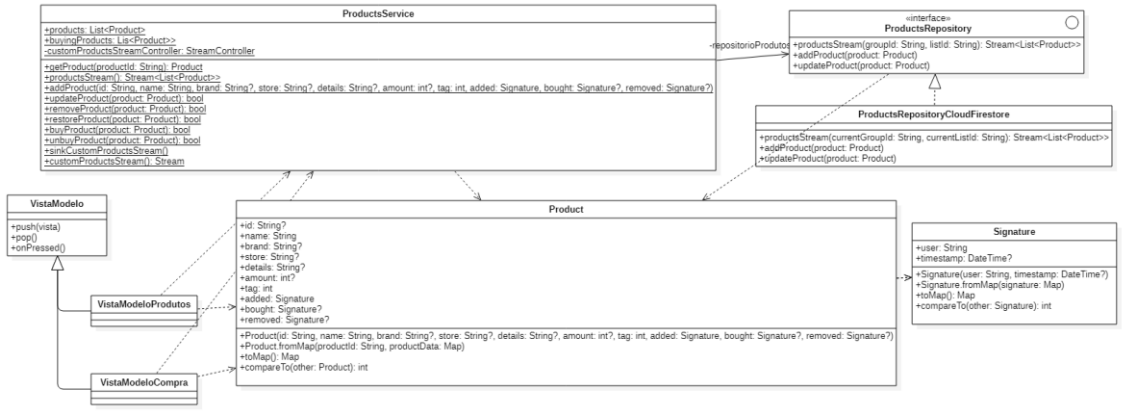

Figura 4 - Arquitetura de mecanismos para o caso de utilização comprar produtos

- **Criar grupo**

Fez-se a mesma coisa para este caso de utilização e o mecanismo deste caso de utilização está apresentado na Figura 5.

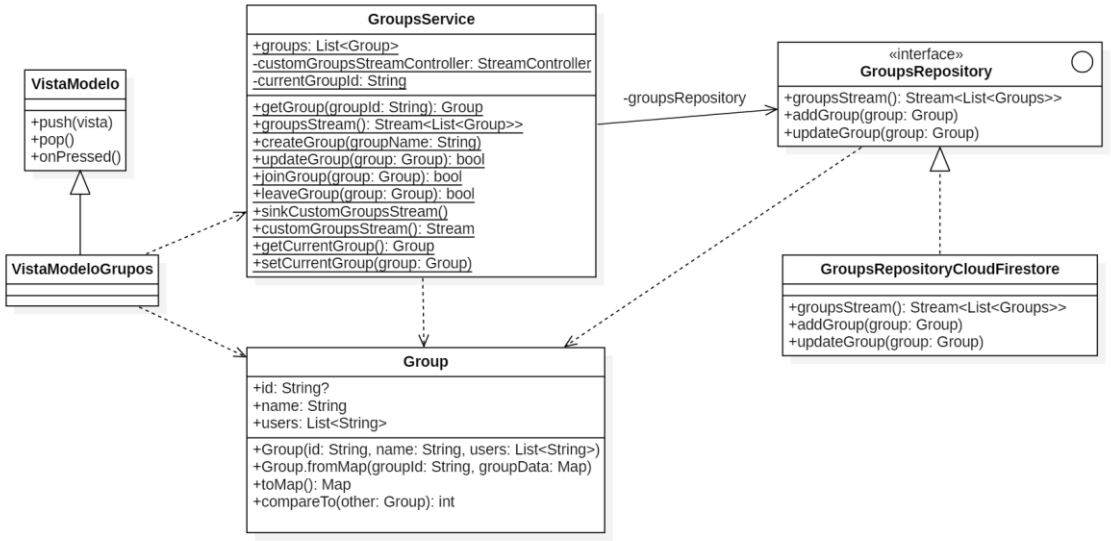

Figura 5 - Arquitetura de mecanismos para o caso de utilização criar grupo

Decidiu-se colocar os serviços como classes estáticas pois estas classes são acedidas por diferentes *viewModels* que pretendem obter por exemplo os grupos a que o utilizador pertence e assim elimina-se ter de criar uma instância e passar entre todas as camadas necessárias.

3.1.4 Arquitetura geral da solução

Neste capítulo organiza-se a arquitetura geral da solução, separando todos os componentes em 3 camadas principais, sendo estas a camada de apresentação, de domínio e de acesso a dados, como mostra a Figura 6.

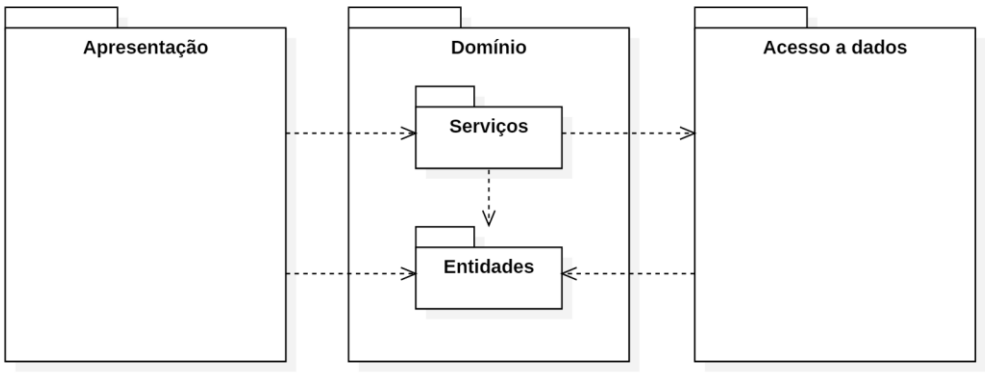

Figura 6 - Arquitetura de subsistemas

A ideia de fazer esta divisória em 3 camadas serve para tornar independentes os módulos entre elas, isto é, caso se pretenda alterar a base de dados utilizada, é possível fazê-lo sem alterar nada na camada de apresentação nem na de domínio. Caso se queira alterar a interface também é possível fazê-lo, só tendo de evocar os mesmos métodos da camada de domínio.

Isto facilita o desenvolvimento do código pois está mais organizado e caso se pretenda fazer mudanças futuras como adicionar funcionalidades, é muito mais fácil adaptar. Outra das razões para implementar algo com este formato é para aumentar a coesão, isto é, agrupar um conjunto de funções que trabalham sobre algo semelhante, como o acesso a base de dados, tornando o código coeso e não há tanto acoplamento, ou seja, uma certa entidade não está dependente de muitas outras. O problema seria que, ao alterá-la, as que dependiam também tinha de ser modificadas de acordo, aumentado a complexidade e a possibilidade de cometer erros.

É na camada de domínio que fica a lógica da aplicação e são os serviços que manipulam os dados obtidos da base de dados de maneira a apresentá-los na camada de apresentação.

As figuras seguintes mostram as interligações entre as classes consoante a camada a que pertencem. Estas interligações são as mesmas da arquitetura de mecanismos.

- **Comprar produtos**

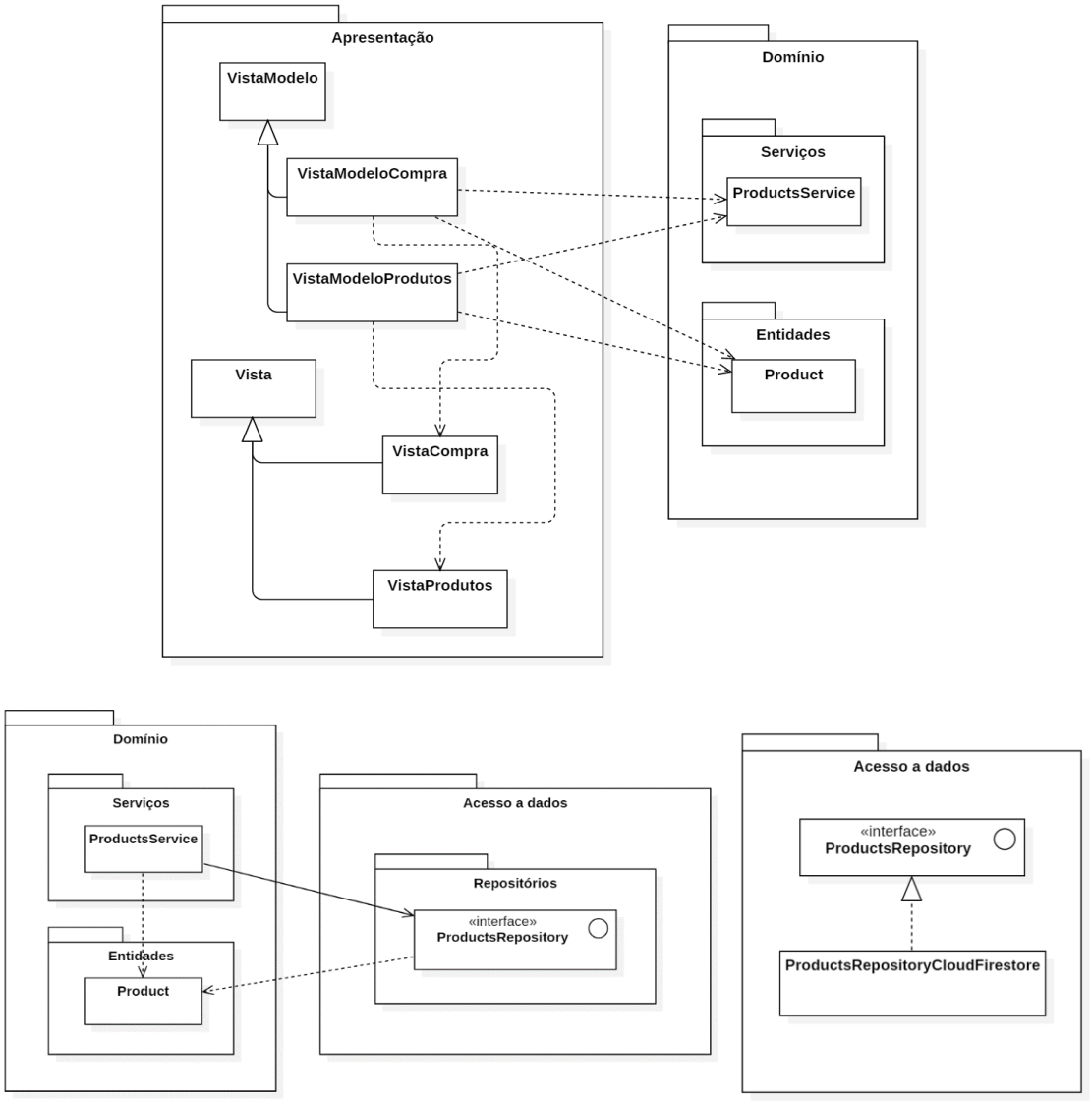

Figura 7 - Arquitetura de subsistemas para o caso de utilização comprar produtos

- **Criar grupo**

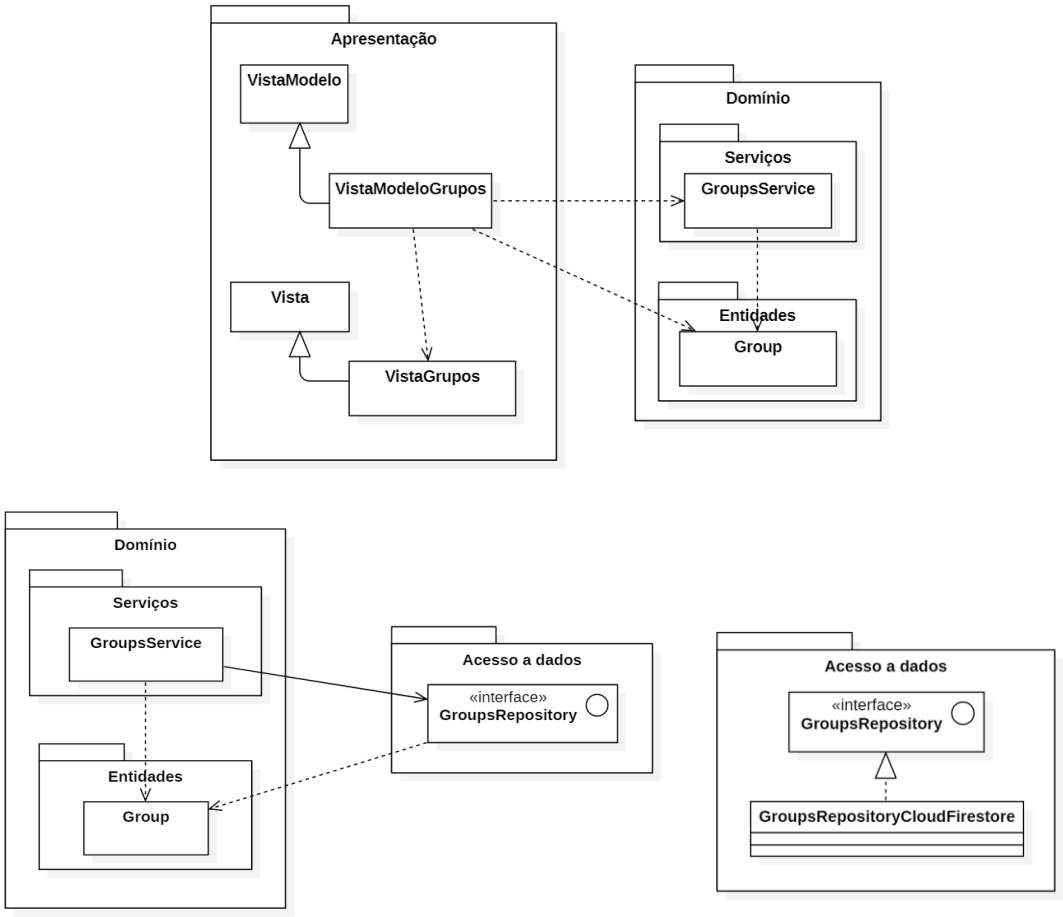

Figura 8 - Aruitetura de subsistemas para o caso de utilização criar grupo

3.2 Arquitetura detalhada

Nesta fase especifica-se quais as plataformas a serem usadas bem como os dispositivos físicos em ter em conta, planeando com mais detalhe como será feita a implementação e as dependências entre partes.

A plataforma que se decidiu usar foi o *Flutter*, que utiliza a linguagem de programação *Dart*. Também se escolheu usar os serviços de base de dados *Firestore* e autenticação da *Google*.

3.2.1 Modelo de dinâmica

O modelo de dinâmica é usado para definir o comportamento de um objeto e as suas restrições. São mostrados 3 modelos de dinâmica, um para a dinâmica dos produtos, outro para a dinâmica das listas de compras e outro para a dinâmica dos utilizadores, nas figuras seguintes.

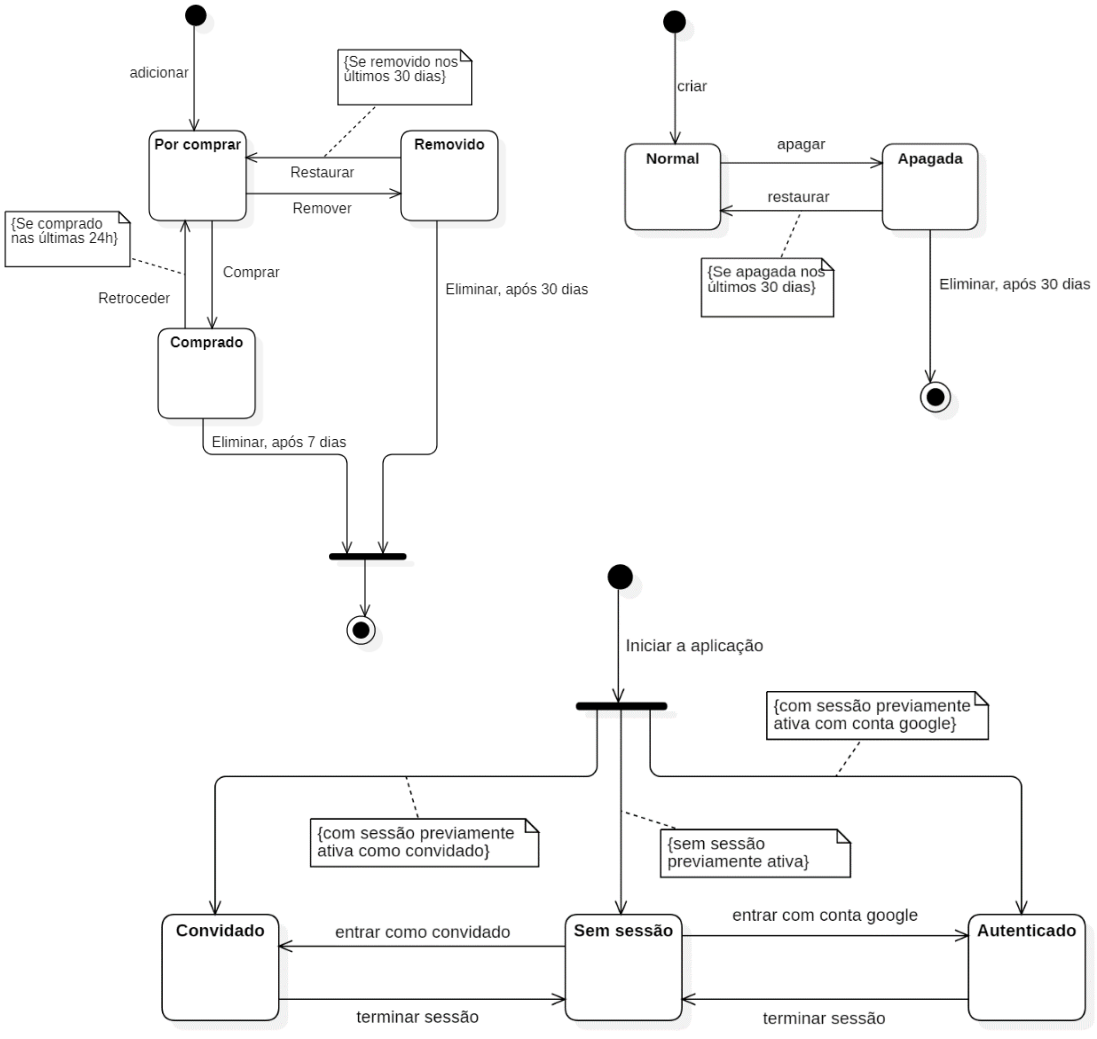

Figura 9 - Modelos de dinâmica

3.2.2 Detalhe de partes e mecanismos

A Figura 10 mostra a relação entre todas as entidades deste projeto com os métodos que cada um deve ter.

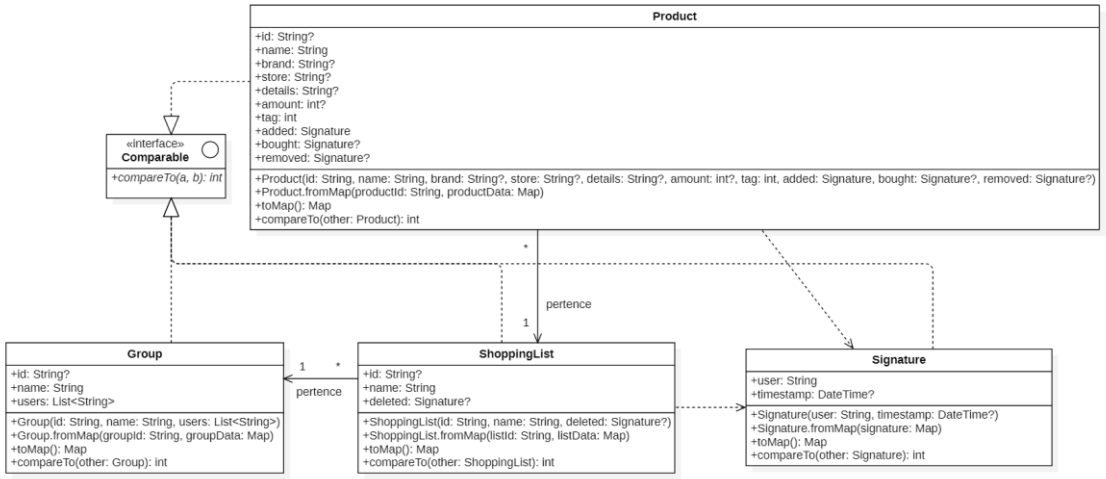

Figura 10 - Diagrama de classes das entidades

- **Comprar produtos**

As partes para implementar este caso de utilização estão ilustradas na Figura 11. Haverá uma função *main* que irá lidar com as janelas apresentadas ao ator. Felizmente o *Flutter* abstrai o processo para instanciar as vistas. É feita uma classe dedicada para o repositório de acesso a dados da *Firestore*, implementando os métodos de maneira a retornar de acordo com a interface que implementa.

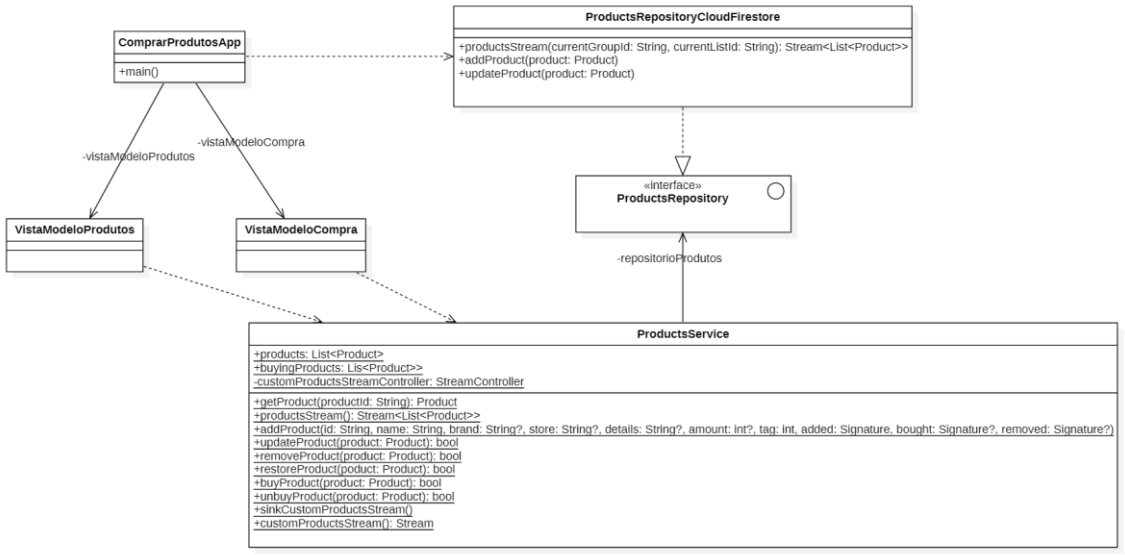

Figura 11 - Detalhe de partes e mecanismos para o caso de utilização comprar produtos

- **Criar grupo**

Na Figura 12 observa-se o esboço semelhante ao anterior, apenas adaptado a este caso de utilização.

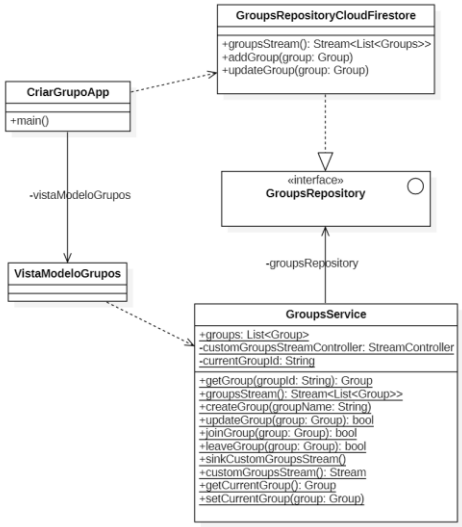

Figura 12 - Detalhe de partes e mecanismos para o caso de utilização criar grupo

3.2.3 Arquitetura de teste

Na arquitetura de teste indicam-se as camadas e os mecanismos para implementar o protótipo de teste. A ideia é conseguir implementar um protótipo de teste que não dependa de questões da plataforma como a interface, ou de serviços externos como o acesso à base de dados, com o objetivo de verificar a camada de domínio.

Uma vez que o teste esteja concluído, esta camada de domínio fica inalterada quando for feito o protótipo aplicacional. Desta maneira, a probabilidade de erros diminui devido a resolver uma camada de cada vez, estando a de domínio já completa e funcional.

- **Comprar produtos**

Criou-se uma classe de base de dados para simular o acesso a dados, tendo os dados localmente e implementando os métodos retornando o tipo de dados de acordo com a interface, como mostra a Figura 13.

Também se observa a relação entre estes componentes a partir de uma função *main* que evocará os métodos para realizar o teste à camada de domínio, apresentado os resultados obtidos.

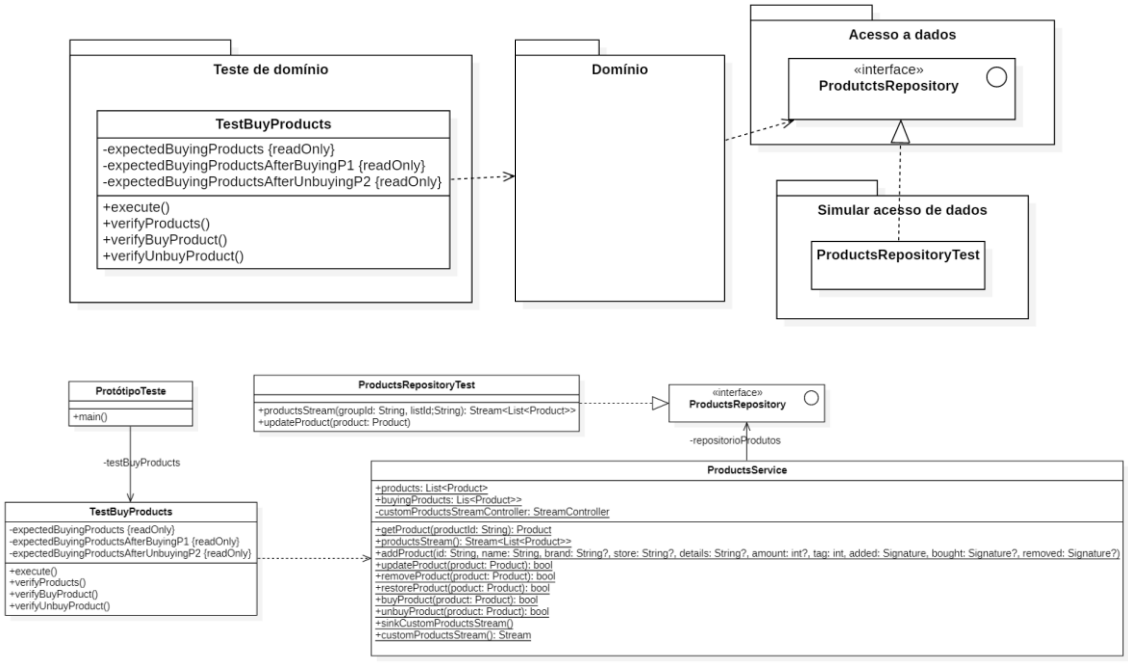

Figura 13 - Arquitetura de teste para o caso de utilização comprar produtos

- **Criar grupo**

As figuras seguintes mostram o mesmo processo para o caso de utilização de criação de grupo.

Neste caso a classe *TestCreateGroup* depende do próprio repositório de teste apenas porque se quer observar quais os grupos já na base de dados para verificar se a filtragem (obter apenas grupos a que o utilizador pertence) está correta. No caso anterior não era necessário porque todos os produtos eram obtidos e a filtragem era feita já tem a informação de todos.

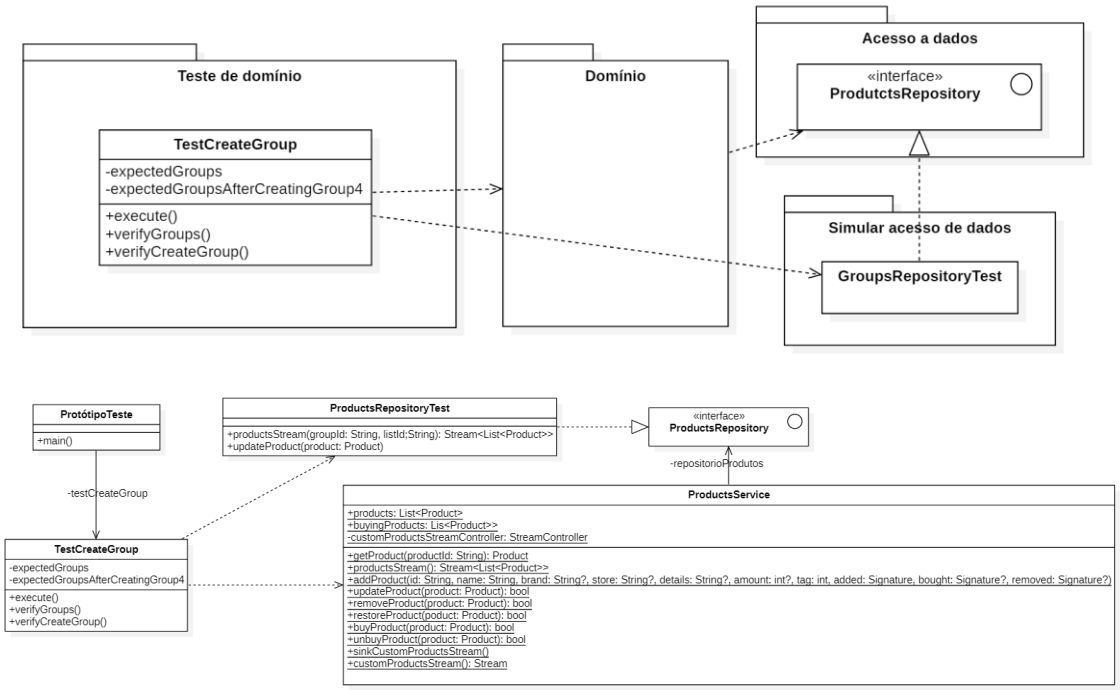

Figura 14 - Arquitetura de teste para o caso de utilização criar grupo

3.2.4 Modelo de implantação da aplicação

O modelo de implantação serve para indicar os componentes físicos que são necessários para permitir implementar o sistema. Na Figura 15 vê-se que é necessário um dispositivo *smartphone* com sistema operativo *Android* para correr a aplicação. É também preciso uma ligação à internet para poder comunicar com os servidos de base de dados e autenticação da *Google*, estando estes num servidor, sobre o serviço da *Firebase*.

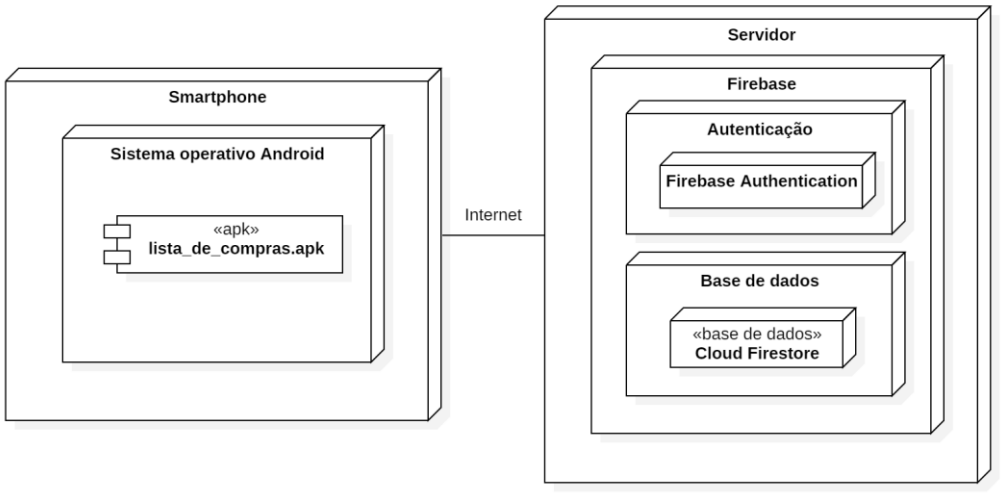

Figura 15 - Modelo de implantação

4 Implementação do protótipo de teste

A implementação do protótipo teste pretende testar a camada de domínio sem qualquer dependência da interface e bibliotecas externas usando apenas a linguagem de programação em modo consola.

Para tal, simula-se uma base de dados e para os 2 casos de utilização apresentam-se os dados esperados ao executar um determinado caso de utilização e os resultados realmente obtidos após executar.

Para este projeto, no caso de utilização de comprar produtos, existe dependência do tempo a que foram adicionados e comprados os produtos então assume-se manualmente um valor em que os produtos foram adicionados e um tempo atual. Isto serve para ver que os produtos comprados à mais de 24h não devem ser apresentados ao utilizador. Também se introduzem manualmente utilizadores que fizeram as operações para confirmar que um produto comprado por outra pessoa não aparece.

Assumiu-se que o ID do utilizador é ‘u1’ e todas as operações que necessitam do ID usam este valor. Pode-se observar quando são apresentados os dados atuais na base de dados e aqueles que é suposto o utilizador ver bem como os grupos existentes e quais é que devem ser apresentados ao utilizador (que devem ser apenas aqueles a que o ‘u1’ pertence).

O código foi feito em linguagem *Dart* e juntaram-se as classes de teste dos dois casos de utilização no mesmo código. Realizou-se um *script* (teste.bat), na pasta ‘prototipo de teste’, para executar o código de maneira a facilitar a apresentação dos resultados para ambos os casos.

5 Implementação do protótipo aplicacional

Nesta fase implementaram-se 3 casos de utilização, incluindo os dois falados anteriormente mais o caso de ver listas, que transita para a janela das listas de compras associadas ao grupo escolhido, usando as plataformas aplicacionais mencionadas. Utilizou-se o *Flutter* e os serviços de base de dados *Firestore* da *Google*.

Escolheu-se estas plataformas pois já se tinha conhecimento do uso tanto do *Flutter* como dos de serviços da *Firebase*. Decidiu-se usar pois também facilita a implementação para várias outras plataformas, pois o *Flutter* permite multiplataforma. A escolha do uso dos serviços da *Google* foi também porque a compatibilidade é ótima pois o *Flutter* também é feito pela empresa. Tinha-se conhecimento de programar aplicações em *Android Studio*, mas quando se decidiu experimentar *Flutter* viu-se que era muito mais fácil implementar muitas das coisas que por outro lado no *Android Studio* iria ser bastante mais complexo.

Ao transitar para implementar o protótipo aplicacional, utilizaram-se os mesmos ficheiros da camada de domínio sem qualquer alteração pois estes foram feitos independentemente da plataforma e serviços externos, bastando fazer a interface e o repositório de acesso a dados.

Utilizaram-se as bibliotecas necessárias para usar os serviços de *Firebase* e outras extra para facilitar com certos elementos da interface.

De maneira a facilitar, realizaram-se os 3 casos de utilização na mesma aplicação, podendo selecionar qual verificar na janela inicial, como mostra a Figura 16.

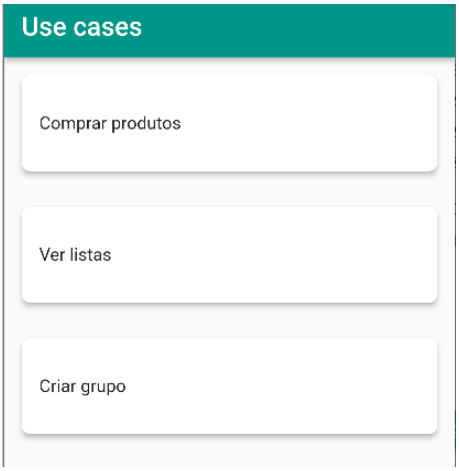

Figura 16 - Interface do protótipo aplicacional na página inicial

- **Criar grupo**

Ao escolher o caso de utilização para criar grupo, observa-se que o utilizador só vê os grupos 1, 2 e 3, mas na base de dados existe mais um com nome 4. No entanto, esse só pertence ao utilizador 'u2', logo não é apresentado ao utilizador teste 'u1', como mostra a Figura 17.

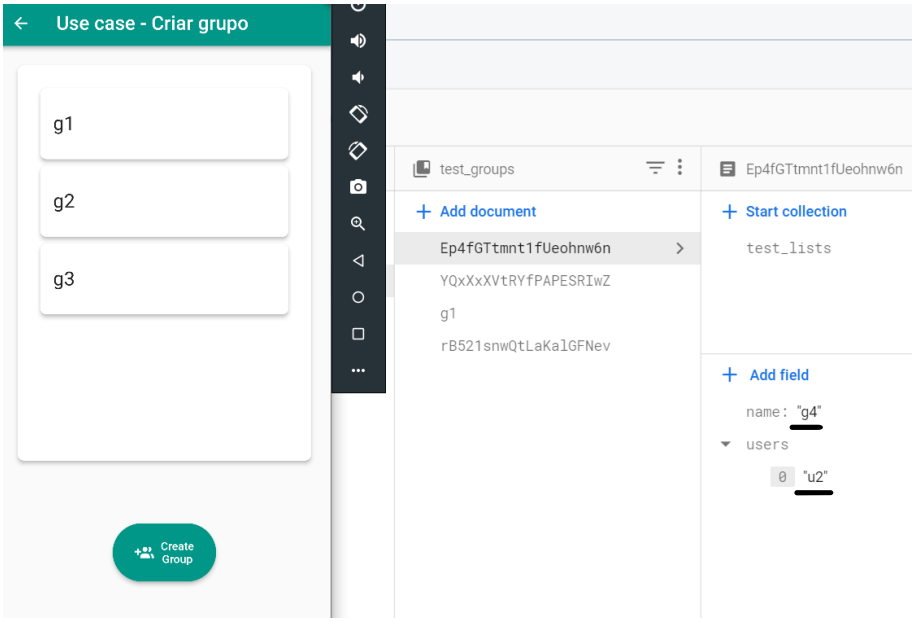

Figura 17 - Dados apresentados na interface do utilizador consoante os dados na base de dados para o caso de utilização criar grupo

- **Ver listas**

Ao criar um grupo neste caso de utilização, esse irá aparecer no caso de utilização de ver listas, começando sem nenhuma lista de compras. A interface deste caso tem o aspeto mostrado na Figura 18, em que à esquerda estão os grupos a que o utilizador pertence e à direita estão as listas que pertencem ao grupo que foi escolhido.

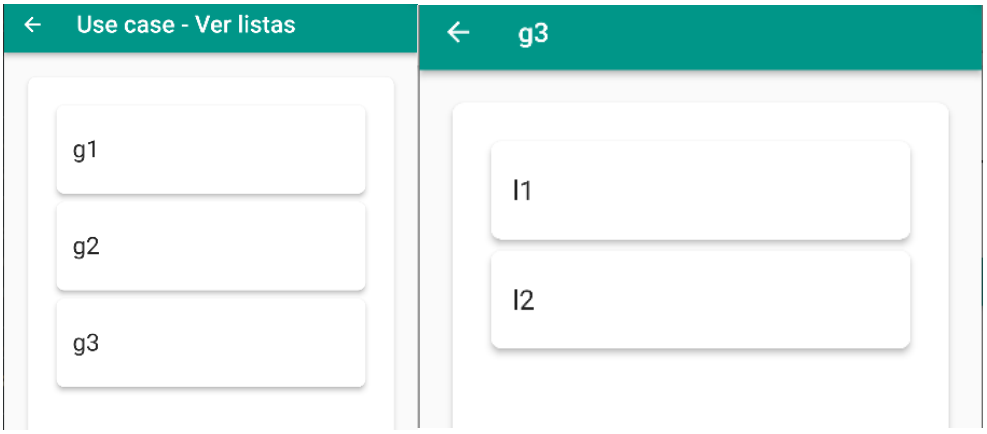

Figura 18 - Interface para o caso de utilização ver listas

- **Comprar produtos**

Na compra de produtos é possível observar na Figura 19 que o produto foi comprado há mais de 24h horas logo não aparece na lista que o utilizador vê. (Data do teste: 5 de fevereiro; Data a que o produto 'p3' foi comprado: 3 de fevereiro)

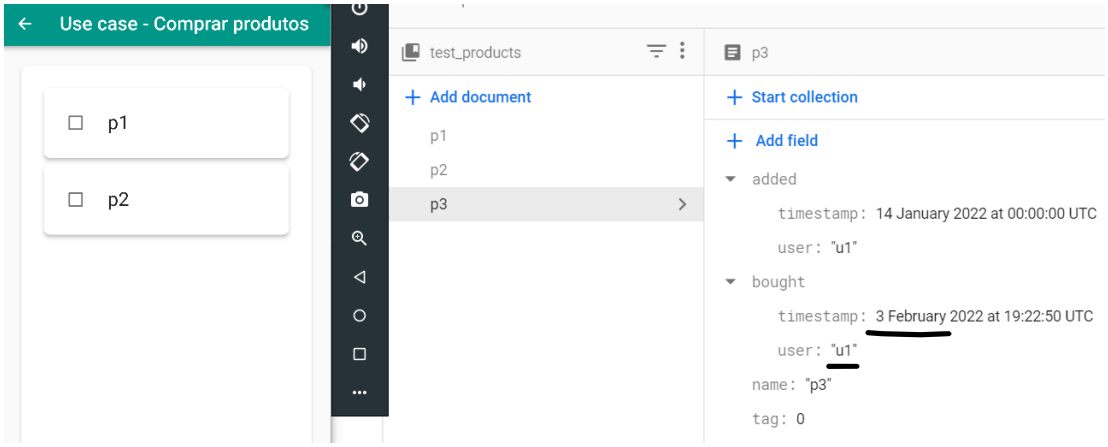

Figura 19 - Dados apresentados ao utilizador consoante os dados na base de dados para o caso de utilização comprar produtos

Como se separaram os casos de utilização, foi criada a interface de escolha e, após escolher o caso de utilização, o utilizador é redirecionado para a janela correspondente devido ao troço de código apresentado na Figura 20.

```
routes: {
  // Comprar produtos
  'ComprarProdutos_Products': (context) => ComprarProdutos_Products(),
  'ComprarProdutos_Buying': (context) => ComprarProdutos_Buying(),
  // Criar grupo
  'CriarGrupo_Groups': (context) => CriarGrupo_Groups(),
  // Ver listas
  'VerListas_Groups': (context) => VerListas_Groups(),
  'VerListas_Lists': (context) => VerListas_Lists(),
},
```

Figura 20 - Troço de código da organização das várias janelas

Este parâmetro de *routes* atribui uma *string* para facilitar a transição entre as várias vistas, sendo apenas necessário indicar o nome correspondente. Cada uma das classes associadas a essas *strings* tem a sua lógica para apresentar a sua interface, a sua janela. Estas classes são denominadas de *widgets* que no fundo são classes que estendem *widgets* com os métodos necessários para construir aquilo que aparece na interface.

Para correr a aplicação aconselha-se a usar o ficheiro *apk* em anexo ('prototipo aplicacional.apk', situado na pasta 'prototipo aplicacional') e instalar num dispositivo *Android* ou num emulador. Caso se queira correr o código pelo ficheiro *main*¸ o código estará em modo *debug* e serão necessárias ter algumas coisas em conta. É necessário ligar um dispositivo *Android* ou ligar um emulador e instalar o SDK do *Flutter*. Aconselha-se o uso do *VScode* pois instala também a linguagem *Dart* automaticamente. Caso se pretenda usar um emulador, será preciso instalar o *Android Studio* para criar um.

6 Análise crítica do projeto realizado

Neste projeto, para completar toda a aplicação, bastava seguir os mesmos métodos para cada caso de utilização, passando só no final para o protótipo teste para definir as classes e entidades na camada de domínio passando só depois para a implementação do protótipo aplicacional. Desta maneira foi bastante mais fácil a implementação dos casos pois tratou-se de uma forma independente a camada de domínio, onde se situa a parte mais complexa da aplicação, podendo fazer o desenho da interface sem preocupar com a lógica de domínio. Assim evitaram-se vários erros, tornando a implementação do código muito mais eficiente, já que sem esta preparação e organização, não seria possível implementar a aplicação em relativamente pouco tempo, passando mais tempo a procurar e corrigir erros que a implementar código.

Também era possível ter definido com mais detalhe os métodos e interações necessárias com a linguagem UML pois algumas ferramentas permitem gerar o código quase totalmente, poupando ainda mais tempo a programar, evitando assim mais erros, e tendo logo tudo estruturado para mais facilmente colaborar com outras pessoas e permitindo que elas percebam o que se está a implementar e o comportamento que o sistema terá. Algo que facilitaria também era ter um esboço da interface a acompanhar os requisitos, dessa maneira, quando se fosse fazer a interface já se teria uma ideia feita, preocupando apenas em desenhá-la de acordo. No entanto, neste caso também existem ferramentas que, de acordo com o esboço da interface e as operações sobre ela, permitem a geração de código, diminuindo a probabilidade de cometer erros e dessa maneira poupando o tempo desperdiçado a resolvê-los.

A grande vantagem é que se for necessário alterar a base de dados, basta criar uma classe dedicada para o uso dessa nova base de dados com os métodos de maneira a obter os dados no formato esperado na camada de domínio. É exatamente este nível de organização que torna fácil a programação e manutenção de um sistema de software.

7 Conclusão

Com o desenvolvimento deste projeto, concluiu-se a importância da preparação e organização dos passos para a resolução da solução que se procura para resolver o problema apresentado. Entendeu-se bem a importância de diminuir a complexidade do sistema prevenindo erros e tempo perdido a resolvê-los, apontando sempre para uma coesão alta (mais não em excesso) e um acoplamento baixo entre as partes que constituem o sistema.

Também se entendeu o quão poderosa é a linguagem UML e as suas capacidades, pois no fundo também uma linguagem de programação, mas de muito mais alto nível podendo escalar para qualquer tipo de sistema, mantendo tudo organizado e de muito mais fácil leitura.

Percebeu-se também a importância de ir fazendo revisões ao código que se vai fazendo pois permite evitar erros que de outra maneira dificilmente seriam detetados. No fundo, é este o maior problema de programar um sistema de software, é a deteção de erros e o tempo perdido que vem associado a resolvê-los. As abordagens e conceitos aprendidos nesta UC serão certamente utilizados em qualquer projeto de software futuro.