INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores

Believe

FILIPE JOSÉ ALVES MENDES

Licenciado em Engenharia Informática e de Computadores

Trabalho de Projecto para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Luis Filipe Graça Morgado

Júri:

Presidente: Pedro Alexandre Seia Cunha Ribeiro Pereira

Vogal: Rui Manuel Feliciano de Jesus

Novembro, 2017

Aos meus pais por toda a educação que me deram e aos meus irmãos por serem a referência que são.

Agradecimentos

Manifesto a minha gratidão ao Doutor Luis Morgado, orientador deste projecto de mestrado, pela sua simpatia e disponibilidade desde o nosso primeiro encontro, pelas críticas e conselhos, mas sobretudo, pela forma como me fez olhar para o software de uma forma simples, clara, equilibrada e objectiva nas disciplinas que leccionou.

Quero também agradecer a todos os amigos que fiz, que conheci em vários momentos da minha vida, das mais variadas áreas profissionais, e que me tornam melhor a cada dia que passa.

Gostaria de dar uma palavra de apreço também a uma antiga professora que tive, Graça Coelho, por palavras de confiança que me prestou e que ainda hoje tenho gravado em mim.

Por fim agradeço profundamente à minha família por todo o apoio que me deram desde o início e por nunca colocarem barreiras em quaisquer que fossem as minhas decisões académicas e profissionais.

A todos, um muito obrigado.

Resumo

Os videojogos, para além de serem uma forma de entretenimento, são também uma ferramenta de aprendizagem [36]. Apesar de toda a polémica de violência associada, quando reduzidos à sua forma mais básica, os videojogos são capazes de ensinar a explorar, a entender relações espaciais, a melhorar a precisão, etc.

Com o avanço da tecnologia, novos dispositivos foram surgindo e a possibilidade de jogar em qualquer lugar tornou-se algo normal no dia-a-dia. Primeiro com as consolas portáteis e mais recentemente com os smartphones e tablets, a emergência dos videojogos móveis, permitiu a criação de novas formas jogar, como a realidade virtual e a realidade aumentada, mas poucos são os jogos que tiram partido do contexto do utilizador para determinar o decurso de um jogo.

Nessa perspectiva, Believe é um videojogo móvel que tira partido de componentes contextuais do utilizador ao longo do tempo, baseado em eventos (*puzzles*, chamadas simuladas, pesquisa de informações, etc.) que necessitam de interacção do utilizador, desencadeados por ações tomadas por este (ou não).

Os eventos são temporalmente curtos e preferencialmente lançados em situações menos oportunas, de forma a causar *suspense* e ansiedade no utilizador.

Palavras-chave: Videojogo móvel, contexto, localização, evento, android, inteligência artificial

Abstract

Videogames, besides being an entertainment form, are also a learning tool [36]. Even though they are sometimes associated with violence, when reduced to their most basic form, videogames are able to teach on how to explore, on understanding spacial relations, to improve precision, etc.

With the improvements in technology, new devices have appeared and the possibility to play anywhere (or any place) has become something we take for granted in our daily lives. First with portable gaming consoles and more recently with smartphones and tablets, the emergence of mobile videogames, allowed the creation of new ways to play, like virtual or augmented reality, but very few games take advantage of the user context to determine a game's course.

Believe is a mobile videogame that takes advantage of contextual information from the user throughout time, based on events (puzzles, fake calls, quizzes, etc.) that need the user interaction and triggered by actions taken by it (or not).

These events will be short and preferably triggered in inopportune situations, in order to cause suspense and anxienty on the user.

Keywords: Mobile videogame, context, location, event, android, artificial intelligence

Índice

Lista de Figuras **xv**

Lista de Tabelas **xvii**

Lista de Listagens **xix**

1 Introdução **1**

1.1 Definição do Problema 2

1.2 Motivação 3

1.3 Objectivos 3

1.4 Estrutura do Documento 4

1.5 Convenções 4

2 Enquadramento **5**

2.1 Videojogos Móveis 5

2.2 Plataforma Android 6

2.3 Estilos de jogo 8

2.4 Casos de estudo 9

3 Requisitos da Solução **11**

3.1 Visão 11

3.2 Análise de Requisitos 14

4 Solução Proposta **19**

4.1 Mecânica do Jogo 19

4.2 Estados do Jogo 24

4.3 Arquitectura 25

4.4 Interface e Experiência de utilização 30

5 Implementação **35**

5.1 Enquadramento da plataforma Android 35

5.2 Estatísticas e Distribuição 37

5.3 Estrutura e Fluxo de dados 39

5.4 Aplicação 41

5.5 Lógica de Negócio 50

5.6 Repositório 59

5.7 Dependências do Projecto 62

6 Teste e Qualidade de Código **65**

6.1 Testes 65

6.2 Qualidade de Código 69

6.3 Variantes da Aplicação 70

7 Processo de Desenvolvimento **73**

7.1 Processo Ágil 73

7.2 Prototipagem 74

7.3 Ferramentas de Suporte 74

8 Conclusões **79**

8.1 Videojogo 80

8.2 Tecnologias Utilizadas 80

8.3 Desenvolvimentos Futuros 81

Referências **83**

A Divulgação e Promoção **i**

A.1 Twitter i

A.2 Medium ii

A.3 Publicação e Promoção ii

Lista de Figuras

2.1 A plataforma Android 7

3.1 Demonstração de um desafio 12
3.2 Casos de Utilização 17

4.1 Relação entre tolerância e frequência de desafios 20
4.2 Fluxo 1 21
4.3 Fluxo 2 21
4.4 Exemplos de desafios 23
4.5 Mecanismo agendamento de um desafio 23
4.6 Mecanismo de agendamento de uma consequência 24
4.7 Exemplos de consequências 24
4.8 Modelo de Domínio 25
4.9 Diagrama de Estados do Jogo 25
4.10 Arquitectura Geral 26
4.11 Detalhe geral da arquitectura 28
4.12 Detalhe da camada Repositório 29
4.13 Detalhe da camada Apresentação 29
4.14 Fluxo de Navegação da Aplicação 31
4.15 Wireframe do Jornal 32
4.16 Wireframe do Bloco de Notas 32

4.17 Wireframe do fragmento Contactos 33

5.1 Adopção das versões do SO (referente à semana 08/08/17) 38
5.2 Inicialização de uma aplicação [9] 41
5.3 Diagrama de dependências de StartGameActivity 47
5.4 Classes do ecrã principal 48
5.5 Intervenientes no fragmento DashboardNewsFragment 49
5.6 Intervenientes na lógica das Notícias 50
5.7 Diagrama de classes de alguns Desafios 51
5.8 Diagrama de classes de algumas Consequências 52
5.9 Esquema de utilização de Android-Job 54
5.10 Integração de Jobs e Challenges 55
5.11 Diagrama de Sequência de um Job e Challenge 56
5.12 Diagrama de JobsModule 57
5.13 Diagrama de sequência de um pedido de Notícias 59
5.14 Diagrama simplificado do Padrão DAO 61

6.1 Pirâmide de testes por Martin Fowler [21] 66
6.2 Exemplo de utilização do FindBugs 69

7.1 Excerto da organização feita no Trello 75
7.2 Resultado de duas builds no CircleCi 78

Lista de Tabelas

4.1 Percentagem mínima de desafios completados com sucesso para completar o capítulo com sucesso 20

4.2 Frequência de eventos 21

5.1 Dependências utilizadas 63

Lista de Listagens

5.1 Troço de código em RxJava 41
5.2 Exemplo de injecção de uma Activity 44
5.3 Definição de AppModule 44
5.4 Definição de AppComponent 45
5.5 Definição de ActivityBuilderModule 45
5.6 Definição de BelieveApp com Dagger 46
5.7 Exemplo de injecção numa Activity 46
5.8 Método GetNews() de NewsPresenter 49
5.9 Verificação do estado metereológico 53
5.10 Definição do endpoint "/articles" em Java 60
5.11 Invocação do pedido "/articles" em Java 60
5.12 Definição de um DataAccessObject 62
5.13 Invocação da operação no Repositório 62
6.1 Teste verificação de comportamento com Mockito 67
6.2 Teste de início de activity com Espresso 68

1

Introdução

"Tanto os videojogos como os jogos tradicionais, são difíceis de estudar por serem tão multidimensionais. Há tantas formas diferentes de os abordar. O design e a produção de jogos envolve aspectos de psicologia cognitiva, ciência da computação, design ambiental, e narrativa só para mencionar alguns. Para realmente compreender o que são os jogos, é necessário analisá-los de todos estes diferentes pontos de vista." - Will Write.

Com os primeiros jogos de vídeo comercializados na década de 70 e 80, os videojogos tornaram-se uma forma popular de entretenimento e a sua constante disseminação fez com que fizesse parte da cultura em muitas partes do mundo. Desde *arcades*, até às consolas ou mesmo computadores, os locais e dispositivos para jogar cresceram e evoluíram exponencialmente durante as últimas décadas.

Mais recentemente, com a introdução no mercado de *smartphones* e da sua rápida adopção, tornou-se imperativo para as produtoras e editoras de videojogos a criação de novas formas de jogar, ou até mesmo a adaptação de formas antigas, ao novo estilo de vida dos seus utilizadores. Para além de que, a indústria conseguiria assim "obter" novos utilizadores que nunca jogaram em dispositivos estacionários ou dedicados, visto que, os seus *smartphones* seriam também as suas consolas.

Com mais de 5 milhões de aplicações móveis disponíveis nas lojas de aplicações, a oportunidade de sucesso de uma aplicação é extremamente reduzida, e mesmo

quando se torna relevante, a sua importância vai decrescendo ao longo das semanas. Assim, apenas uma quantidade ínfima de aplicações se consegue manter relevante e essencial para os utilizadores, o que implica a criação de novas formas de os cativar e de os reter, como a utilização contextual do utilizador e a personalização de certos componentes de uma aplicação.

1.1 Definição do Problema

Para além de uma experiência tipicamente mais simples, os videojogos em dispositivos móveis com mais sucesso são de rápida iniciação, (não necessariamente) mais curtos e com um sistema de publicidade e/ou de compras dentro do jogo (*in-app purchases*) como forma de monetização.

Contudo, com o intuito de obter o maior lucro possível, através da distribuição dos jogos no maior número de plataformas possível, as produtoras acabam por não tirar partido de todas as funcionalidades e características (e muitas vezes específicas) disponíveis em cada uma das plataformas, limitando a experiência dos utilizadores.

No caso da plataforma Android, é possível obter dados contextuais, como a actividade e a localização actual do utilizador, sem a necessidade de interacção do mesmo, característica que pode ser aproveitada para determinar o curso de uma história, sem necessitar da interacção contínua como a grande maioria, senão mesmo todos, os jogos que existem hoje em dia.

Não obstante, um dos maiores desafios seria o balanceamento e a utilização correcta do contexto do utilizador. Uma imprópria ou má utilização contextual poderá assustar o utilizador, ao ponto de o fazer desinstalar a aplicação por temer que esta observe e capture demasiada informação pessoal.

Deverá ser tomado como bom exemplo de boa utilização contextual, a aplicação Google Now, que tira partido das acções, localizações e hábitos do utilizador automaticamente para personalizar a experiência que o mesmo tem, sem nunca mostrar tudo o que esta já analisou.

Para além disso, seria necessário balancear o jogo de forma a que os desafios e as consequências sejam lançadas em momentos oportunos para evitar frustração no utilizador e evitar a desinstalação da aplicação.

1.2 Motivação

Na década de 90 e no inicio dos anos 2000, alguns videojogos continham pequenas segredos que existiam para além da experiência gráfica nas consolas. Um desses exemplos, é o primeiro título da saga God of War que continha uma linha telefónica real onde era possível ligar e ouvir uma mensagem de voz [1] do seu criador, David Jaffe. Esta transposição do lado virtual para o mundo real, torna a experiência mais cativante, surpreendente e única.

Numa outra vertente, com a evolução dos dispositivos e do sistema operativo Android, novas funcionalidades são constantemente disponibilizadas, no entanto, este aspecto representa uma dificuldade para os programadores em conseguir acompanhar todas essas novidades. Assim, surgiu a ideia de criar um jogo que utilize estas novidades tecnológicas (não necessariamente gráficas), dando-lhes um propósito com impacto directo ou indirecto no jogo, com alguns elementos do mundo real à mistura.

1.3 Objectivos

O objectivo deste projecto é o desenvolvimento de um videojogo contextual móvel, exclusivo para a plataforma android, que consiga tirar partido das suas características para a criação de uma experiência móvel mencionada na problemática indicada anteriormente.

[1] Apesar de esta linha telefónica já não existir, é possível ouvir a chamada em questão em www.youtube.com/watch?v=0MMzF6cjK9o

1.4 Estrutura do Documento

O presente projecto está estruturado nos seguintes capítulos:

- O capítulo 1 apresenta a identificação do problema, a motivação e os objectivos do presente projecto;

- O capítulo 2 faz um enquadramento tecnológico dos videojogos e do sistema operativo Android, prosseguindo com uma breve análise aos estilos de jogo e a um conjunto de casos de estudo na área dos videojogos móveis com as características que se pretende para o presente projecto;

- O capítulo 3 descreve uma visão geral do jogo e as ideias que surgiram para a sua concepção, prosseguindo com uma análise mais detalhada dos requisitos e funcionalidades que o jogo disponibilizará;

- Já no capítulo 4, é apresentada a dinâmica e a mecânica que o jogo deve oferecer e uma solução arquitectural robusta o suficiente para as realizar;

- Seguidamente em 5, é detalhada a implementação em linguagem java com as bibliotecas e ferramentas necessárias para atingir o objectivo proposto e uma pequena demonstração gráfica do seu resultado;

- No capítulo 6 constará uma pequena descrição dos testes realizados para assegurar a qualidade do projecto e no capítulo 7 qual o processo de desenvolvimento e ferramentas utilizadas para manter o projecto organizado;

Por fim no capítulo 8, é feita uma análise crítica do trabalho desenvolvido e da solução proposta, assim como direcções futuras de investigação.

1.5 Convenções

O presente documento não foi escrito ao abrigo do novo Acordo Ortográfico. Todo o código apresentado foi escrito em Java, salvo indicação contrária.

Enquadramento

Esta secção faz uma introdução mais detalhada aos videojogos móveis, à arquitectura do sistema operativo Android, ao género de jogo escolhido para o projecto e ao estilo utilizado para determinar o seu rumo. Serão analisados alguns dos jogos já disponíveis no mercado com características semelhantes, comparando-os com aquilo que Believe pretende ser, desde a ocorrência aleatória de desafios à utilização do mundo real como característica fundamental para o decorrer do jogo.

2.1 Videojogos Móveis

De modo geral, o termo "videojogo móvel" refere-se aos videojogos jogados em *smartphones* e *tablets* [42] - apesar de uma grande quantidade de jogos serem jogados em pequenas consolas portáteis, como a Nintendo 3DS, Sony PS Vita, entre outros. Os videojogos móveis são também uma parte de um mundo maior que é a área dos jogos electrónicos e do entretenimento móvel.

A adaptação de uma variante do jogo Tetris para o dispositivo Hagenuk MT-2000 em 1994, é considerado como o primeiro videojogo disponibilizado num telemóvel. Anos mais tarde, Snake ganhou uma popularidade à escala global devido a terminais populares como o modelo Nokia 3310 que se tornou um dos dispositivos mais vendidos em todo o globo no ano 2000.

Um ano mais tarde, Botfighters é lançado e é considerado como o primeiro videojogo móvel baseado na localização dos seus utilizadores, onde estes teriam de dividir a sua atenção entre a informação digital presente no jogo e a informação física do mundo real. Porque o mundo real também fazia parte do jogo, o utilizador era forçado a mover-se, para encontrar outros lugares e jogadores a enfrentar. Estas características, fizeram com que Sotamaa argumentasse [48] que Botfighters ilustrava as possibilidades de um verdadeiro jogo móvel, por tirar partido do ambiente físico e do espaço urbano.

No entanto, apenas com o lançamento da App Store do sistema iOS em 2008, e seguido pelo Android Market do sistema Android mais tarde, os videojogos móveis começaram a ganhar relevância no mercado, chegando ao final de 2016 a representar 42% (46.1 mil milhões) dos lucros gerados na industria dos videojogos, sendo previsto que supere a metade do valor total já em 2020 [23].

Com um domínio quase absoluto do mercado, Android (81.7%) e iOS(17.9%) foram responsáveis por 99.6% de todas as vendas de *smartphones* no último trimestre de 2016, deixando à margem, outras plataformas como o Windows Phone e o Blackberry.

2.2 Plataforma Android

A um nível técnico, o Android é um sistema operativo *open source* baseado em linux desenvolvido para ser executado num conjunto grande de dispositivos e de tamanhos diferentes [7]. Como é possível observar pela figura 2.1, o Android é composto pelos seguintes componentes:

- Linux Kernel - onde estão implementados os *drivers* relacionados com o hardware do dispositivo.
- Camada de abstracção de hardware (HAL) - disponibiliza múltiplos módulos com bibliotecas que expõem as capacidades do hardware numa API java de mais alto nível, como os módulos de camera ou *bluetooth*. Quando uma API da plataforma faz uma invocação sobre um componente em específico, o sistema android carrega a biblioteca respectiva.
- Ambiente de execução (*Runtime*) - A partir da versão 5.0 do android, cada aplicação corre no seu único processo com a sua própria instância do ambiente de execução ART. O ART foi desenvolvido para ser corrido em dispositivos com pouca memória como grande parte dos dispositivos Android.

- Bibliotecas nativas C/C++ - Muitos componentes e serviços centrais do android, como o ART e o HAL, foram desenvolvidos com código nativo que necessitam de bibliotecas escritas em C e C++.

- Java API Framework - Todas as funcionalidades fornecidas pelo sistema operativo Android são acessíveis por uma plataforma em Java. Estas API's são a base para a criação de aplicações porque incluem o sistema de *views* para o desenvolvimento da *user interface*, as notificações, um gestor de recursos, etc. A plataforma, expõe também algumas funcionalidades das bibliotecas nativas mencionadas anteriormente para as aplicações.

- Aplicações de Sistema - O Android vem já integrado com um conjunto de aplicações, como o *email*, *browser*, contactos, etc. e será nesta camada que as aplicações para o utilizador são desenvolvidas.

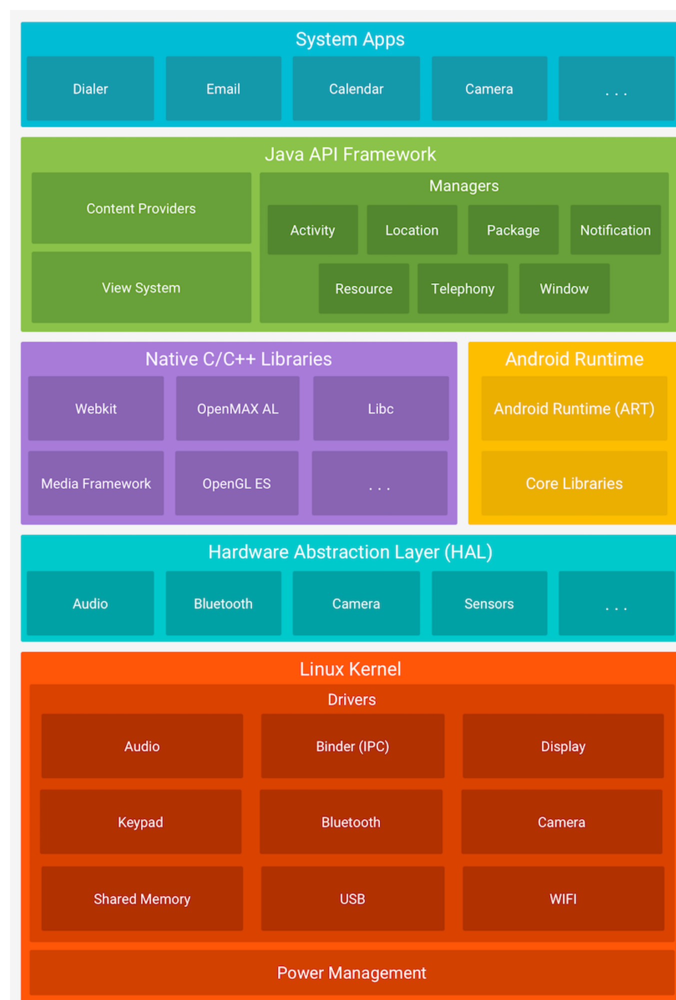

Figura 2.1: A plataforma Android

Para além do desenvolvimento de aplicações nativas para android, existem também um conjunto de motores de desenvolvimento específicos de videojogos muito relevantes no mercado actualmente que têm já características como módulos de física e modelos visuais já desenvolvidos como o GameMaker [52], Unity [51], HaxeFlixel [35] e Gideros [24].

Sendo que Believe será um jogo graficamente simples e maioritariamente estático, não serão utilizados quaisquer motores de jogo.

2.3 Estilos de jogo

Um género de videojogo é uma categoria específica de jogos relacionados por uma característica de jogabilidade similar e não definidos pelo seu conteúdo ou dispositivo onde se joga. Cada género pode ainda ter vários subgéneros, como por exemplo, um jogo de acção tem (entre outros) um jogo de plataformas ou de sobrevivência como subgéneros. Contudo, como simplificação, iremo-nos cingir ao género de aventura e ao subgénero novela visual, por serem o escolhido para Believe.

O género de aventura é caracterizado por uma jogabilidade que exige que o jogador resolva vários *puzzles* através da interacção com pessoas ou o ambiente sem confronto directo, isto é, tende a excluir elementos de acção para além de mini-jogos.

Dentro dos jogos de aventura, os *visual novels* destacam-se por serem focados no enredo, nos quais o jogador acompanha uma história por meio de textos, músicas e imagens, e, em alguns raros casos, cenas gravadas com atores reais. Em momentos-chave desses jogos o jogador deve decidir que caminho o protagonista deve seguir e, desta forma, o jogo avança. O desenvolvimento da trama destes jogos costuma depender das escolhas que os jogadores fazem durante o jogo.

Para além das características indicadas anteriormente, Believe será também um jogo do tipo penetrante (*Pervasive Gaming* [37]) que pode ser definido como um jogo abrangente e sempre presente. Por outras palavras, o jogo nunca pára e está a decorrer durante as 24 horas do dia. Neste tipo de jogos, o mundo virtual tem por base o mundo real, aproximando-se muito de uma experiência de realidade aumentada [44]. Na prática, a realidade do jogo traz um novo significado ao ambiente real e as acções tomadas no mundo real, influenciam o decorrer do jogo.

2.4 Casos de estudo

Dentro dos vários jogos analisados, foram escolhidos os seguintes três jogos como casos de estudo por conterem características de jogabilidade e de estilo semelhantes às que existirão em Believe. Sendo estes, Sara is Missing, Pokemon Go! e Botfighters.

2.4.1 Sara is Missing

Plataforma: iOS/Android/MacOs/Windows
Sara is Missing [22] é um videojogo móvel de terror com uma duração de 30 a 60 minutos, em que o utilizador toma posse de um smartphone e é encarregue de descobrir o que se passou com o seu dono, neste caso, Sara. É uma novela visual cuja história se desenrola por meio de imagens, texto e video sendo o utilizador o responsável por observar e analisar este conteúdo para progredir na história. Não é do tipo penetrante porque durante o decorrer do jogo não há involvência do mundo real. Porém, contém alguns momentos em que o jogador necessita de tomar decisões num curto espaço de tempo para seleccionar o rumo que pretende dar ao jogo.

2.4.2 Pokemon Go

Plataforma: iOS/Android
Um dos maiores sucessos dos videojogos durante o ano de 2016, com mais de 100 milhões de downloads só na plataforma Android, Pokemon Go [41] tornou-se num fenómeno à escala global. Um jogo de realidade aumentada com base na localização dos seus utilizadores, cada jogador tem de se deslocar no mundo real para capturar monstros e competir com outros jogadores pela posse de ginásios. Devido à natureza do jogo, muitos utilizadores relatam que a sua saúde física e mental melhorou significativamente devido à necessidade de caminhar e de comunicar com outros utilizadores. Pode ser considerado um jogo penetrante porque apesar de não entrar em conflito directo com outros jogadores, utiliza elementos e localizações do mundo real, para por exemplo, apenas disponibilizar monstros em certas áreas geográficas ou mediante de condições meteorológicas.

2.4.3 Botfighters

Plataforma: Telemóvel mediante operadora.
Produzido por uma empresa sueca e lançado na primavera de 2001, Botfighters tinha como objectivo localizar outros jogadores e vencê-los. Para escolher um outro jogador como target, o utilizador teria de enviar uma mensagem de texto com a palavra "hunt" seguido do nome do oponente. Cada mensagem de texto, devolvia uma resposta com informação sobre a proximidade do oponente e quando este estivesse próximo, a batalha começaria com uma mensagem de "shoot". Simultaneamente, o oponente receberia uma mensagem de que estava a ser atacado permitindo que os oponentes pudessem escolher entre fazer parte da batalha ou fugir. A rede GSM era utilizada para identificar se um utilizador estaria perto de outro ou não para o poder enfrentar. Botfighters é considerado como um dos primeiros grandes exemplos de jogos penetrantes por ser necessário localizar outros jogadores no mundo real, a qualquer momento do dia e em qualquer lugar.

3

Requisitos da Solução

Este capítulo apresenta uma visão abrangente do projecto e das suas potencialidades. Serão descritas algumas ideias que surgiram ao longo do tempo e que moldaram as características do produto final, acompanhadas de alguns exemplos sob a forma de modelos gráficos (*mockup*).

De seguida serão descritos os requisitos funcionais base do sistema e alguns casos de utilização para se entender a interacção e experiência de utilizador pensada para o jogo.

3.1 Visão

Na idealização do projecto, várias ideias surgiram com o intuito de tornar a experiência o mais contextual possível ao utilizador. Porém, apesar de diferente e com mais significado para o utilizador, o contexto por si só, apresenta um lado previsível e calculista que influencia o conteúdo do jogo mas não a forma de jogar.

Assim, surgiu a ideia de adicionar uma componente dinâmica e imprevisível, ocorrendo num período de tempo pré-determinado (sensivelmente 20 dias) que se inicia assim que o utilizador abrir a aplicação pela primeira vez. Isto significa, que o jogo decorrerá com ou sem a interacção do utilizador, com as devidas consequências para ambos os casos, e que o utilizador deverá interagir quando for invocado para o fazer e não quando o decide fazer.

Ao invocar o utilizador para jogar em momentos inesperados, espera-se que a interacção seja curta e rápida, tal como ler uma SMS, escrever uma nota ou atender uma chamada, para não correr o risco de perder a atenção devido à exigência prolongada de esforço cognitivo. Contudo, a sua gestão deve ser o mais oportuna possível para evitar a desinstalação do jogo por parte dos jogadores.

Imaginemos o seguinte cenário demonstrado na figura 3.1. O utilizador está a caminhar pela rua e a aplicação detecta que este está em movimento. A aplicação lança um desafio sob a forma de um *pop-up* com um componente gráfico circular. O utilizador necessitará de realizar um movimento circular contínuo para terminar o desafio com sucesso.

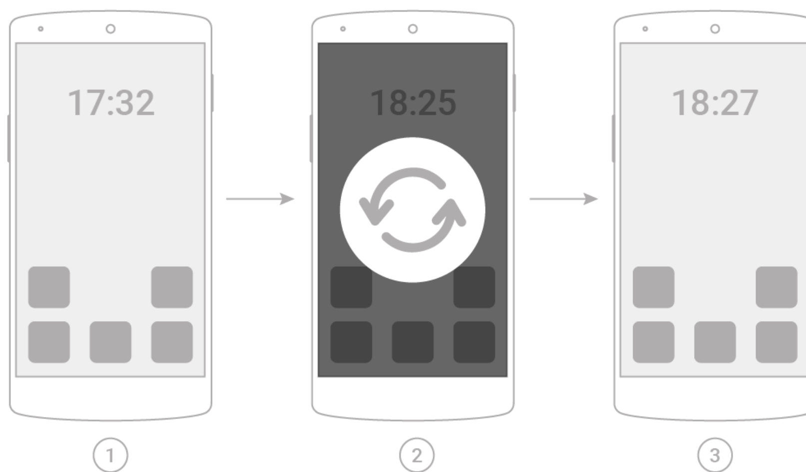

Figura 3.1: Demonstração de um desafio

De seguida ser-lhe-á atribuída uma pontuação mediante o tempo que este demorou a realizar o desafio. Caso termine o desafio sem sucesso, ser-lhe-á atribuída uma consequência que terá impacto nos desafios seguintes ou até a um nível de utilização diária do seu dispositivo, por exemplo, com o efeito de chuva (*noise*) a ser mostrado temporariamente ao longo do dia.

3.1.1 História

O jogo terá como objectivo salvar o maior número de pessoas possível ao longo de um período de 20 dias do mundo real. A história contempla a existência de seis personagens, com personalidades distintas, às quais um determinado infortúnio na sua vida (rapto, acidente grave,etc.) necessitará da ajuda do utilizador para as salvar.

Cada personagem estará em foco durante uma porção desse período de 20 dias, onde os jornais incluirão mais informações e dicas sobre si. Para ajudar a salvar cada personagem, serão lançados diariamente vários desafios (ex: como *puzzles*, *quizzes*) que o jogador terá de completar com sucesso.

Existirá também uma personagem paralela, o vilão, que tentará complicar a vida ao jogador. Caso este não responda aos seus estimulos e desafios, o vilão irá impor mais dificuldades na salvação da personagem em foco, através do lançamento de desafios mais frequentes e complicados.

3.1.2 Desafios

A resolução dos desafios será o ponto central de Believe. Estes poderão tomar várias formas, visuais ou não, e serem lançados de forma totalmente autónoma. A plataforma android e as bibliotecas disponíveis, permitem que se tome acções com base num número grande de condições, como por exemplo:

- Actividade detectada do utilizador, como caminhar ou conduzir.
- Estado dos *headphones*, se estão ligados ou não.
- Localização do utilizador.
- As condições meteorológicas do local onde se encontra.
- O dia da semana ou a hora do dia.
- Alterações de estado de internet (Wi-Fi e dados móveis).
- Percentagem restante da bateria.

Seria assim possível atribuir o lançamento de desafios caso uma (ou mais) condições se verificassem, necessitando o utilizador de realizar desafios como:

- Deslocar-se até um certo local.
- Realizar um *puzzle*.
- Responder a uma pergunta.
- Atender uma chamada falsa ou enviar uma SMS.
- Tirar uma foto a um objecto e verificar se o seu tipo corresponde ao esperado.

3.1.3 Consequências

Caso o utilizador não responda ao desafio lançado, ou o terminasse com insucesso, as seguintes consequências poderiam ocorrer como forma de punir o utilizador:

- Se não atender a chamada, a personagem pode eventualmente gritar por ajuda em alta-voz.
- Vibrações do telemóvel contínuas e não canceláveis.
- Efeito de chuva descontrolado no visor do telemóvel.
- Gasto de bateria alto, com indicação sob a forma de notificação.
- Lançar de uma notificação quando o telemóvel tiver o ecrã desligado para simular X chamadas perdidas.

É importante notar que estas sugestões são apenas algumas das possibilidades, pelo que podem a vir ou não a ser implementadas, ou até combinadas entre si.

3.2 Análise de Requisitos

Tendo todas estas ideias em consideração, foram estabelecidas três características que representarão a base do jogo, as quais são de seguida apresentadas.

Imprevisibilidade

Os hábitos e actividades do utilizador terão impacto directo nos tipos de desafios seguintes e na sua dificuldade, bem como os anteriormente resolvidos. Isto significa que o momento em que alguns desafios são lançados, dependerá não só do estado e do contexto do utilizador mas também dos seus resultados anteriores, pelo que não será fácil de replicar com outros utilizadores os mesmos desafios à mesma hora e com as mesmas características.

Facilidade de Evolução

O sistema terá de ser concebido de forma a que seja possível adicionar novos elementos (desafios, consequências, personagens) sem alterar estruturalmente o que já esteja desenvolvido e em execução nos dispositivos dos utilizadores. A título de exemplo, caso se implemente dois novos tipos de desafios e se publique com uma nova versão da aplicação, a sua inclusão terá de ser transparente para o utilizador, a mecânica do jogo deverá se manter intacta e o estado actual da sua história inalterado.

Exclusividade

O projecto terá como objectivo o desenvolvimento de um jogo primeiramente exclusivo para Android. O que significa que a implementação das funcionalidades e características definidas para o jogo, bem como as ideias que possam surgir no futuro com a inclusão de novas API's, apenas deverão cumprir a sua viabilidade no sistema operativo Android, pelo que a decisão de inclusão ou não, não deverá ter em conta uma eventual adaptação para outros sistemas móveis no futuro.

3.2.1 Funcionalidades

Como se trata de um videojogo móvel focado numa experiência pessoal, apenas existirão no máximo dois actores a interagir directamente com a aplicação. O utilizador final que jogará o jogo, e o criador que poderá acrescentar novos desafios, consequências e personagens sob a forma de actualizações da aplicação.

Da perspectiva do utilizador, este terá acesso apenas a um conjunto limitado de funcionalidades, como indica a figura 3.2.

3.2.1.1 Consultar Notícias

O jogador poderá consultar a qualquer altura do dia um jornal diário virtual com notícias do mundo real e que incluirá informações das personagens a salvar, sob a forma de notícia, declarações, acontecimentos, etc. Para o fazer, o utilizador terá de abrir a aplicação e seleccionar o ecrã das notícias, sendo que por omissão, será o primeiro a ser apresentado. Estando dentro do ecrã, é possível depois percorrer a lista de notícias e abrir o detalhe da mesma num novo ecrã através de um clique.

3.2.1.2 Guardar Notícia

Dentro do ecrã de notícias, será possível também seleccionar uma opção para guardar essa notícia para ser lida noutro dia. Essa notícia estará disponível para ser consultada num ecrã de bloco de notas com as notícias guardadas ao longo dos vários dias.

3.2.1.3 Consultar Notas

Tal como o ecrã de notícias, para abrir o ecrã de notas quando se liga a aplicação, bastará seleccioná-lo caso ainda não esteja visível. Este ecrã faz sentido na medida em que o jornal se actualizará diariamente e algumas das informações publicadas nos dias anteriores, poderão ser úteis na conclusão de desafios futuros. Ao abrir este ecrã, serão listadas todas as notas até então criadas e guardadas.

3.2.1.4 Escrever Nota

Dentro do ecrã de notas, é também possível seleccionar um botão que abrirá um ecrã que permite criar uma nova nota, podendo ser constituída por texto ou imagens. Ao preencher todos os dados, basta clicar no botão de salvar e a nota é adicionada à lista presente no ecrã anterior.

3.2.1.5 Consultar Comunicações

Para além destes ecrãs, existirá um terceiro ecrã principal que permitirá fazer comunicações através de mensagens ou efectuar uma chamada virtual com as personagens. Para abrir este ecrã, é necessário seleccionar a opção de comunicações e uma lista dos últimos contactos realizados será apresentada. De forma resumida, cada entrada nesta lista apresentará a última chamada realizada ou mensagem enviada.

3.2.1.6 Realizar Chamada e Enviar Mensagem

Ao seleccionar uma entrada dessa lista, é aberto um novo ecrã de detalhe que permite realizar uma chamada para essa personagem através de um clique para iniciar a chamada, ou escrever uma mensagem numa caixa de texto. Contudo, não é garantida uma resposta imediata (ou de todo) às suas tentativas de contacto.

3.2.1.7 Criar contacto

Neste ecrã de comunicações terá também a possibilidade de seleccionar um botão que abrirá um novo ecrã de criação de contacto, sendo necessário preencher os dados necessários, nome e número, e pressionar o botão "salvar" para guardar o contacto e voltar ao ecrã anterior.

3.2.1.8 Realizar Desafio

Por fim, o utilizador terá de resolver os desafios que forem lançados ao longo dos dias. Para o fazer, e da forma como forem apresentados, será necessários desbloquear o telemóvel e realizar as acções que forem pedidas no desafio para o realizar. Quando terminar, o desafio, o utilizador não necessita de realizar mais nenhuma acção no dispositivo.

Figura 3.2: Casos de Utilização

Por outro lado, a aplicação deverá respeitar um conjunto de restrições e regras que deverão estar sempre presentes durante o desenvolvimento.

Em termos de segurança, os dados do utilizador não podem sair da aplicação. Num momento em que a privacidade dos utilizadores é cada vez mais importante, e a constante regulação neste campo aumenta, é imperativo que dados sensíveis como os hábitos de utilização ou os dados pessoais nunca sejam distribuídos para outros repositórios de informação externos, seja de que forma for.

No que respeita ao desempenho e aos recursos utilizados, dada a necessidade de constante observação de padrões e acções por parte do utilizador e do seu contexto, bem como os agendamentos de desafios, é importante que este trabalho não se traduza num consumo de bateria considerável.

Para além disso, a aplicação deverá depender o menos possível de uma fonte de dados externa ao dispositivo móvel durante toda a sua jogabilidade, estando

preparada para a não existência de uma ligação à internet durante vários dias, de forma a não sacrificar a jogabilidade e a continuação do jogo.

Em relação à usabilidade, o jogo deverá ter desafios curtos e de interacção fácil, evitando um acrescento de complexidade na aprendizagem de resolução dos mesmos. Os desafios deverão ser independentes das consequências, salvas raras excepções, e deverá ser possível parar as suas execuções caso o utilizador assim o pretenda, com as devidas consequências.

4

Solução Proposta

Esta secção introduz a solução conceptual da proposta para a implementação do jogo Believe. Apresentará o domínio do problema e as entidades que o constituem, bem como a dinâmica e a mecânica de todo o processo. Por fim será apresentada uma arquitectura, onde se explicam os elementos que nela participam e a forma como comunicam.

4.1 Mecânica do Jogo

O objectivo do jogo será salvar todas as personagens e, de forma opcional, determinar a identidade do vilão. Cada utilizador terá também uma pontuação final consoante a celeridade com que terminou os desafios e a precisão com que o fez.

Ao longo de um período de 20 dias, cada personagem estará em foco durante três dias, sendo que cada um destes terá o seguinte propósito:

- Dia 1 - Inicia-se o capítulo com o jornal a publicar a história da personagem e os rumores e mensagens enigmáticas sobre a personagem começam a surgir;
- Dia 2 - O número de desafios e rumores intensifica-se;
- Dia 3 - Ocorre o desafio final para salvar a personagem;

É importante também reforçar que existem eventos em todos os dias do capítulo e que para o completar com sucesso, é necessário atingir uma percentagem de desafios com sucesso até ao fim do capítulo.

Capítulo	1	2	3	4	5	6	7
Percentagem	50%	55%	60%	65%	70%	75%	85%

Tabela 4.1: Percentagem mínima de desafios completados com sucesso para completar o capítulo com sucesso

Para nivelar a dificuldade do jogo e premiar/punir o utilizador, existirá uma variável implícita (que nunca deve ser explicada durante o jogo) que será actualizada com base no resultado dos eventos que o utilizador realizou e que terá impacto nos desafios a realizar.

Esta variável deve ser interpretada como "Tolerância" que o vilão tem em relação ao jogador e como se pode observar na figura 4.1, caso o seu valor se encontre numa das extremidades, existirá uma influência directa no tipo de desafios e na sua dificuldade.

Para uma experiência mais equilibrada, o jogador deverá manter o valor o mais central possível. Senão vejamos:

Extremidade A
Serão invocados mais desafios e mais fáceis. Esta situação ocorrerá quando o utilizador não estiver a interagir o suficiente com o jogo, pelo que poderão existir mais dicas para resolver os desafios, ou até consequências mais leves.

Extremidade B
Tal como a extremidade A, serão invocados mais desafios mas neste caso, mais difíceis. Ocorre quando o utilizador está a interagir demasiado com o jogo e a submeter muitas informações erradas sobre algumas das questões e pistas que o vilão vai lançando.

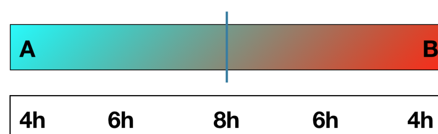

Figura 4.1: Relação entre tolerância e frequência de desafios

4.1.1 Desafios

Como indicam as figuras 4.2 e 4.3, existirão dois fluxos de lançamento de eventos em simultâneo durante o decorrer do jogo. A história ocorrerá ao longo de vários dias e em cada dia serão lançados vários desafios com base no estado actual da história, de forma cíclica. Este tipo de desafios farão parte da história do jogo e têm em vista a ajuda às personagens.

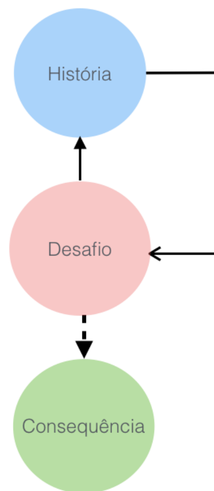

Figura 4.2: Fluxo 1

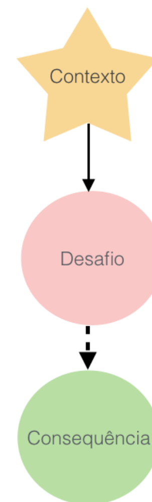

Figura 4.3: Fluxo 2

Assim, a história vai prosseguindo à medida que se for resolvendo cada desafio, sendo que o número de desafios lançados no fluxo 1, dependerá do nível de tolerância até ao momento, como indica a figura 4.1. Com base na qualidade da resolução do desafio, uma consequência pode ou não ser agendada.

Apesar de um dia ter 24 horas e sendo que em casos extremos poderião ser lançados eventos a cada 4, foi estabelecido um limite mínimo e máximo de desafios lançados por dia de acordo com a figura 4.1, tal como indica a Tabela 4.2.

Frequência	Período
3 eventos	1 a cada 8 horas
4 eventos	1 a cada 6 horas
6 eventos	1 a cada 4 horas

Tabela 4.2: Frequência de eventos

No entanto, existe também um outro fluxo (figura 4.3) alternativo de desafios que dependerão do contexto do utilizador para serem lançados. Estes, não terão influência directa na história mas apenas nas características dos desafios seguintes.

Quando um desafio é agendado necessita de ser configurado com três parâmetros essenciais para um bom funcionamento, a dificuldade, a hora e o seu tipo:

Dificuldade - Pode assumir um valor de 1 a 5, 1 sendo mais fácil e 5 sendo mais difícil, dependendo directamente da tolerância. É no entanto importante referir que o resultado do desafio anterior altera também o valor da tolerância verificado no momento, pelo que se pode assumir que o resultado do desafio anterior também tem impacto, ainda que de forma indirecta.

Hora - A hora a que o evento é lançado vai depender do número de desafios completos anteriormente nesse capítulo da história e do resultado do desafio anterior.

Tipo de Desafio - O tipo de um desafio vai depender directamente do tipo dos desafios anteriores, de forma a não o repetir.

Este tipo categoriza o desafio em relação aos outros desafios, consoante a interacção entre a aplicação, o utilizador e o ambiente que o rodeia. Será importante na medida em que permitirá escolher um desafio que exigirá do utilizador uma interacção diferente entre desafios consequentes. Como tal, foram definidos os seguintes tipos:

- *In* - O desafio terá de ser resolvido inteiramente dentro da aplicação.
- *Out* - O desafio necessita de interacção com o ambiente, como uma deslocação até um certo ponto. Quando completado com successo, o desafio terminará automáticamente.
- *Mixed* - Uma mistura de interacção entre o ambiente/contexto do utilizador e a realização de uma acção na aplicação.
- *Story* - Um desafio com esta categoria, apenas ocorrerá ao longo da história.
- *Other* - Quando um desafio não se enquadra em nenhuma das categorias anteriores.

A utilização deste tipo é importante no momento da escolha do desafio a agendar, sendo que os do tipo *Story* não entram para a selecção. A título de exemplo e utilizando os vários desafios ilustrados pela figura 4.4, assumindo que o tipo *In* foi o escolhido, tanto o desafio QA (*Question/Answer*) como o desafio *Puzzle* são dois possíveis candidatos a serem seleccionados.

É importante também indicar que quando se refere o contexto do utilizador, este tanto pode ser a sua localização, a sua actividade corrente (andar, correr, etc.), a sua identidade, entre outros.

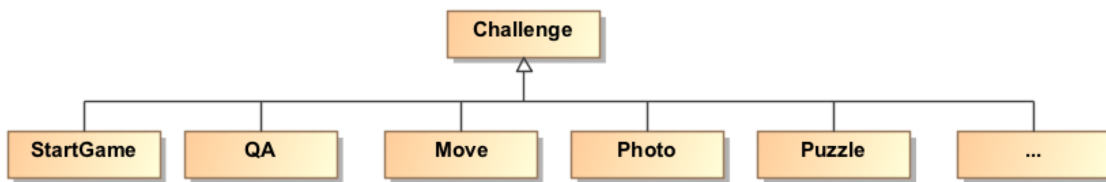

Figura 4.4: Exemplos de desafios

4.1.1.1 Agendamento

Após o processo de seleccção do próximo desafio a ser lançado, a sua execução só ocorrerá algumas horas depois. Para isso, é necessário agendar a sua execução recorrendo a um mecanismo de agendamento de unidades de trabalho (*job*) para um momento distante no tempo.

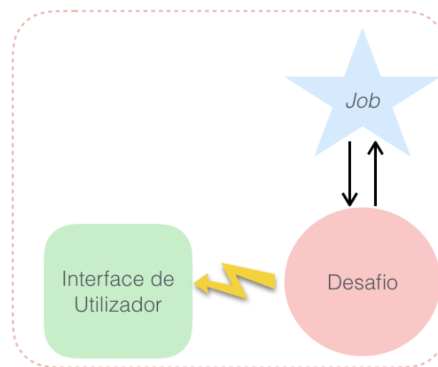

Figura 4.5: Mecanismo agendamento de um desafio

Quando o momento chegar, o desafio lançará um componente gráfico na interface de utilizador (ecrã, notificação, etc.) com o desafio a realizar, com as características definidas previamente.

4.1.2 Consequências

Apesar de ter um mecanismo de agendamento semelhante aos dos desafios, um desafio não tem necessariamente de originar uma consequência. A sua ocorrência dependerá exclusivamente do nível de tolerância nesse momento, nomeadamente, quando assume valores muito altos ou muito baixos.

Poderão apresentar-se sobre a forma de alterações gráficas (efeito de *noise*), alteração de configurações do dispositivo (nível do som), vibrações constantes, etc.

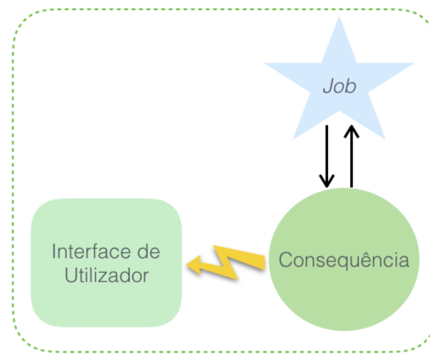

Figura 4.6: Mecanismo de agendamento de uma consequência

No entanto, uma consequência não tem necessariamente de ser negativa. Poderá existir a possibilidade de agendamento de consequências positivas de forma a compensar por uma ou várias resoluções de desafios de forma exemplar. Uma melhoria na tolerância, seria um bom exemplo a ser aplicado.

A figura 4.7 apresenta algumas das consequências definidas.

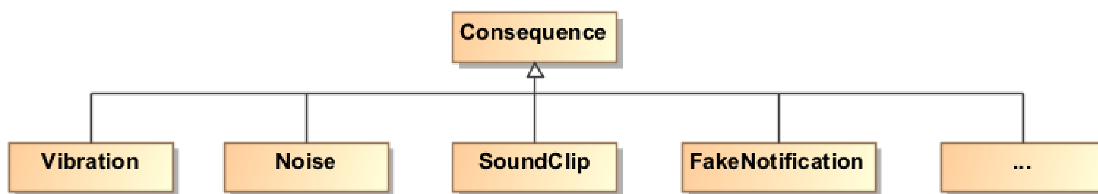

Figura 4.7: Exemplos de consequências

4.2 Estados do Jogo

Por forma a auxiliar na transição do processo de análise para o projecto, foi definido um diagrama que pretende estruturar as entidades existentes no domínio do problema (4.8).

Como indica a figura 4.9, a história do jogo será composta por sete capítulos. Cada capítulo exigirá que o jogador resolva um conjunto de desafios específicos, para além dos desafios contextuais que possam ser lançados, para salvar a personagem em foco nesses 3 dias.

Quando este período terminar, será feito um balanço global dos desafios realizados e decidido o destino da personagem. De seguida, um novo capítulo começará.

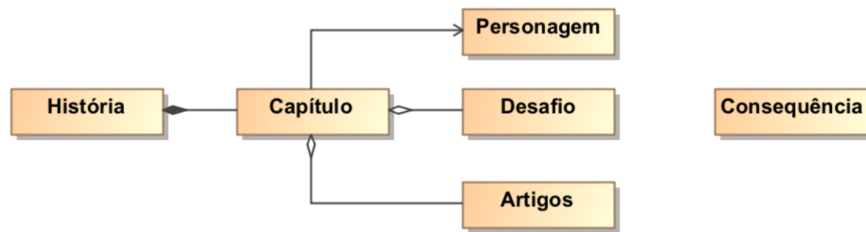

Figura 4.8: Modelo de Domínio

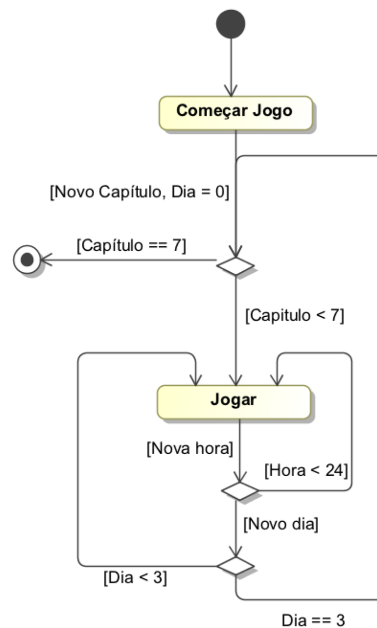

Figura 4.9: Diagrama de Estados do Jogo

4.3 Arquitectura

A solução proposta para a implementação do jogo, visa uma organização modular de forma a ser possível adicionar novos capítulos de história com novos desafios e novas funcionalidades sem a necessidade de alterar qualquer componente do jogo que tenha sido desenvolvido anteriormente.

A Figura 4.10 apresenta a arquitectura geral da solução proposta, da qual fazem parte os seguintes elementos: (1) o modelo de domínio da aplicação; (2) a camada da lógica de negócio onde vão estar específicados os casos de utilização da aplicação; (3) a camada de repositório e acesso a dados; e (4) a camada de apresentação;

De seguida detalham-se cada uma das camadas indicadas e o seu papel na arquitectura da solução proposta.

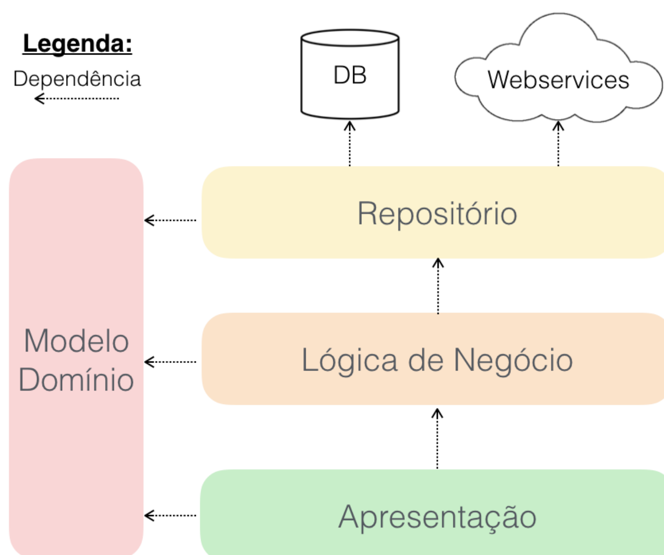

Figura 4.10: Arquitectura Geral

4.3.1 Modelo de Domínio

Esta camada define todas as classes do modelo de dados que serão transversais às restantes camadas. Ou seja, define as classes que representarão um desafio, uma personagem, etc. seguindo por base o modelo de domínio indicado na figura 4.8. Todos os desafios estarão definidos aqui, e as suas variantes, bem como as respectivas classes de unidade de trabalho responsáveis por os agendar.

Inicialmente, estava previsto um modelo de domínio específico por cada camada de software, evitando a dependência a camadas exteriores, sendo que a camada de repositório de dados não deverá conter metadados referentes à camada de apresentação.

Por exemplo, em Android para fazer o envio de dados entre ecrãs (sem consultar os repositórios) é necessário que o tipo de dados em questão seja serializado através da implementação da interface *Parcelable*.

Porém, foi decidido assumir um compromisso de partilha de um único modelo de domínio entre camadas, de forma a evitar inúmeros mapeamentos sucessivos de dados entre camadas por cada pedido transversal entre as mesmas, desde acessos à base de dados ou invocação de serviços *web*.

Outra razão para esta decisão de violação de boas práticas de software, depara-se com a muito baixa probabilidade de alteração da base de dados ou serviços *web* no futuro.

Para melhor compreender esta decisão é necessário entender algumas restrições do sistema operativo Android. A menos que indicado em contrário, todas as execuções de código são realizadas no fio de execução (*thread*) principal.

O que significa que todas as operações, desde a lógica de negócio, aos acessos a base de dados e até a renderização gráfica, é executada nesta thread. Ora, de forma a atingir uma performance de utilização de alto nível, a documentação oficial indica que todas as aplicações android deverão correr em 60 frames por segundo. Isto quer dizer que existem 16 ms por *frame* para realizar todas as operações necessárias de forma a que a aplicação se mantenha fluída.

Ora, esta restrição poderia ser facilmente contornada com a instanciação e mapeamento de dados entre as camadas em fios de execução (*threads*) alternativos. No entanto, a execução do *garbage collector* para a limpeza de objectos teria uma sobrecarga na janela temporal de 16 ms que pode não ser desprezável, visto que a quantidade de objectos por limpar seria 3 vezes superior. Para se ter uma noção e dependendo de muitos factores, a invocação do *garbage collector* atinge tempos de execução superiores a 2ms, e algumas vezes até, a marca dos 5ms.

Apesar de não ter medido o impacto concreto e comparar ambos os cenários, com a complexidade crescente nas animações e transições em android, é boa prática reservar o máximo de tempo possível para a renderização gráfica, para evitar o arrastamento visual de imagens ou interromper animações em curso.

4.3.2 Lógica de Negócio

A lógica de negócio conterá todas de classes que implementam as regras de domínio relativas a operações comuns na aplicação, como a persistência local de um desafio, a sua actualização, a obtenção de notícias, etc.

Estas operações estarão definidas nesta camada, de forma a isolar a camada de apresentação das fontes de dados e da sua lógica. Existem várias classes *Interactor* que implementam os vários casos de utilização de cada domínio em específico, seja de um desafio, da utilização da agenda ou das notícias. A título de exemplo, `NewsInteractor` disponibilizará a operação "consultar notícias" que devolve uma lista de notícias.

Para além de disponibilizar estas operações para serem invocadas na camada de apresentação, a camada de lógica de negócio faz também a gestão de dados dos vários repositórios respectivos ao domínio pelo que é responsável, isto é por

exemplo, saber obter as notícias do repositório remoto e guardá-las no repositório local, de forma totalmente autónoma.

É importante referir que na camada de lógica de negócio, a interacção com os repositórios é feita com base em interfaces, pelo que não há dependência tecnológica entre camadas, como veremos de seguida.

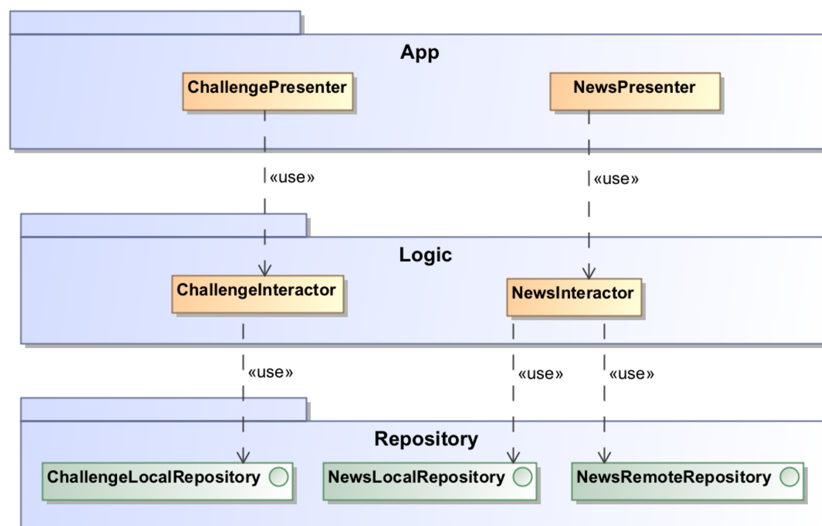

Figura 4.11: Detalhe geral da arquitectura

4.3.3 Repositório

A camada repositório implementará a lógica de acesso a dados, a sua persistência local, a obtenção de dados remotos e o consequente mapeamento para o modelo de domínio utilizado nas restantes camadas. A Lógica de Negócio é agnóstica às implementações das fontes de dados utilizadas na camada Repositório.

Assim, a definição dos repositórios é feita através de interfaces para serem invocadas na lógica de negócio. Sobre estas interfaces, são implementadas várias classes que realizarão a comunicação com os serviços *web*, no caso dos repositórios remotos, ou com as estruturas de dados em bases de dados locais, no caso dos repositórios locais.

4.3.4 Apresentação

É responsável por definir a *user interface* que será utilizada pelos utilizadores. Será implementada em Android, tirando partido do kit de desenvolvimento nativo disponibilizado pela Google.

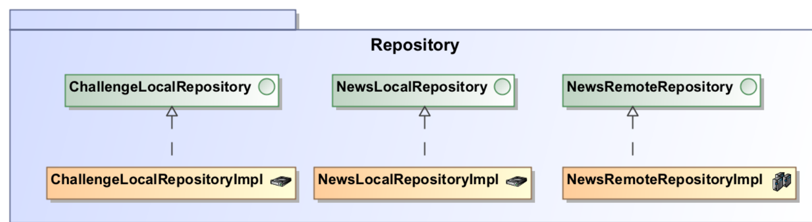

Figura 4.12: Detalhe da camada Repositório

Cada ecrã seguirá o padrão de apresentação *Model-View-Presenter (MVP)*, sendo que será representado por uma *Activity* passiva, sem qualquer lógica de apresentação e um *Presenter* que comunicará com os *Interactor* da camada de lógica de negócio.

A lógica de apresentação é assim movida das *Activity* para um *Presenter* não apenas porque é possível em Android ter várias orientações do mesmo ecrã (horizontal/vertical), permitindo centralizar a lógica apenas num único sítio, mas também porque simplifica a utilização das bibliotecas de teste sobre esses componentes gráficos, removendo a dependência à framework Android quando se pretende testar a lógica de apresentação.

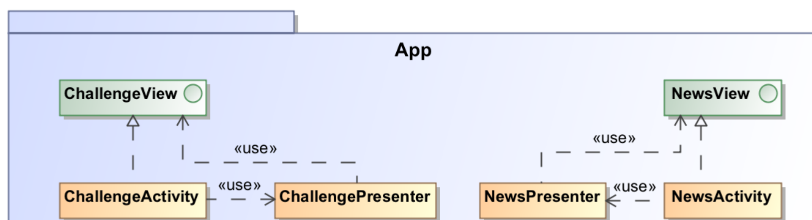

Figura 4.13: Detalhe da camada Apresentação

4.3.4.1 Model-View-Presenter

O padrão MVP é um padrão arquitectural de apresentação, criado para facilitar a utilização de testes unitários e a separação de responsabilidades na lógica de apresentação.

Senão vejamos:

- O *model* é uma interface que define os dados a serem demonstrados ou com os quais o utilizador poderá interagir.
- O *presenter* comunica com o *model* e a *view*. Obtém os dados do *model*, formáta-os e envia-os para a *view*.
- A *view* é uma classe passiva que simplesmente apresenta os dados provenientes do presenter e notifica o *presenter* das acções do utilizador.

4.4 Interface e Experiência de utilização

Para além da conclusão dos desafios, o jogo será constituído por vários componentes que ajudam na sua progressão. Estarão disponíveis a qualquer momento em que se entra na aplicação e servirão de base para toda a história do jogo.

4.4.1 Ecrã Principal

Ao iniciar a aplicação, o utilizador terá acesso a um ecrã com três separadores. Um separador para o jornal, um para um bloco de notas e um separador para possíveis contactos com as personagens. Cada acção em cada um dos separadores poderá abrir novos ecrãs com finalidades distintas, seja a criação de um contacto, a visualização do detalhe de uma notícia, etc.

Essa navegação está descrita na figura 4.14.

4.4.1.1 Jornal

O separador do jornal é dos três disponíveis, o mais importante. Conterá informações actualizadas diariamente sobre o mundo real (notícias, tempo) de forma a simular o dia-a-dia e aproximar o jogo o mais possível com a realidade. Entre as notícias, poderão eventualmente encontrar-se informações relativas à personagem do capítulo em questão.

Esse tipo de informações trará uma visão mais detalhada da personagem, da sua história, dos seus gostos e personalidade e que poderão ser elementos chave na resolução de alguns desafios. Será também possível que nas notícias apareçam

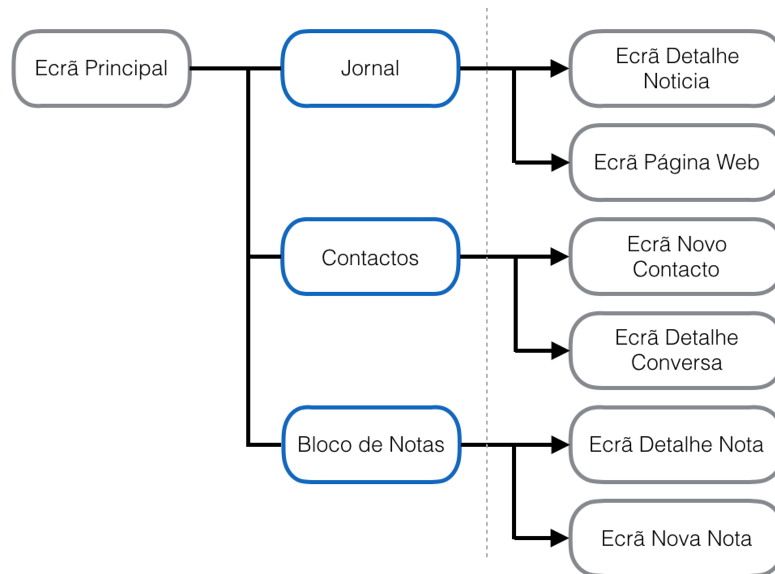

Figura 4.14: Fluxo de Navegação da Aplicação

dicas ou pistas relativas aos próximos capítulos sem nunca revelar a identidade da personagem em questão.

Como se pode observar pela figura 4.15 , quando se clica nas notícias é aberto um ecrã de detalhe dessa notícia onde será possível ler todo o conteúdo integrante da mesma. Para além das notícias, o jornal poderá conter outros componentes adicionados dinamicamente que não façam parte da história. Um desses exemplos, é a presença de anúncios comerciais com o objectivo de gerar dinheiro para suportar o desenvolvimento da aplicação, sob a forma de uma secção de classificados.

Por fim, existe também a possibilidade de algumas notícias, quando seleccionadas, poderem dar início a novos desafios, à abertura de páginas *web* para consulta, entre outros. Esta liberdade e diversidade de conteúdo apresentado permitirá acrescentar novas histórias de forma dinâmica sem grandes restrições.

4.4.1.2 Bloco de Notas

No bloco de notas será possível guardar e consultar notícias ou outros pedaços de informação recolhidos ao longo dos dias no jornal. Acumular notas será útil na medida em que permitirá consultar informações antigas para resolver alguns desafios que possam a vir a aparecer no futuro, por exemplo, os dados pessoais de uma personagem introduzida no dia 1, podem ser essenciais para resolver um desafio no dia 3.

Figura 4.15: Wireframe do Jornal

Mas mais do que isso, o bloco de notas poderá ser um auxiliar de memória para o utilizador não se perder no fluxo da história em momentos de muitos desafios e informações contraditórias. Por essa mesma razão, será possível ao utilizador introduzir as suas próprias notas ou até um conjunto de fotografias que ache relevante.

Em termos gráficos, a sua apresentação será semelhante ao separador do jornal se bem que com algumas particulares. Como por exemplo, será possível apagar as notícias guardadas anteriormente, deslizar o dedo sobre várias fotografias ou criar notas livres.

Figura 4.16: Wireframe do Bloco de Notas

4.4.1.3 Contactos

O fragmento de contactos disponibilizará ao utilizador uma forma de contactar directamente as personagens conhecidas até então, seja por mensagem ou por chamada. Será também possível adicionar novos contactos que não estejam directamente ligados à história mas em que a possível comunicação dispare um conjunto de desafios paralelos à mesma.

Contudo, tal como na vida real, diferentes pessoas comunicam de diferentes formas. Isto significa que poderão existir cenários em que a realização de uma tentativa de comunicação com alguém possa não produzir qualquer resposta.

Outra situação também possível, é a tentativa de comunicação constante por parte de uma das personagens para com o utilizador, podendo-se tornar numa situação stressante caso este não consiga parar a situação, seja de que forma for, ou pela realização de um desafio ou pela necessidade de responder à necessidade da personagem.

Como se pode ver pela figura 4.17 , é possível observar a última actividade que o utilizador teve com cada um dos contactos. Ao clicar num dos contactos, será aberto um novo ecrã com a descrição detalhada das comunicações e a possibilidade de escrever e realizar um telefonema com o contacto em questão.

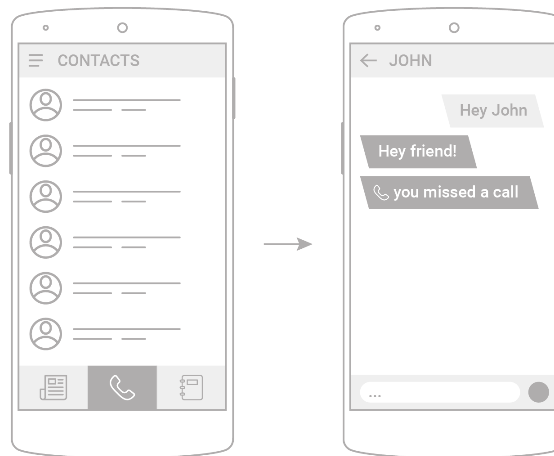

Figura 4.17: Wireframe do fragmento Contactos

5

Implementação

As aplicações Android nativas são programadas em Java ou Kotlin, ambas linguagens para a *Java Virtual Machine*. As ferramentas do *kit* de desenvolvimento Android compilam o código desenvolvido e em conjunto com os restantes recursos utilizados, dão origem a um ficheiro de extensão `.apk` que é utilizado para instalar nos smartphones Android.

Uma aplicação tem por base os quatro componentes estruturais disponibilizados pela plataforma e pode também utilizar bibliotecas externas ao *kit* de desenvolvimento, como o *Google Play Services* (disponibilizado pela Google) ou outras desenvolvidas por entidades particulares com o intuito de facilitar a programação.

5.1 Enquadramento da plataforma Android

A plataforma android disponibiliza 4 componentes base que são os blocos fundadores que ajudam a definir o comportamento geral de uma aplicação android. Nem todos são pontos de entrada reais para o utilizador e nem todos necessitam de existir numa aplicação mas cada um existe como uma entidade independente, desempenha uma função específica e tem um ciclo de vida único. Esses componentes são, a *Activity*, o *Broadcast Receiver*, *Service* e *Content Provider*.

5.1.1 Activity

Uma *Activity* é uma classe responsável por criar uma janela de interface para o utilizador interagir e é o único componente obrigatório numa aplicação android. As activities são geridas com uma estrutura do tipo pilha (*stack*), isto é, quando se inicia uma nova *Activity*, esta é colocada por cima da *Activity* anterior.

Devido às limitações dos dispositivos móveis, uma *Activity* pode ser destruída pelo sistema operativo para recuperar memória e poupar recursos (quando se encontra em segundo plano).

5.1.2 Broadcast Receiver

É um componente do android com mecanismo de publicação-subscrição que permite à aplicação reagir com base na notificação de eventos do sistema operativo, como a pouca bateria, a alteração do estado da internet, etc.

Permite também a difusão dos eventos criados pela aplicação para serem consumidos por outras aplicações, bem como, a notificação de novos dados para os *widgets* existentes no ambiente de trabalho.

5.1.3 Service

Um *Service* é um componente que representa a intenção de realizar uma operação de longa duração sem interface de utilização. Um service continua a execução mesmo que o utilizador mude de *Activity* ou de aplicação. Todas as execuções de um service, a não ser que indicadas em contrário, são executadas no contexto da *Thread UI* (Fio de execução da user interface).

5.1.4 Content Provider

Os *Content Providers* são responsáveis por gerir os acessos a conjuntos de dados estruturados (que se mantém no dispositivo em offline) e são especialmente utilizados na publicação de dados para outras aplicações. Existem alguns providers definidos na plataforma android que se podem utilizar, como por exemplo o content provider dos contactos ou o content provider do calendário.

5.1.5 Application

Por fim, o objecto *Application* não é considerado um dos quatro componentes base da plataforma mas é igualmente importante. Representa a aplicação em si e permite a execução de código antes de qualquer um dos restantes componentes.

Quando a aplicação necessita de executar um conjunto de código, é iniciado um processo linux que a representa e continuará em funcionamento até não ser mais necessário ou o sistema necessitar de recuperar memória para outras aplicações com prioridades mais altas no momento.

Isto significa que é o sistema que controla o ciclo de vida da aplicação e não a aplicação em si, através de um conjunto de regras pré-definidas (ex: memória existente, prioridade da aplicação, etc.).

5.2 Estatísticas e Distribuição

Antes de iniciar o desenvolvimento de uma aplicação android, é importante analisar as últimas estatísticas da plataforma, seja para verificar quais as versões do sistema operativo mais utilizadas ou os tamanhos e densidades de ecrã para poder tomar uma decisão informada sobre os dispositivos do público alvo.

Este tipo de informação é importante porque, por exemplo, se a grande maioria dos dispositivos tiver uma resolução alta, ou se apenas pertendermos desenvolver a aplicação para uma resolução em concreto, não há necessidade de lançar a aplicação com imagens de qualidade inferior a aquela que se está a planear, evitando-se o aumento do tamanho da aplicação sem necessidade.

Para além disso, é necessário tomar uma decisão face à versão mínima da plataforma suportada pela aplicação a desenvolver, porque apesar de ser tentador desenvolver apenas para a última versão do android para tirar partido das mais recentes funcionalidades, esta decisão limitaria muito o público que poderia utilizar a aplicação.

Assim, o que se costuma fazer quando não há restrições específicas face à versão a utilizar, é decidir por uma versão mínima que seja a mais popular no momento para poder disponibilizar à maioria dos utilizadores android e ao mesmo tempo acrescentar funcionalidades específicas de versões superiores para os utilizadores que tenham dispositivos mais recentes, sempre que possível.

Contudo, caso a aplicação seja limitada em termos de funcionalidades e não existir um esforço adicional de programação para suportar versões mais antigas, é

então aconselhado a reduzir a versão mínima suportada para disponibilizar a aplicação a um maior número de utilizadores.

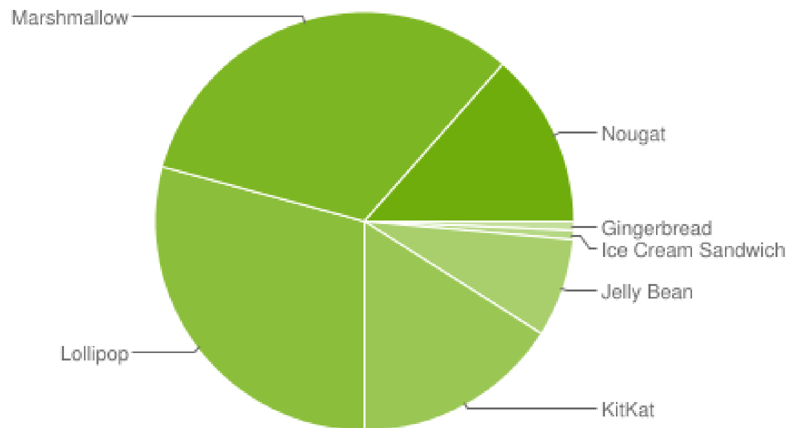

Figura 5.1: Adopção das versões do SO (referente à semana 08/08/17)

A lenta adopção das versões mais recentes do sistema operativo nos terminais dos utilizadores de android tem várias razões. Primeiro, quando a Google disponibiliza uma nova versão, os vários fabricantes necessitam de a modificar de forma a que respeite a imagem e as características que projectaram para os seus terminais, sendo um processo que leva normalmente vários meses a ser realizado.

Posteriormente, caso os terminais tenham sido vendidos por uma operadora de comunicações, estes necessitam de receber a versão alterada dos fabricantes e introduzir as suas próprias alterações e aplicações adicionais, resultando num atraso acrescido.

Por fim, pode-se também especular que alguns fabricantes não têm interesse em garantir futuras actualizações dos terminais já vendidos porque pretendem convencer o público a comprar novos equipamentos, sendo que, uma versão mais recente do android é um bom motivo de venda.

Como se pode observar pela figura 5.1, a versão mais utilizada até então é a *Marshmallow* que saiu em 2015. Mas mais interessante ainda é verificar que a versão *Kitkat* ainda tem uma base considerável de utilizadores, sendo que foi lançada em 2013.

Assim, foi decidido suportar como versão mínima, o Android *Kitkat*, permitindo alcançar mais de 90% dos utilizadores android até à data e como terminal de teste, o Nexus 5x, por ser o dispositivo de que o autor dispõe e por ser dos poucos terminais no mercado a receber actualizações directamente da Google, nomeadamente, a última versão android disponível.

5.2.1 Características únicas

Para além do android, existem outros sistemas operativos móveis no mercado, sendo o iOS [3] o segundo mais popular. Apesar de na sua grande maioria possuírem as mesmas funcionalidades, tanto um como o outro dispõem de algumas características únicas que não existem nas plataformas concorrentes.

A possibilidade de escolha de aplicações por omissão [2] para responder a certos eventos, como abrir um *link* ou enviar uma mensagem, as Instant Apps [8], que permitem a execução instântanea sem necessidade de instalação ou até a versatilidade do Picture-in-Picture [11] (que actualmente só existe nos iPad) são alguns dos exemplos disponíveis no android que podem ser utilizados nas aplicações móveis.

Para além disso, com a existência de vários pontos de entrada da aplicação, será possível dar início da mesma através de acções sobre notificações que surjam, através do clique sobre o lançador (*Launcher*) da aplicação na lista de aplicações, ou através de outros componentes que possam a vir a ser adicionados no futuro, como os widgets [13].

Este tipo de características e funcionalidades serão uma presença constante em Believe com o objectivo de introduzir imprevisibilidade e descoberta por parte do utilizador nos desafios, consequências e no decurso da história, tentando levar o progresso do jogo muito para além da aplicação em si, até ao sistema operativo que a gere e ao ambiente e contexto que rodeia o utilizador.

5.3 Estrutura e Fluxo de dados

Com a expansão das aplicações disponíveis e pela competitividade que existe no mercado, a utilização de bibliotecas que acelerem o desenvolvimento e implementem funcionalidades não incorporadas no *kit* de desenvolvimento torna-se essencial.

Contudo, é boa prática escolher as bibliotecas mais populares entre a comunidade, não só por normalmente garantirem suporte para o futuro (e novas funcionalidades) mas por cobrirem casos específicos de utilização e erros que facilmente se deixaria passar caso se implementasse essas funcionalidades de raíz. Veremos alguns exemplos utilizados mais à frente.

Para além desse tipo de bibliotecas que fornecem funcionalidades e aceleram o

desenvolvimento, existem outras que organizam, facilitam e melhoram a qualidade do código em geral e que apesar de não serem essenciais em certos contextos, a sua adopção tem vindo a aumentar. Um exemplo disso é a utilização de bibliotecas derivadas das *Reactive Extensions*.

Reactive Extensions

As *Reactive Extensions* (Rx) [39], originárias na plataforma .NET, são uma biblioteca desenvolvida que aplica os conceitos de programação reactiva. Simplificam a criação de programas assíncronos e baseados em eventos através do uso de sequências observáveis.

Tem como intuito que uma aplicação subscreva um fluxo de dados (*data stream*), denominado como uma sequência observável (*Observable*) em Rx e todas as actualizações da fonte são lhe entregues. A aplicação é passiva no processo de recolha de dados, reagindo apenas aos dados que lhe são entregues, evitando a necessidade de verificação constante de dados novos (*polling*).

Quando o fluxo deixa de emitir dados ou se deu a ocorrência de um erro, a fonte notifica o subscritor (*Subscriber*). Desta forma, a aplicação não bloqueará à espera que a fonte se actualize.

RxJava

RxJava [45] é a implementação em Java das *Reactive Extensions*. É útil na medida em que permite compor sequencialmente um conjunto de operações, permitindo utilizar operadores sobre os seus resultados, realizar o tratamento de erros de forma localizada e até indicar quais os fios de execução onde essas operações são realizadas.

Decidiu-se utilizar a biblioteca RxJava a nível arquitectural do projecto, apesar de ser possível utilizar apenas algumas das suas características (ex: operadores) de forma isolada. Ao tomar esta decisão estrutural, existem várias vantagens no desenvolvimento android.

Como se pode observar pela listagem 5.1, num só troço de código, é possível obter os dados de uma fonte (neste case o *interactor*) e de forma encadeada, declarar que esta operação é realizada num fio de execução alternativo, o resultado é enviado para o subscritor em caso de sucesso ou insucesso, tomando duas direcções possíveis, e executado no fio de execução principal (`AndroidSchedulers.mainThread()`).

```
interactor.getRandomNews()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(new ResourceSubscriber<List<Article>>() {
        @Override public void onNext(List<Article> articles) {
            mView.onNewsLoaded(articles);
        }
        @Override public void onError(Throwable t) {
            mView.onErrorOcurred(t);
        }
        @Override public void onComplete() {}
    });
```

Listagem 5.1: Troço de código em RxJava

Para além desta declaração directa, RxJava permite ainda realizar um conjunto de operações para tratamento dos dados, mapeamento ou até de acções em paralelo de forma encadeada.

5.4 Aplicação

Tal como indicado anteriormente, a plataforma Android permite executar código (no método `onCreate()`) antes de qualquer um dos quatro componentes base, através da extensão da classe que representa a aplicação (*Application*).

Este é um cenário típico indicado pelas bibliotecas externas à plataforma Android, que sugerem a sua inicialização no método `onCreate()` de *Application*, para garantir que já estejam inicializadas no momento da sua utilização.

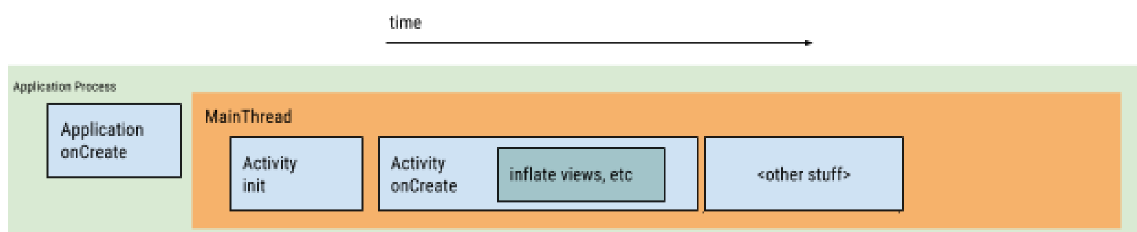

Figura 5.2: Inicialização de uma aplicação [9]

Porém é preciso ter em atenção que ao acrescentar execuções de código neste método, estamos a atrasar a inicialização da aplicação [9] e respectiva criação das activities, podendo causar um atraso considerável entre o momento de início da aplicação e a sua utilização.

5.4.1 Padrões e Ferramentas

Uma das soluções para mitigar este problema, passa pelo atraso na inicialização destas bibliotecas até o momento em que sejam utilizadas, através do uso do padrão *Singleton*. Outra hipótese, passa pela utilização de uma plataforma de injecção de dependências que cria os objectos e as suas dependências quando forem injectadas pela primeira vez.

5.4.1.1 Singleton

O padrão *Singleton* foi criado para limitar o uso de um tipo a apenas uma instância. Esta solução permite assegurar o estado desse objecto entre várias invocações e ser partilhada por vários intervenientes, sem permitir que se construa novas instâncias desse tipo.

5.4.1.2 Injecção de Dependências

O conceito básico da injecção de dependências [19] consiste na existência de um objecto que tem a responsabilidade de providenciar uma implementação específica de um tipo a uma classe que o necessita.

Esta característica permite que a classe que utiliza e depende desse tipo, não necessita de o instanciar nem de saber o seu tipo concreto.

Um dos grandes benefícios desta abordagem é uma considerável melhoria no facilidade de teste do software, visto que terá como consequência, classes modulares mais pequenas e focadas na sua única responsabilidade.

Contudo, a injecção de dependências não tem apenas em vista o teste de software. Também permite a criação de módulos reutilizáveis e substituíveis entre várias aplicações ou variantes, como por exemplo, um módulo pode ter comportamentos e valores que apenas fazem sentido durante o desenvolvimento do software mas não no momento em que este chega ao cliente.

Por fim, existem três estilos de injecção de dependências. Por via do seu constructor, através de métodos *Setter* ou através de interfaces. Para um detalhe mais aprofundado sobre estas variantes, aconselha-se a leitura de referência [19].

5.4.1.3 Dagger

Dagger é uma plataforma de injecção de dependências para Java e Android. Apesar de existirem hoje em dia duas versões, a utilizada no projecto será a versão 2 adaptada pela Google, da versão 1 criada pela Square.

Esta versão difere na medida em que as injecções são totalmente realizadas em tempo de compilação, ao contrário das soluções que o fazem em tempo de execução através do mecanismo de reflecção, resultando num impacto significativo de performance em sistemas com recursos limitados como é o caso dos dispositivos móveis.

Devido à complexidade e quantidade de funcionalidades disponibilizadas pelo Dagger, torna-se inviável uma descrição geral neste documento. Contudo, é importante referir pelo menos dois elementos básicos, o *component* e o *module*.

Um *module* é o elemento mais básico e que define as dependências. Isto significa que se deve declarar os objectos a disponibilizar nos módulos.

Um *component* é o grafo de dependências. É o elemento que combina os módulos e fornece as dependências para as classes que as necessitam.

Para evitar confusões entre nomenclaturas Dagger e Android, a seguinte secção utilizará o termo componente para descrever um *component* da plataforma Dagger e elemento para descrever um componente da plataforma Android.

5.4.1.4 Dagger em BelieveApp

Embora o código desenvolvido para android seja em Java (agora mais recentemente também em Kotlin), é normalmente diferente de código para o servidor na medida em que é preciso ter em conta as limitações de um dispositivo móvel.

Mais, uma das maiores dificuldades na criação de uma aplicação android com Dagger é que as classes dos elementos da plataforma Android são instanciadas pelo sistema operativo em si (ex: *Activity*), no entanto, a documentação do dagger recomenda que seja o próprio a instanciar todos os objectos injectados.

Com a limitação da utilização do construtor desses elemento, a solução passa pela sua injecção num método do ciclo de vida, tipicamente no método `onCreate()`, levando a soluções semelhantes à listagem 5.2:

Porém, esta solução quebra um dos princípios fundamentais da injecção de dependências, nomeadamente, o facto de que uma classe não deve saber como foi

```
public class FrombulationActivity extends Activity {
  @Inject Frombulator frombulator;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ((SomeApplicationBaseType) getContext().getApplicationContext())
        .getApplicationComponent()
        .newActivityComponentBuilder()
        .activity(this)
        .build()
        .inject(this);
    // ...
  }
}
```

Listagem 5.2: Exemplo de injecção de uma Activity

```
@Module
public class AppModule {
  public static final String SUBSCRIBE_SCHEDULER = "SUBSCRIBE_SCHEDULER",
                             OBSERVE_SCHEDULER = "OBSERVE_SCHEDULER";
  @Provides @Named(SUBSCRIBE_SCHEDULER) Scheduler providesSubscribeScheduler(){
    return AppSchedulers.Worker();
  }
  @Provides @Named(OBSERVE_SCHEDULER) Scheduler providesObserverScheduler(){
    return AppSchedulers.Android();
  }
  @Provides AppPreferences providesPreferences(){
    return new PreferencesManager();
  }
  @Provides Context provideContext(BelieveApp application) {
    return application.getApplicationContext();
  }
}
```

Listagem 5.3: Definição de AppModule

injectada. Tendo isto em conta, a Google disponibilizou posteriormente um conjunto de classes sob um package com o nome *dagger.android* [29] que permite resolver este problema.

Primeiro começa-se pela criação dos módulos. Em cada um destes, estarão definidas as dependências a disponibilizar. A título de exemplo, a listagem 5.3 indica o módulo `AppModule` que pretende disponibilizar dependências comuns a toda a aplicação, nomeadamente os schedulers e um objecto de preferências.

De seguida deverá ser criado uma interface que represente o grafo de dependências. Chamaremos de `AppComponent`. Neste `AppComponent`, define-se a função `inject()` necessária para que a plataforma saiba injectar o *Application* ao grafo

```
@Component(modules = { ActivityBuilderModule.class, ServiceBuilderModule.class,
                        FragmentBuilderModule.class, AndroidInjectionModule.class,
                        AppModule.class, DebugModule.class,
                        JobsModule.class, LoggerModule.class})
public interface AppComponent {
  void inject(BelieveApp application);
}
```

Listagem 5.4: Definição de AppComponent

```
@Module
public abstract class ActivityBuilderModule {
  @ContributesAndroidInjector(modules = QAActivityModule.class)
  abstract QAActivity bindQaActivity();

  @ContributesAndroidInjector(modules = StartGameActivityModule.class)
  abstract StartGameActivity bindStartGameActivity();
}
```

Listagem 5.5: Definição de ActivityBuilderModule

de dependências representado por `AppComponent`.

Todos os módulos utilizados devem assim ser adicionados à anotação `@Component` de `AppComponent`, inclusive o módulo `AndroidInjectionModule` já fornecido pela plataforma do Dagger, que assegura a injecção dos 4 elementos base da plataforma Android à instância de *Application*.

De entre os módulos existentes, nota-se a adição dos módulos `ActivityBuilderModule`, `ServiceBuilderModule` e `FragmentBuilderModule`. Estes são módulos especiais na medida em que através de uma convenção entre o tipo de retorno e a anotação `@ContributesAndroidInjector`, são capazes de associar um módulo (Module) específico a um elemento Android (*Activity*, *Service*, etc.).

A plataforma Dagger gerará em tempo de compilação um subcomponente Dagger que conterá o grafo de dependências para esse elemento android com o módulo de dependências indicado, sendo posteriormente adicionado ao `AppComponent` da aplicação.

Isto significa que para cada nova *Activity* com um módulo de dependências respectivo, deve ser adicionado nesta módulo.

De seguida, o objecto que representa a aplicação, neste caso BelieveApp, necessita de implementar os métodos `HasActivityInjector`, `HasServiceInjector`, `HasFragmentInjector`, `HasBroadcastReceiverInjector` ou

```java
public class BelieveApp extends Application implements HasActivityInjector,
        HasServiceInjector,HasFragmentInjector{
    @Inject DispatchingAndroidInjector<Activity> dispatchingActivityInjector;
    @Inject DispatchingAndroidInjector<Service> dispatchingServiceInjector;
    @Inject DispatchingAndroidInjector<Fragment> dispatchingFragmentInjector;

    @Override public AndroidInjector<Activity> activityInjector() {return
            dispatchingActivityInjector;}
    @Override public AndroidInjector<Fragment> fragmentInjector() {return
            dispatchingFragmentInjector;}
    @Override public AndroidInjector<Service> serviceInjector() {return
            dispatchingServiceInjector;}

    @Override
    public void onCreate() {
        super.onCreate();
        DaggerAppComponent.create().inject(this);
        //...
    }
}
```

Listagem 5.6: Definição de BelieveApp com Dagger

```java
public class StartGameActivity extends BaseActivity implements StartMVP.View {
    @Inject StartGamePresenter mPresenter;

    @Override protected void onCreate(@Nullable Bundle savedInstanceState) {
        AndroidInjection.inject(this);
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_start_game);
    }
}
```

Listagem 5.7: Exemplo de injecção numa Activity

`HasContentProviderInjector` respectivamente, se realizar injecções de um desses elementos.

Por fim no método `onCreate()` de *Application* deverá injectar a própria instância de *Application* na plataforma do Dagger.

Utilização em BelieveApp

A listagem 5.7 apresenta um exemplo de utilização após as adaptações necessárias, em que a própria *Activity* só necessita de invocar `AndroidInjection.inject(this)` para usufruir da injecção das suas dependências, não sabendo como está a ser injectada.

Para uma melhor compreensão do fluxo de dependências, a figura 5.3 descreve

os módulos utilizados e as dependências injectadas até se conseguir injectar o `StartGamePesenter` em `StartGameActivity`.

Como nota adicional, convém relembrar que as duas instâncias de *Scheduler* representam o fio de execução onde vão ser executadas as operações (numa *thread* em *background*) e onde serão observadas (na *thread* principal do android), tal como descrito anteriormente na subsecção de estrutura e fluxo de dados (5.3) aquando da utilização de RxJava.

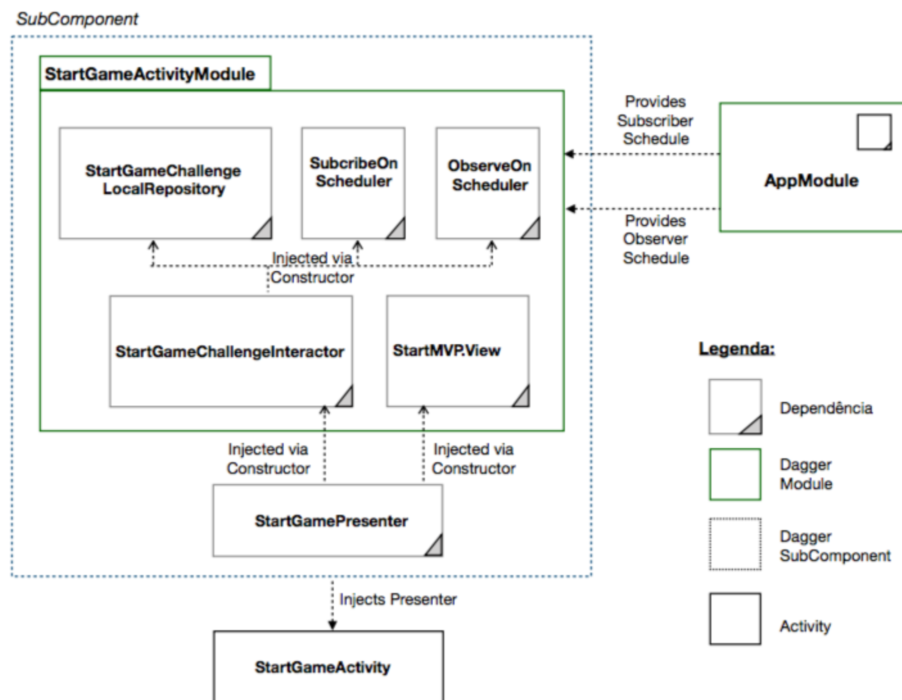

Figura 5.3: Diagrama de dependências de StartGameActivity

5.4.2 Início do jogo

Seguindo a lógica do fluxo de navegação indicado na figura 4.14, o jogo tem acesso a três ecrãs principais para interacção. O jornal, o bloco de notas e os contactos.

Contudo, a aplicação ao detectar que foi iniciada pela primeira vez, lança um ecrã (*Activity*) chamado `StartGameActivity` que contém uma pequena introdução da história e um mecanismo simples de início de jogo. Após o fim da introdução, será persistido em memória que o início já ocorreu e o jogo começa.

5.4.3 Ecrã Principal

O ecrã principal representado pela `DashboardActivity`, como podemos observar pela figura 5.4 estende o tipo `BaseActivity` que não é nada mais que um tipo subtipo de *Activity* com a introdução de um log na redefinição de cada um dos métodos do ciclo de vida da *Activity*, para efeitos de depuração.

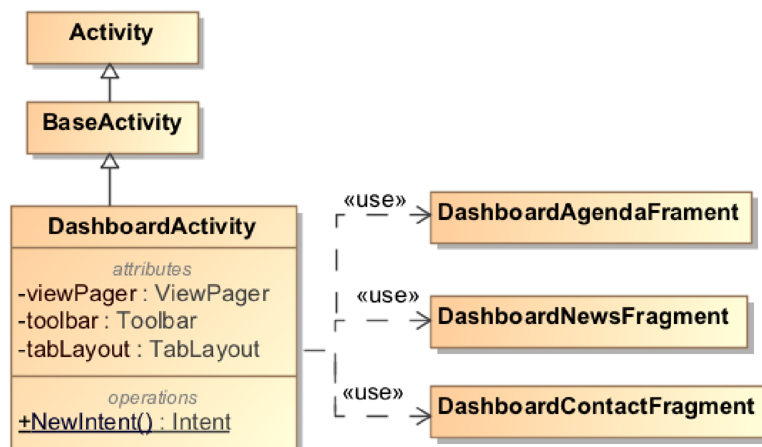

Figura 5.4: Classes do ecrã principal

Esta *Activity* contém três fragmentos que representam os três componentes indicados anteriormente e que através de separadores é possível navegar entre eles sem sair do ecrã, mantendo um contexto visual global.

5.4.3.1 Fragmentos

Todos estes ecrãs seguem uma estrutura arquitectural semelhante. Tal como as activities, também estendem um subtipo que pretende registar a invocação dos métodos de ciclo de vida.

Seguindo o padrão MVP descrito no capítulo 4, cada fragmento, implementará uma interface *View* e conterá um *Presenter* a quem requisita dados para apresentação.

O *Presenter* por sua vez recebe a *View* no seu construtor e um *Interactor* que faz a ponte entre a lógica da aplicação e os repositórios de dados.

É de notar também a existência de um `BaseSubscriptionPresenter` que foi implementado como consequência do uso da biblioteca RxJava, isto porque, cada subscrição de dados do que se faça ao repositório, a cada actualização deste último, serão reenviados de forma reactiva os dados a todos os seus subscritores.

```
@Override public void getNews() {
    execute(interactor.getRandomNews()
            .subscribeWith(new DefaultResourceSubscriber<List<Article>>() {
              @Override public void onNext(List<Article> articles) {
                mView.onNewsLoaded(articles);
              }
              @Override public void onError(Throwable t) {
                mView.onErrorOcurred(t);
              }
            }));
}
```

Listagem 5.8: Método GetNews() de NewsPresenter

Para evitar que novos dados continuem a ser reenviados a fragmentos que já não existem (quando são destruídos), correndo o risco de fugas de memória, é necessário manter uma lista de subscrições que devem deixar de ser subscritas no momento de troca de contexto visual.

A listagem 5.8 mostra a invocação de um pedido que retorna uma subscrição que é adicionada como argumento ao método `execute()` que a adiciona à lista.

Quando a plataforma android destrói os fragmentos, são removidas todas as subscrições de dados feitas pelo seu *Presenter*.

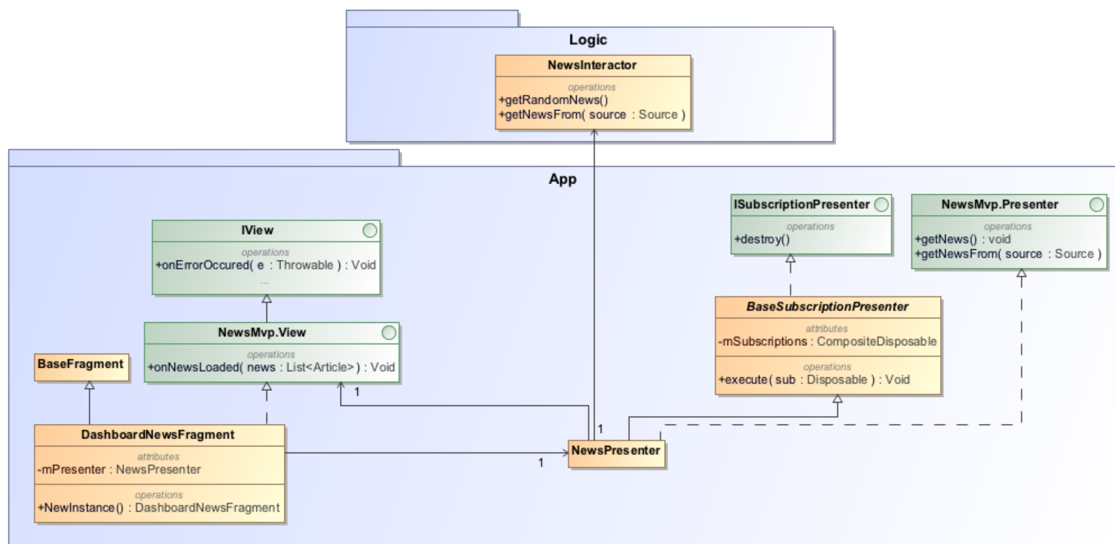

Figura 5.5: Intervenientes no fragmento DashboardNewsFragment

Por fim, todos os três fragmentos desenvolvidos implementam a estratégia de módulos de dependências descritos na figura 5.3 com a mesma convenção.

5.5 Lógica de Negócio

A camada de lógica de negócio é fulcral em dois aspectos fundamentais. Primeiro, é nesta camada que se realizarão todas as decisões relativas à história do jogo, desde o planeamento à selecção dos desafios e consequências seguintes a invocar pela camada de apresentação.

Em segundo lugar, é também aqui que se fará a utilização e gestão dos dados entre os repositórios de dados locais e remotos de forma a permitir a utilização da aplicação em qualquer cenário e da forma mais actualizada possível.

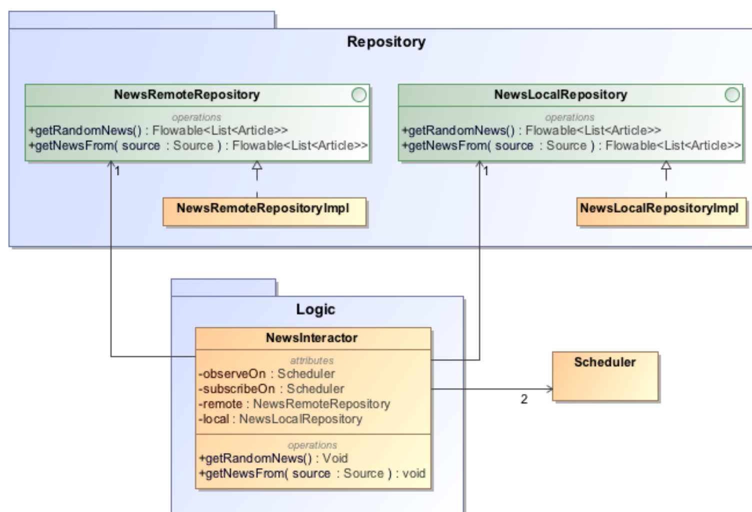

Figura 5.6: Intervenientes na lógica das Notícias

5.5.1 Desafios e Consequências

Em termos gerais, tanto os desafios como as consequências dispõem do mesmo fluxo operacional necessitando de três fases até à sua invocação. Primeiramente, a fase de decisão/detecção para definir qual o tipo de desafio ou consequência, de seguida a fase de agendamento para decidir quando os invocar e a fase de execução para os disparar e executar.

Em ambos os casos, sendo que um desafio ou uma consequência são um conjunto de operacões a executar no futuro, decidiu-se associar para cada um destes elementos, um tipo *Job* que saiba fazer essa gestão temporal.

Assim, quando decidido qual a consequência ou desafio a invocar, agenda-se a sua execução. Depois de executada, o seu resultado (no caso dos desafios) é guardado num repositório local através da invocação de um *IntentService* e planeado

um novo desafio. No caso de ser uma consequência ou um desafio contextual, este último planeamento não ocorre.

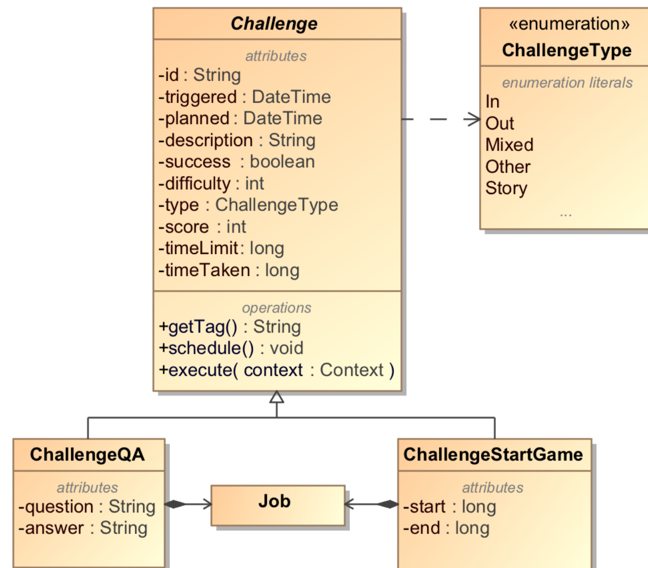

Figura 5.7: Diagrama de classes de alguns Desafios

Podemos verificar segundo a figura 5.7 que os desafios partilham um conjunto de características, como a data/hora em que foram planeados, bem como quando foram lançados, a dificuldade, o seu sucesso ou insucesso e o tipo que os categoriza.

Estes valores, serão ajustados consoante o progresso e influenciarão os desafios seguintes, no entanto, cada desafio em particular terá um conjunto de propriedades que farão sentido mediante o desafio em si, pelo que no caso de um desafio pergunta-resposta, este terá mais duas strings que representam a pergunta e a resposta correcta ao desafio.

Cada concretização de desafio terá também um *Job* associado que será responsável pelo seu agendamento e que será explicado mais adiante.

As consequências tal como os desafios, seguem a mesma filosofia em termos de estrutura e de agendamento. Estendem um tipo abstracto e definem um *Job* capaz de as lançar num futuro próximo.

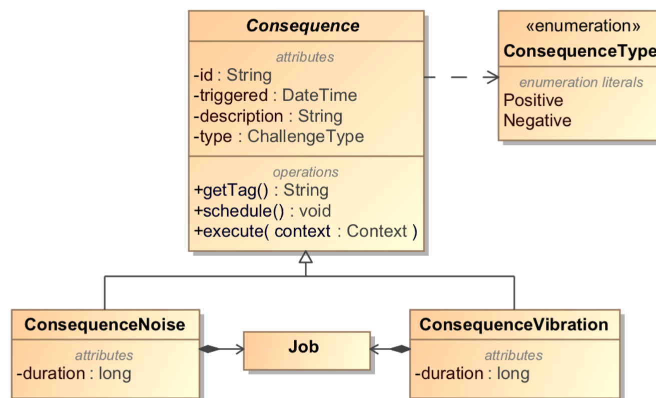

Figura 5.8: Diagrama de classes de algumas Consequências

5.5.1.1 Fase 1 - Decisão/Detecção

Believe será composto por dois tipos de desafios. Os desafios planeados, para o progresso na história, e os desafios contextuais disparados com base no contexto actual do utilizador.

Esta fase inicial do processo será responsável por decidir, no caso dos desafios planeados, e detectar, no caso dos desafios contextuais, qual o desafio correcto a agendar posteriormente.

Em relação aos desafios planeados, os tipos a agendar estão já definidos à partida, sendo que é necessário seguir um enredo e mediante o sucesso ou insucesso na resolução desses desafios, os desafios seguintes poderão ser diferentes de forma a compensar a história.

Isto significa que a título de exemplo, caso o utilizador falhe o desafio inicial, poderá ser necessário agendar um desafio nas horas seguintes para poder prosseguir com a história no dia seguinte.

No caso dos desafios contextuais, estes serão detectados através de eventos publicados pelo sistema operativo Android, como o estado de conectividade, e pela *Context Awareness* API que detectará outras componentes.

Context Awareness API

Com a adição da *Context Awareness API* [14], uma aplicação pode tirar partido de um conjunto de sinais e sensores do dispositivo com o intuito de obter dados contextuais como a localização [10], actividade, identidade e contexto do utilizador, entre outros para utilização na aplicação.

```
Awareness.SnapshotApi.getWeather(googleApiClient).setResultCallback( new ResultCallback<
    WeatherResult>() {
    @Override public void onResult(@NonNull WeatherResult weatherResult) {
        if (weatherResult.getStatus().isSuccess()) {
            Weather weather = weatherResult.getWeather();
            for(int condition : weather.getConditions())
            {
                if(condition == weather.CONDITION_RAINY){
                    //do something
                }
            }
        }
    }
});
```

Listagem 5.9: Verificação do estado metereológico

Para aceder a esses valores, a *Context Awareness API* é composta por outras duas APIs. Primeiro, a *Fence API* que pretende notificar a aplicação quando ocorrerem alterações específicas no ambiente em que o utilizador se encontra ou quando certas condições se reunirem. Em segundo lugar, existe também a *Snapshot API* que indica os valores num instante em particular, como a meteorologia.

É a partir destas duas APIs que se fará a detecção do contexto do utilizador e se seleccionará um desafio contextual. Como se pode observar pela figura 5.9, Utilizando a *Snapshot API* conseguimos detectar se na região em que o utilizador se encontra, se está a chover e em caso afirmativo, poderíamos agendar um novo desafio.

Escolha do desafio

Todos os desafios têm associados um tipo de categoria que os representa de forma a que desafios semelhantes sejam agendados temporalmente o mais afastados possível. Para além disso, as condições contextuais que despoletaram o agendamento, são também um factor decisivo na escolha do desafio, pelo que certas condições estarão associadas a vários tipos de eventos.

De seguida, dentro do conjunto de tipos de desafios seleccionados, são descartados os tipos de desafios que tenham sido lançados mais recentemente, isto é, nos últimos 2 desafios.

Por fim, é sorteado um tipo entre os disponíveis e de seguida uma instância de um desafio desse tipo. Após a selecção do desafio a agendar, as restantes variáveis explicadas no capítulo da mecânica do jogo, influenciarão directamente as

características do desafio, desde a hora de lançamento, à duração do mesmo bem como a sua dificuldade.

5.5.1.2 Fase 2 - Agendamento

A plataforma Android ao longo das versões do seu sistema operativo, foi acrescentando novos mecanismos de agendamento de blocos de trabalho, seja por serem mais eficientes ou por terem características específicas, mas a sua portabilidade para versões anteriores pode não ter sido feita ou simplesmente comportar-se de forma inconsistente.

Android-Job [15] tenta resolver este problema, abstraíndo o programador dos diferentes mecanismos e fornecendo uma API única entre todas as versões do sistema operativo.

Para utilizar a biblioteca, o programador necessita de seguir inicialmente as seguintes instruções. Primeiro, deverá definir as classes que representem as unidade de trabalho a invocar, sendo necessário implementarem o tipo *Job*.

Em segundo lugar, deverá adicionar uma instância de `JobCreator` ao `JobManager`, com o mapeamento entre uma *Tag* (valor do tipo String) e uma instância a lançar, definida anteriormente.

Por fim, para utilizar a biblioteca apenas necessita de invocar o método `schedule()` de `JobRequest` com a *Tag* correspondente ao *Job* que se pretende lançar, configurando com as características desejadas. Mais tarde, quando estiver para ser lançado, internamente a biblioteca fará uma análise de qual a instância de *Job* a instânciar e invocará o método `onRunJob()` correspondente.

Figura 5.9: Esquema de utilização de Android-Job

Integração

Sendo que um desafio (*challenge*) tem de ser agendado e executado no futuro, faz sentido que este esteja associado a um *job* capaz de realizar este comportamento.

Assim, cada tipo concreto de um desafio (ex: `ChallengeQA`) tem definido na sua classe um tipo estático *Job* que estende de `ChallengeJob`, uma classe abstracta que define o comportamento por omissão de `onRunJob()`.

Neste método, quando o *Job* for executado, o seu método base `onRunJob()` em `ChallengeJob` irá obter o challenge correspondente através do seu *Interactor*, e executar o seu método `execute()` que lançará a *Activity* correspondente a esse desafio (ex: `QAActivity`).

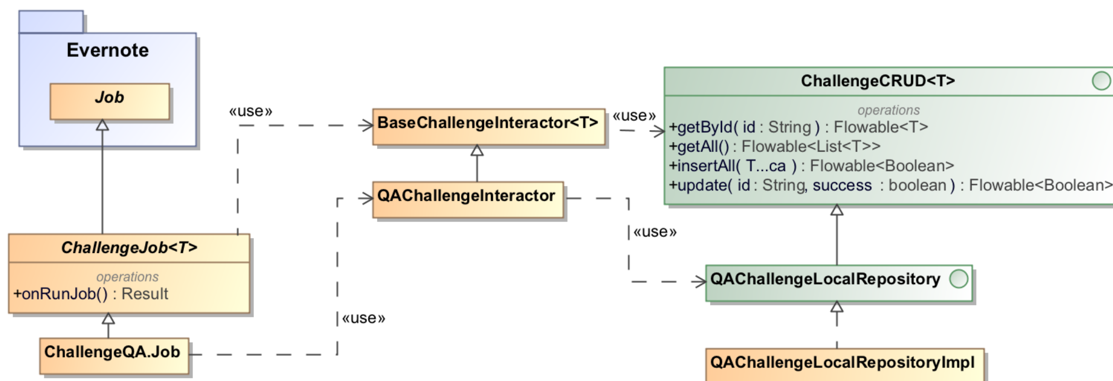

Figura 5.10: Integração de Jobs e Challenges

A figura 5.11 indica este processo de forma mais detalhada com todas as classes envolventes para o caso de um desafio do tipo `ChallengeQA`.

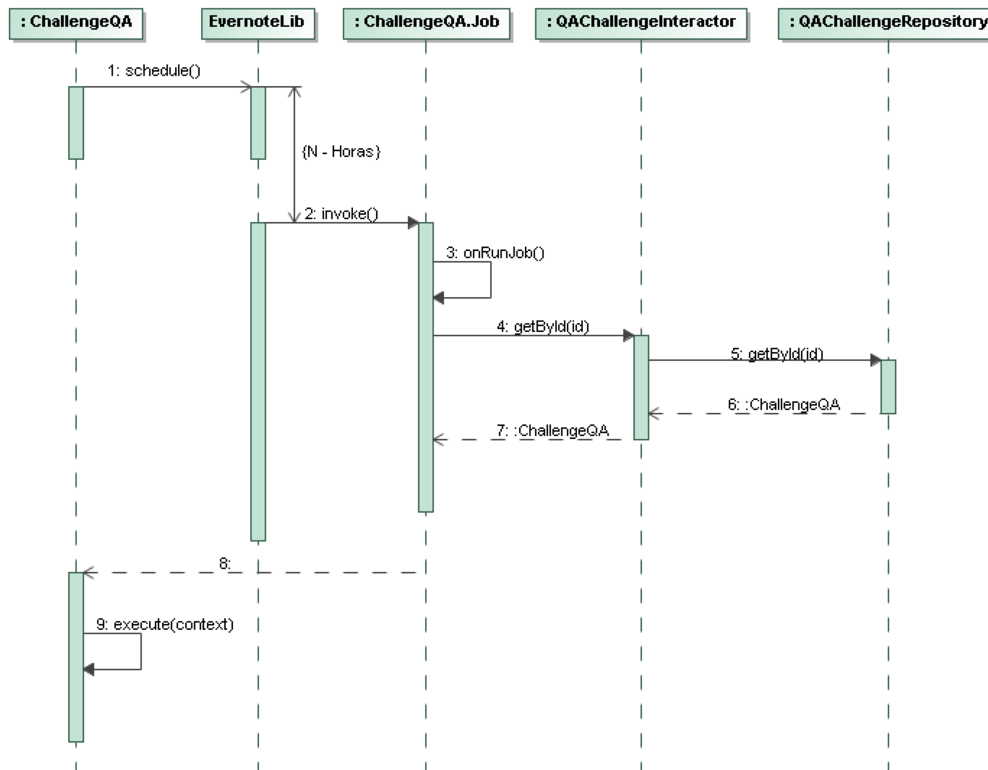

Figura 5.11: Diagrama de Sequência de um Job e Challenge

Integração com Dagger

Quando um *Job* é executado, este necessita de obter o desafio que pretende lançar, e para isso, é guardado internamente no *Job* um identificador do mesmo durante o processo de agendamento.

Mais tarde, no momento em que é executado, este obtém este identificador e acede aos repositórios de desafios para obter a instância de desafio específica.

Assim, a cada *Job* está associado um *Interactor* concreto que sabe fazer esse tipo de operações específicas desse tipo de desafio, levando a uma dependência directa.

Como cada *Interactor* tem várias dependências associadas, seria ineficiente construir todas as dependências de cada *Interactor* no `JobCreator` para cada *Job*.

É possível então usufruir da plataforma Dagger para resolver este problema. Para isso, criou-se um `JobCreator` em que à medida que estes *Jobs* vão sendo requisitados, são injectados pela plataforma.

Este módulo dagger, define assim já todas as dependências necessárias para cada *Job*, tirando partido também do facto de reutilizar a mesma instância de *Interactor*, tanto nos *Jobs* como nos *Presenters*.

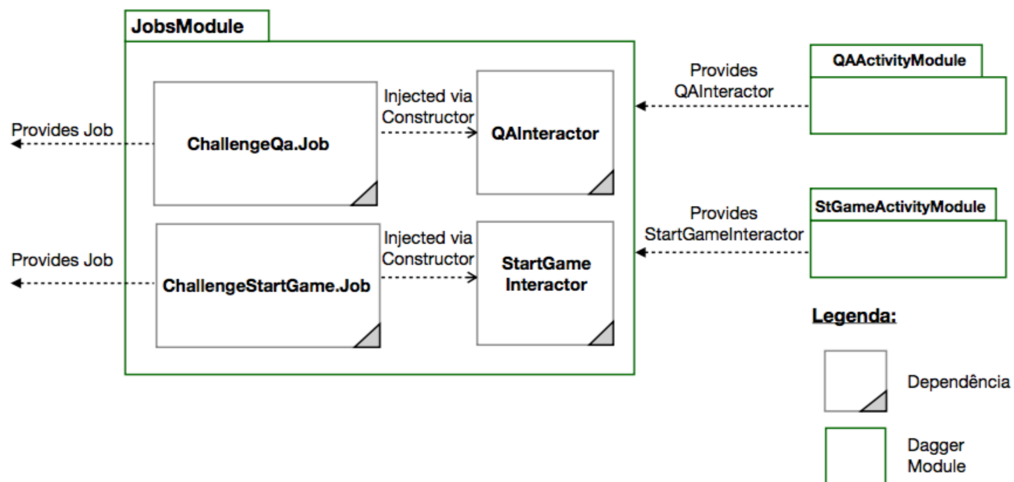

Figura 5.12: Diagrama de JobsModule

5.5.1.3 Fase 3 - Execução

A última fase do processo ocorre no momento da execução de `onRunJob()` em que cada desafio ou consequência dará inicio a uma nova *Activity* para interacção com o utilizador. A partir daí, toda a interacção ficará a cargo da *Activity* e a responsabilidade da instância de *Job* termina.

Quando terminar o desafio, é necessário proceder à gravação do resultado. Para isso, é lançado um serviço do tipo *IntentService* para realizar a gravação num fio de execução alternativo na fonte de dados correcta e de seguida actualizar o valores da mecânica do jogo.

Todos estes passos são transversais a todos os desafios e consequências, com um passo adicional nos desafios planeados para a história, sendo que é necessário definir um novo desafio e eventualmente uma consequência, voltando à fase 1 do processo.

5.5.2 Gestão de Dados

Ao contrário dos sistemas fixos com ligação à internet e fornecimento de energia constante, como o caso de computadores pessoais ou servidores empresariais, os dispositivos móveis encontram-se limitados a nível de hardware e de conexão de forma a conseguir responder à necessidade dos seus utilizadores em qualquer momento e em qualquer lugar.

Assim, o Android fornece suporte nativo para SQLite de forma a permitir a criação e utilização de uma base de dados local, evitando a necessidade de uma

ligação à internet contínua.

Apesar de nem todas as aplicações necessitarem de uma base de dados local ou sequer aceder a dados remotos, uma grande maioria das aplicações móveis serve de cliente móvel para um sistema informático já existente, o que significa, que a aplicação necessitará de obter e alterar os dados já existentes nesses sistemas.

No entanto, com o argumento de que os dados mais actualizados se encontram nesses sistemas remotos e que a implementação de uma base de dados local e respectiva sincronização é pouco vantajosa face ao custo de implementação, muitas empresas acabam por implementar as suas aplicações com acessos a fontes de dados exclusivamente remotos, resultando numa experiência de utilização decepcionante, ou até impossível, em ambientes diferentes do esperado pelos seus desenvolvedores.

Quem projecta e desenvolve uma aplicação móvel, necessita de ter em conta vários aspectos. Primeiro, a ligação à internet nem sempre é boa ou sequer possível, seja porque o utilizador não se encontra numa rede Wi-fi ou até pelas redes móveis não existirem no local onde se encontra. Mesmo existindo, a utilização de dados móveis traz um custo monetário associado que muitos utilizadores não querem suportar.

Em segundo lugar, o acesso a uma base de dados local é também mais rápido e fiável face à utilização de serviços *web*, visto que não existe a possibilidade da fonte de dados não responder ou de nem sequer existir.

Por fim, a utilização de uma base de dados local gastará menos bateria, visto que o dispositivo não necessitará de estar constantemente à procura de sinal ou de realizar operações específicas para a comunicação com o exterior, devendo assim, a comunicação com bases de dados remotas ser reduzidas ao mínimo necessário para assegurar o bom funcionamento e coerência da aplicação.

5.5.2.1 Believe

Na aplicação Believe apesar de existirem vários tipos de dados, tendo em conta a natureza da aplicação, apenas existirá necessidade para dois cenários distintos de gestão dos dados.

Primeiro, e o mais complexo, a gestão das notícias. Sendo que as notícias serão actualizadas uma vez por dia e obtidas de uma API de notícias externa, será necessário fazer um pedido remoto de obtenção de notícias e respectiva escrita

dessas mesmas notícias localmente. Isto permite que consultas e leituras de notícias subsequentes apenas consultem a base de dados local, não sendo necessário realizar um novo pedido remoto.

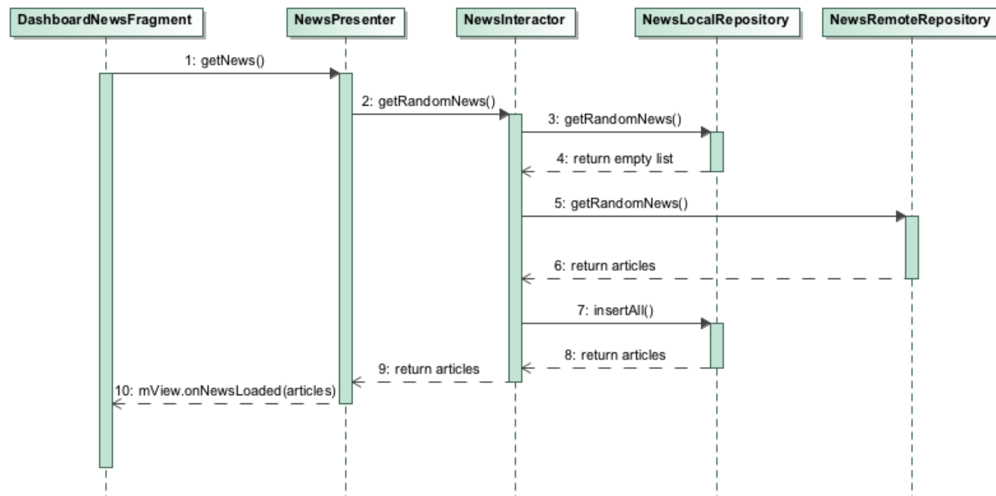

Figura 5.13: Diagrama de sequência de um pedido de Notícias

Em segundo lugar, todas as notas guardadas no contexto do jogo, serão mantidas e consultadas localmente. Em relação aos contactos, apesar de não estarem disponíveis no jogo desde o seu início, estes já irão empacotados nos recursos da aplicação e serão "desbloqueados" à medida que o utilizador vai progredindo no jogo. Quanto aos desafios, estes serão criados localmente mediante as condições já indicadas anteriormente.

Resumindo, a aplicação believe realizará pedidos de leitura e escrita a repositórios locais mas apenas de leitura a repositórios remotos.

5.6 Repositório

Apesar de em Believe a gestão dos dados ser realizada através de interfaces na camada de lógica de negócio, é na camada de repositório que se realiza a sua implementação.

Retrofit [49] foi a biblioteca escolhida para realizar as operações com serviços web e Room [12] a biblioteca eleita para a comunicação com a fonte de dados local. Ambas as bibliotecas foram escolhidas por serem de utilização fácil, por serem relativamente pequenas e com um alto desempenho, resultando numa importância em crescendo na comunidade de android.

```
public interface NewsApi {
        @GET("articles")
        Flowable<GetNewsDTO> getRandomNews(@Query("apiKey") String apiKey,@Query("
            source") String source );
        @GET("articles")
        Flowable<GetNewsDTO> getNewsFromSource(@Query("source") String source, @Query(
            "apiKey") String apiKey);
}
```

Listagem 5.10: Definição do endpoint "/articles" em Java

```
@Override
public Flowable<List<Article>> getNewsFrom(Source source) {
    return AppApi.getNewsService()
            .getNewsFromSource(source.getName(),mProvider.getNewsApiKey())
            .map(new Function<GetNewsDTO, List<Article>>() {
                @Override
                public List<Article> apply(@NonNull GetNewsDTO getNewsDTO) throws
                    Exception {
                    return Arrays.asList(getNewsDTO.getArticles());
                }
            });
}
```

Listagem 5.11: Invocação do pedido "/articles" em Java

5.6.1 Retrofit

Utilizado para realizar pedidos HTTP a vários serviços *web*, Retrofit é um cliente HTTP *type-safe* que gera interfaces Java através da indicação de uma API HTTP fornecida.

Como indica a listagem 5.11, a invocação de um pedido de obtenção de notícias é tão simples como aceder ao objecto que representa o endpoint do serviço e invocar a função definida anteriormente.

5.6.2 Room

A plataforma do Android fornece suporte nativo para a utilização directa de interrogações e operações SQL. Apesar das suas APIs serem muito complexas e flexíveis, são relativamente de baixo nível e necessitam de algum tempo e esforço para a sua utilização correcta.

De forma a contornar este problema, a Google disponibiliza a biblioteca Room, que pretende ser uma camada de abstração sobre o SQLite, permitindo o acesso

a base de dados de forma rápida sem comprometer as suas funcionalidades, seguindo o padrão DAO.

5.6.2.1 Padrão DAO

O padrão DAO (*Data Access Object*) é um padrão que visa reduzir o acoplamento entre a lógica de negócio e a persistência de dados. Desta forma, qualquer alteração na fonte de dados, seja por uma substituição integral ou uma modificação de uma implementação específica, necessita apenas da adaptação dos respectivos DAO, mantendo a lógica de negócio intacta.

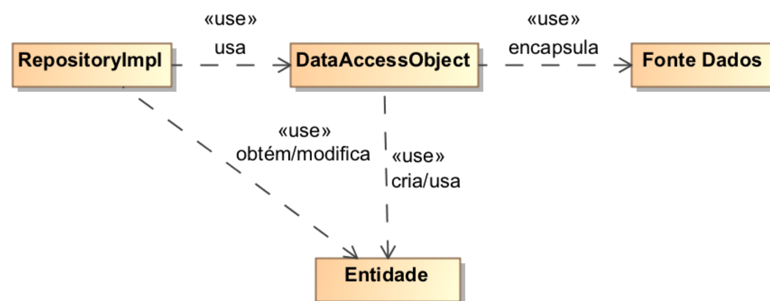

Figura 5.14: Diagrama simplificado do Padrão DAO

Este padrão é constituído por 3 elementos. Uma fonte de dados que pode tomar a forma de um ficheiro, uma base de dados SQL ou outro qualquer implementação. Um conjunto de DAOs que definem os métodos necessários para aceder à fonte de dados e por fim, as entidades que representam uma instância na fonte de dados, ex: uma linha numa tabela SQL.

A implementação deste padrão em Room é bastante mais simples porque todo o código de implementação dos DAOs e respectivas interrogações em SQLite são geradas automaticamente. Assim, o programador apenas necessita de definir as entidades que vai persistir, através da criação de classes anotadas com `@Entity`.

De seguida, é necessário criar as classes que representam os respectivos DAOs e as funções que se consideram necessárias para a lógica de negócio, de acordo com um conjunto de regras definidas pela biblioteca.

Por fim basta apenas criar uma classe anotada com `@Database` e que estenda de `RoomDatabase`, onde estarão listados todos os DAOs pertencentes a esta base de dados. Findo estes passos, no momento da compilação, todo o código de implementação será gerado automaticamente.

```
@Dao
public interface ChallengeStartGameDbDao {
  @Query("SELECT * FROM ChallengeStartGame")
  Flowable<List<ChallengeStartGame>> getAll();
}
```

Listagem 5.12: Definição de um DataAccessObject

```
public class StartGameChallengeLocalRepositoryImpl implements
    StartGameChallengeLocalRepository {
      @Override public Flowable<List<ChallengeStartGame>> getAll() {
        return BelieveApp.getDatabase().challengeStartGameDao().getAll();
      }
}
```

Listagem 5.13: Invocação da operação no Repositório

Como podemos observar pela figura 5.13, para obter localmente um desafio, apenas é necessário realizar a operação `getAll()` do *Data Access Object* da tabela respectiva.

5.7 Dependências do Projecto

Por algumas razões já indicadas anteriormente, a utilização de bibliotecas externas ao kit de desenvolvimento, foram cruciais no projecto e apesar de nem todas terem sido detalhadas, é importante descrever resumidamente quais as que foram utilizadas.

Por simplificação, será indicado apenas o nome da dependência, a sua versão e uma breve descrição.

Nome	Versão	Descrição
RxJava	2.0.8	Permite a criação de programas orientados a eventos e assíncronos através de sequências observáveis.
Android Support	26.0.1	Acrescenta classes e funcionalidades não disponíveis na plataforma do android para um desenvolvimento mais facilitado e para suportar mais dispositivos.
Bugsnag	3.9.0	Monitoriza os erros com impacto no utilizador nas aplicações web e móveis.
Android-job	1.2.0-alpha3	Cria e gera unidades de trabalho nos fios de execução alternativos.
RxAndroid	2.0.1	Fornece um Scheduler que agenda trabalho no fio de execução principal ou em qualquer Looper
Awareness	11.0.4	Unifica 7 sinais de localização e contexto numa única API.
Retrofit	2.2.0	Um cliente HTTP type-safe para Java e Android.
Logging-Interceptor	3.4.1	Um interceptor para o OkHttp que regista nos logs todos os pedidos HTTP e respectivas respostas.
ButterKnife	8.5.1	Procura as views com os ID's correspondentes no layout e automaticamente faz uma conversão explícita para os tipos correspondentes na classe em Java.
LeakCanary	1.5.1	Detector de fugas de memória em Android e Java.
Room	1.0.0-alpha9	Fornece uma camada de abstração sobre SQLite e a sua utilização de uma forma simples.
Dagger	2.11	Injector de dependências em Java e Android.
JodaTime	2.9.9	Uma adaptação da biblioteca Joda-Time para Android.

Tabela 5.1: Dependências utilizadas

6

Teste e Qualidade de Código

A definição de um bom conjunto de testes é vantajosa na medida em que garante a coerência e a funcionalidade do sistema nos vários estágios de desenvolvimento, inclusive, na adição de novas funcionalidades quando o produto já se encontra no mercado.

Apesar de negligenciados nos anos iniciais da plataforma, o desenvolvimento android beneficiou nos últimos 2 anos de uma grande melhoria tanto na quantidade como na qualidade das ferramentas de teste, para além de uma documentação mais aprofundada e actualizada.

Para garantir a qualidade do código e do produto desenvolvido, as secções seguintes descreverão quais as abordagens utilizadas, desde as variantes da aplicação implementadas, até aos testes e ferramentas utilizadas.

6.1 Testes

A melhor forma de garantir que um código funciona é testando-o, através da implementação de testes. Correr os testes a cada introdução de código no produto, permite detectar erros o mais cedo possível no processo de desenvolvimento, garantindo compatibilidade com o resto do sistema.

Na elaboração de uma *suite* de testes equilibrada, existe o conceito de pirâmide de testes [21], que argumenta que se deve fazer mais testes unitários de baixo

nível, isolados e focados, do que testes ponta-a-ponta a executar sobre a interface de utilização, que são mais lentos e rígidos, tal como mostra a figura 6.1:

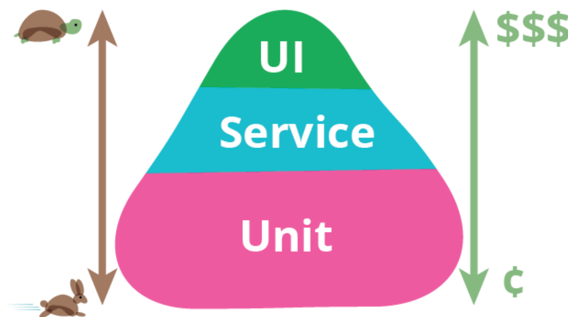

Figura 6.1: Pirâmide de testes por Martin Fowler [21]

6.1.1 Testes Unitários

Os testes unitários deverão ser os mais rápidos e com menor necessidade de recursos para a sua execução. Correm sobre a máquina virtual do Java e devem cobrir toda a lógica de aplicação e as suas dependências associadas. Como são rápidos de serem executados, podem ser incluídos no processo de desenvolvimento e serem utilizados de forma mais regular.

Em android, existem várias ferramentas para este efeito, sendo o JUnit [38] e o Mockito [40] as mais utilizadas. Em relação ao Mockito, este permite a criação de *mocks* que são classes geradas automaticamente, e que simulam o comportamento dessas mesmas classes.

Assim, como podemos observar pela listagem 6.1, verificamos que com uma instância de `NewsInteractor`, ao invocarmos o seu `getRandomNews()`, internamente o pedido está a ser reencaminhado para repositório que, por sua vez, está a invocar o seu `getRandomNews()` como seria de esperar.

6.1.2 Integração

Os testes de integração são úteis para verificar a interacção entre o código e as restantes partes do sistema mas sem a complexidade acrescida de uma plataforma UI.

Para a realização de testes de integração, é comum a utilização da plataforma de testes Robolectric [46], que executa testes em conjunto com ficheiros `.jar` da

```
public class InteractorUnitTests {
    @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();
    @Mock private NewsRemoteRepository mockNewsRemoteRepository;
    @Mock private NewsLocalRepository mockNewsLocalRepository;
    @Mock private Scheduler scheduleOn,observeOn;

    @Test
    public void verifyRemoteRepositoryUsed() throws Exception{
      // Assign
      NewsInteractor interactor = new NewsInteractor(scheduleOn,observeOn,
          mockNewsLocalRepository,mockNewsRemoteRepository);
      // Act
      interactor.getRandomNews();
      // Assert
      verify(mockNewsRemoteRepository).getRandomNews();
    }
}
```

Listagem 6.1: Teste verificação de comportamento com Mockito

plataforma Android mas na máquina virtual do java. Isto é possível através da manipulação de *bytecode* que simula os componentes do sistema android.

Como resultado, os testes são mais realísticos e aproximados à realidade do sistema mas sem a rigidez e sobrecarga de recursos na eventualidade de teste face à plataforma real do Android como num dispositivo ou emulador.

6.1.3 Instrumentação/UI

Os testes na interface de utilizador são os mais lentos e consumidores de recursos. São lentos porque necessitam de ser executados num emulador ou um dispositivo.

Mais do que isso, certos fabricantes costumam alterar algumas características do sistema operativo, o que pode resultar em ligeiras diferenças nas interfaces nas aplicações ou comportamentos inesperados.

São também os testes mais caros monetariamente, na medida em que para detectar estas diferenças ou erros, é comum a utilização de uma rede de dispositivos físicos com um servidor de integração contínua para os executar ou a subscrição de um serviço de teste na nuvem.

Não obstante, os testes automáticos na interface de utilização podem detectar problemas não só relacionados com a interface de utilização mas tambem com o hardware, firmware ou na retrocompatibilidade.

```
@RunWith(AndroidJUnit4.class)
public class ActivityUITests {
    @Rule
    public IntentsTestRule<StartGameActivity> mActivityRule = new IntentsTestRule<>(
        StartGameActivity.class);

    @Test
    public void useAppContext() throws Exception {
        // Assign
        Context appContext = InstrumentationRegistry.getTargetContext();
        // Act & Assert
        assertEquals("org.sharednode.eventsexp", appContext.getPackageName());
    }

    @Test
    public void checkStartGameClicked() {
        // Assign & Act
        onView(withId(R.id.button_start)).perform(click());
        // Assert
        intended(hasComponent(DashboardActivity.class.getName()));
    }
}
```

Listagem 6.2: Teste de início de activity com Espresso

Para a realização destes testes é comum a utilização de ferramentas como o Espresso [30], Robotium [47] ou o UI Automator [32]. Na listagem 6.2 podemos ver a utilização de Espresso para verificar que ao clicar sobre o botão de start, uma nova *Activity* é iniciada.

6.1.4 Cobertura de Testes

A implementação de testes é útil na medida em que permite saber quais as componentes do código que funcionam ou não. Para detectar quais as porções de código que estão a ser testadas, é possível utilizar ferramentas de cobertura de testes para esse efeito.

Tal como Martin Fowler refere no seu blog [20]:

> *"Test coverage is a useful tool for finding untested parts of a codebase."*

Assim, foi introduzido no projecto a ferramenta Jacoco [4], que fornece relatórios sobre a proporção de código que está a ser testada.

Através de um *plugin* já existente no Gradle (o *build system* do Android Studio) é possível adicionar a geração de um relatório no formato `.csv`, `.html` ou `.xml`,

bastando apenas configurar essa informação no ficheiro `build.gradle` do módulo da aplicação.

6.2 Qualidade de Código

Para assegurar qualidade de código, para além de código simples e conciso em conjunto com o uso de boas prácticas de software, é necessário manter o código livre de falhas (*bugs*) e reduzir a complexidade desnecessária e a escolha de soluções momentaneamente simples de implementar mas com um custo acrescido no futuro, ao mínimo possível.

É possível atingir essas características através do uso de ferramentas de análise estática que detectam *bugs* antes de serem introduzidos no produto e permitem a análise de métricas que informam sobre o estado actual do código e do seu progresso.

6.2.1 Análise de Código

Durante o desenvolvimento do projecto foi utilizado o projecto FindBugs [17] que realiza análises estáticas de código Java.

Introduzido via *plugin* FindBugs-IDEA no ambiente de desenvolvimento Android Studio, detecta erros comuns como eventuais `NullPointerException` através de desreferenciação de variáveis, ciclos infinitos, maus usos de bibliotecas Java, *deadlocks*, etc.

Como podemos ver pela figura 6.2, após a execução da análise do módulo da aplicação, uma das más práticas detectadas foi a existência de métodos com letra maiúscula, em vez de minúscula.

Figura 6.2: Exemplo de utilização do FindBugs

Em conjunto com o FindBugs, é utilizada uma outra ferramenta de análise designada Android Lint [43], e que está integrada no ambiente de desenvolvimento

Android Studio. Permite a detecção de problemas comuns em Java mas também de *bugs* específicos no desenvolvimento Android, como a falta de traduções, problemas de desempenho nos *layouts*, recursos não utilizados, problemas de usabilidade como a falta de um *input type* numa caixa de texto, etc.

Estando integrado no ambiente de desenvolvimento, é possível verificar estas detecções enquando se desenvolve.

6.2.2 Estilo de Código

Uma configuração nem sempre tida em conta mas igualmente importante num desenvolvimento em equipas, é a utilização de um estilo (*codestyle*) comum entre todos os programadores. Uma adopção comum por parte de uma equipa (ou não), irá tornar o código mais legível e homogéneo, através de uma convenção comum.

Como resultado, deixarão de existir alguns conflitos nas versões de controlo dos ficheiros de código fonte, que ocorrem devidos aos vários estilos e formatações de cada programador. Mais, permitirá também que seja mais fácil navegar por todo o código fonte e facilitará as revisões de código.

O estilo utilizado no projecto, foi Google-Java-Format [31], um estilo que segue as regras de estilo definidas pela Google para a linguagem de programação Java.

6.3 Variantes da Aplicação

O Android Studio, ambiente de desenvolvimento utilizado, utiliza o Gradle como ferramenta de automação e compilação da aplicação, permitindo a criação de várias configurações com a finalidade de criar e produzir artefactos executáveis (`.apk`) com características diferentes.

Uma das possibilidades é a configuração de diferentes *productFlavours* [34] que representam diferentes versões da aplicação a serem distribuídas pelos utilizadores, tal como uma versão gratuita ou paga.

Esta diferenciação permite a utilização de código ou recursos diferentes enquanto se partilha partes comuns entre as várias versões da aplicação. Contudo, não está planeada a criação de diferentes *productFlavours*.

Por outro lado, é possível criar diferentes *buildTypes* [33]. Os *buildTypes* definem

certas propriedades que o Gradle utiliza durante o processo de *building* e de *packaging* e são típicamente configuradas para diferentes fases do ciclo de vida do desenvolvimento.

Por exemplo, caso a aplicação esteja em modo depuração, a aplicação é assinada com uma chave de depuração, enquanto que no modo de publicação (*release*) é utilizada uma chave diferente, o código é obfuscado, etc.

Apesar de se ter feito algumas configurações básicas, como a obfuscação do código através de Proguard, a remoção de recursos não utilizados ou a utilização de uma chave de release, foi também utilizado uma variável estática gerada pelo gradle que permite verificar no código desenvolvido, qual o *buildType* corrente para fazer uma decisão.

Neste caso, e para efeitos de teste e experimentação dos vários desafios programados, foi utilizada esta variável booleana para acrescentar um fragmento específico de depuração no *dashboard* da aplicação quando em modo de depuração.

Este fragmento permite verificar quais os *Jobs* programados, quais os invocados, e antecipar o próximo agendado.

Processo de Desenvolvimento

Para desenvolver a aplicação e de forma a testar o conceito do jogo sem compromissos de implementação numa fase embrionária, decidiu-se optar por um desenvolvimento orientado à funcionalidade.

Através da implementação de um pequeno desafio, esta decisão mostrou-se adequada para o arranque do projecto porque permitiu ter uma ideia de como todas as camadas se devem interligar e comunicar, criando posteriormente um fluxo que sirva de base para os restantes desafios.

Após a criação de um protótipo funcional e de uma ideia arquitectural, iniciou-se um processo de desenvolvimento sistemático de forma a ser possível uma adaptação mais fácil e mais ágil às alterações de requisitos para o projecto.

7.1 Processo Ágil

Um processo ágil de desenvolvimento [6] pressupõe uma rápida adaptação à mudança, um desenvolvimento sustentável e célere orientado à funcionalidade com valor para o cliente.

Para executar um projecto nessas condições, um processo ágil tem por base quatro características centrais que devem ser mais valorizadas em contrapartida ao que acontece em processos em cascata:

- O foco em indivíduos e interacções em detrimento de processos e ferramentas;
- O foco em software funcional em detrimento de uma documentação abrangente;
- A colaboração com o cliente em vez da obsessão e rigidez de uma negociação contratual;
- Saber reagir à mudança em vez de seguir unicamente um plano.

7.2 Prototipagem

A prototipagem de software é um processo interativo baseado no atendimento de requisitos ainda pouco definidos, permitindo a descoberta de falhas difíceis numa fase embrionária do projecto e simulando a sua aparência e funcionalidade, permitindo que todos os intervenientes no projecto, desde os programadores aos clientes, percebam os requisitos do sistema podendo interagir, avaliar, alterar e aprovar as características mais importantes.

A prototipagem é uma característica importante do processo ágil adoptado, visto que através do desenvolvimento de protótipos é possível efectuar a consolidação das funcionalidades desenvolvidas até ao momento permitindo uma validação da forma como estas estão implementadas e garantindo a sua utilização real.

7.3 Ferramentas de Suporte

7.3.1 Gestão de Projecto

De forma a atingir os objectivos pretendidos, é necessário realizar uma gestão controlada e apropriada do projecto mediante todas as suas características ao longo de todo o processo de desenvolvimento.

Essa gestão vai desde a adaptação dos objectivos durante a vida do projecto, à gestão de tarefas por prioridades, à comunicação com os clientes até à iteracção e distribuição frequente do produto.

7.3.1.1 Trello

Trello [50] é uma aplicação *web* e móvel simples, gratuita e fléxivel para gestão e organização de qualquer tipo de projectos. Quer seja para gerir uma equipa, uma viagem, lista de compras ou um projecto, com o Trello é possível criar vários quadros e notas para organizar de forma colaborativa com outros colegas, amigos ou família.

Entre outras opções, permite personalizar fluxos de trabalho, adicionar listas de a-fazer em cartões, comentar as várias tarefas e anexar ficheiros, tornando-se numa peça importante em qualquer projecto.

Em relação a Believe como se pode ver pela figura 7.1, foi criada uma conta gratuita e feita a organização de cartões por várias àreas, seja desenvolvimento, interface de utilização, documentação etc. Dentro de cada àrea, foram definidas as várias tarefas e atribuídas uma cor de prioridade a cada uma delas.

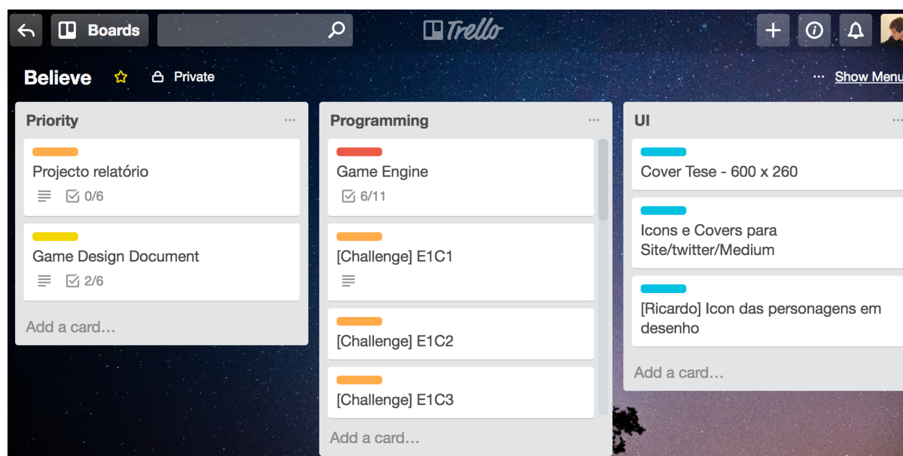

Figura 7.1: Excerto da organização feita no Trello

7.3.2 Controlo de Versões

O controlo de versões é um sistema que grava as alterações a um ficheiro ou conjunto de ficheiros de forma a que mais tarde seja possível revisitar ou alterar versões específicas. Neste capítulo quando é referído controlo de versões, este refere-se a ficheiros de código desenvolvido, apesar de na realidade ser possível ter versões de qualquer tipo de ficheiro no computador.

Este sistema permite reverter um ficheiro, ou até um projecto inteiro, para um estado anterior, comparar alterações ao longo do tempo, verificar quem fez as

últimas submissões que resultaram num problema no produto, quem introduziu uma tarefa e quando, entre outros.

Dentro das várias opções disponíveis para controlo de versões, foi utilizado o sistema Git [25] por ser aberto, gratuito, com diferenciação entre repositórios locais e remotos, bem como, por ser o sistema mais utilizado a nível mundial actualmente.

7.3.2.1 Github

GitHub [27] é uma plataforma online de hospedagem de código-fonte com controlo de versões através de Git. Permite que programadores e utilizadores registados na plataforma contribuam em projectos privados ou de código fonte aberto a partir de qualquer lugar.

7.3.2.2 Organização de repositórios

Uma boa organização do repositório de código é tão importante quanto a utilização do próprio sistema de controlo versões, visto que a desorganização do mesmo pode levar a que diferentes equipas estejam a trabalhar sobre versões erradas do esperado, levando a que o sistema crie mais erros e atritos do que soluções.

Ao longo dos anos foram surgindo várias modelos de como organizar um repositório, sendo o GitFlow [26] o mais conhecido recentemente. Contudo, há várias correntes alternativas que indicam que a utilização desta metodologia não facilita a leitura do histórico para além de que a distinção entre o *branch* (ramo) mestre e um *branch* desenvolvimento é redundante, entre outras observações.

Após uma pequena experiência desta metodologia, foi decidido realizar uma pequena pesquisa por um modelo de organização orientado e testado em aplicações móveis que seja simples e eficaz. Após alguma reflexão, foi adaptado um modelo definido por Artem Zinnatullin [53] e que é descrito da seguinte forma:

- Não existe *branch* mestre, *release* ou `vX.Y`;
- A versão *release* (para distribuição) ou *release candidate* (candidata a distribuição) é marcada com uma etiqueta e não um *branch* ;
- Existirá apenas um *branch* de desenvolvimento seguido da sua versão. Um exemplo poderá ser `dev/1.1.0`;

De seguida é utilizado o seguinte algoritmo:

1. Se a *build* passar os testes de qualidade e se decidir publicar, deve-se colocar uma etiqueta *release* nesse mesmo *commit* `$ git tag v1.1.0`

2. Se a *build* não passar os testes de qualidade e for necessário corrigir algum problema, deve-se fazer as alterações no *branch* `dev/1.1.0` e fazer um novo -rc `$ git tag v1.1.0-rc2` para posteriormente seguir para o ponto 1)

Caso exista um problema numa versão já entregue, faz-se checkout da etiqueta e começa-se a desenvolver sobre essa versão, iterando na versão. Isto é, `dev/1.1.1`. De seguida seguem-se as regras do ponto anterior.

7.3.3 Distribuição e Integração Contínua

A integração contínua (*Continuous Integration*) é uma prática de software em que os membros de uma equipa integram o seu trabalho desenvolvido frequentemente (geralmente diariamente), levando a múltiplas integrações em curtos espaços de tempo.

Cada integração é verificada por uma *build* (compilação) automática incluindo testes para a detecção de erros de integração o mais rápido possível. Muitas equipas de desenvolvimento têm vindo a verificar que esta abordagem leva a uma redução significativa de problemas de integração e que permite desenvolver software de uma forma mais coesiva e rápida.

Para além da integração contínua, é possível também adicionar a distribuíção contínua a um projecto, que implica adicionar nas *builds* definidas, a distribuição para conjuntos de equipas ou clientes, os artefactos resultantes das *builds* de forma automática.

7.3.3.1 CircleCI

CircleCI [5] é uma ferramenta online de automação de *builds* com integração e distribuição contínua de forma as que os programadores se foquem no mais importante, o produto que desenvolvem.

Para utilizar a ferramenta, é necessário associar um repositório de código existente numa das plataforma de repositórios conhecidas (ex:Github) e adicionar

um ficheiro do tipo `.yml` com as configurações necessárias para que a *build* seja invocada automáticamente quando o código for submetido no repositório.

Após a execução de cada *build* é possível verificar o seu resultado e em caso de falha, é necessário realizar as devidas alterações no produto de forma a que a próxima iteracção seja bem sucedida.

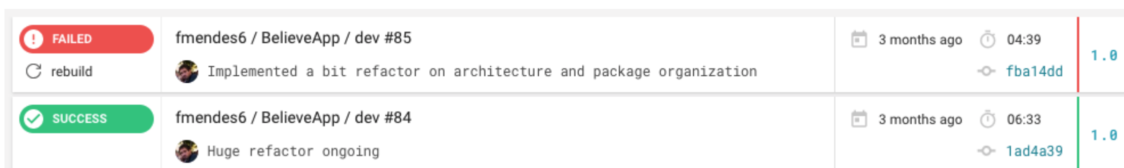

Figura 7.2: Resultado de duas builds no CircleCi

Distribuição contínua em Believe

No caso da aplicação Believe, foram definidos as seguintes configurações de distribuição para os dois grupos de utilizadores que testam a aplicação.

- Por cada vez que houver um *push* para um *branch* com -rc no nome, serão corridos todos os testes desenvolvidos e será gerado um `.apk` (artefato binário android) e distribuído para todos os utilizadores que estejam na lista de testers;
- Por cada vez que houver um *push* para um *branch* com etiqueta de release, serão corridos todos os testes desenvolvidos e será gerado um `.apk` e distribuído para todos os utilizadores que estejam na lista anterior acrescida de mais pessoas, de forma a simular a introdução desta versão no mercado.

Conclusões

Este trabalho centrou-se no desenvolvimento de uma aplicação Android sob a forma de um jogo que tirasse partido da surpresa e *suspense* causado para a criação de uma experiência pouco usual no mundo dos videojogos. Pretendia-se também que o estilo de vida e o contexto pessoal de cada pessoa, tivesse impacto no desenrolar do jogo, tornando difícil de prever os próximos desafios de cada jogador.

A aplicação foi desenvolvida de forma modular, no sentido de garantir a possibilidade de expansão tanto em termos de história como do tipo de desafios e de consequências possíveis de experimentar. Através da separação da lógica de planeamento e agendamento de um desafio da sua componente visual, é possível a modificação e adaptação da experiência de jogo de um desafio, permitindo por exemplo, que para diferentes dispositivos o mesmo desafio possa ter algumas diferenças visuais.

É modular também na fase de planeamento, visto que permite que novas formas de detecção de eventos do sistema operativo e de contexto sejam adicionadas sem grande esforço de implementação porque apenas necessitam de delegar o seu agendamento para o motor de jogo.

A implementação de interfaces para a comunicação entre camadas, deu a liberdade de poder desenvolver independentemente da tecnologia mas mantendo a coesão entre camadas adjacentes.

8.1 Videojogo

A criação de um videojogo levantou dificuldades acrescidas ao projecto pela necessidade de criação de uma história envolvente e que seja desfrutável para o utilizador, não sendo apenas uma aplicação móvel cliente utilitária.

Adicionalmente, a criação de um desafio é tão mais demorado quanto o seu detalhe gráfico e a quantidade de acções a tomar pelo utilizador, pelo que a criação de um capítulo com vários desafios tem um tempo de desenvolvimento associado considerável.

A criação de um desafio contextual representa dificuldades inerentes na medida em que é necessário tentar prever quais as situações mais favoráveis para que o utilizador não seja surpreendido numa altura inoportuna e desinstale a aplicação para não voltar mais a ser perturbado.

8.2 Tecnologias Utilizadas

Em termos de tecnologias de suporte à aplicação, a utilização de bibliotecas não suportadas directamente pela Google mas desenvolvidas pela comunidade e utilizadas em larga escala nas aplicações móveis, permitiu que muitos dos problemas comuns existentes na comunicação com serviços *web* ou persistência de dados, não específicos do negócio de cada aplicação, fossem mitigados e tratados pelos criadores dessas bibliotecas.

A utilização destas bibliotecas é de uma importância muito grande porque permite que o desenvolvimento seja focado no produto e nas suas características únicas em vez de utilizar uma parte significativa do tempo disponível a desenvolvelas de raíz, sendo os casos de Android-Job, Retrofit e Room, os mais cruciais neste ponto.

A utilização da linguagem de programação Kotlin foi também uma hipótese em detrimento do Java, por ser uma linguagem mais concisa, simples e útil para o desenvolvimento de aplicações Android, para além de já ser uma linguagem suportada oficialmente pela Google, contudo, como até à data a experiência com a mesma era limitada, a escolha natural recaíu sobre a linguagem Java.

8.3 Desenvolvimentos Futuros

Com a mecânica do jogo e vários desafios implementados, dar-se-á inicio à restante implementação e adição dos episódios previstos no jogo por meio de actualizações.

Para adicionar novas componentes de imprevisibilidade, a história evoluirá de uma forma a que exista sobreposição de desafios e dicas dos vários capítulos, com a eventual ligação entre personagens e o seu destino.

Deverão também ser acrescentados mais mecanismos influenciadores de tolerância de forma a que todas as acções tomadas pelos utilizadores tenham impacto directo no jogo e que a sua dedicação ao mesmo seja recompensada. Acções como a consulta diária do jornal ou um número alto de horas jogadas poderão ser alguns desses exemplos.

Está também previsto uma maior integração com as ferramentas extra ao jogo, como dicas e porções da história a serem espalhadas pelas redes sociais oficiais do jogo, ou até, a utilização de um *bot* implementado através da plataforma Dialogflow [28] para os utilizadores conversarem via mensagens de texto para obterem respostas a desafios.

Serão também implementadas compras dentro do jogo (*in-app purchases*) de forma a suportar o seu desenvolvimento. Sendo uma das possibilidades, um melhoramento significativo da tolerância de forma a reduzir a dificuldade e frequência dos eventos.

Outra componente que também será implementada será uma avaliação global de quais as personagens que não sobreviveram ao longo da história por parte de cada utilizador. Sendo que cada personagem terá características raciais e culturais diferentes, será interessante verificar se existe uma correlação e empatia entre certos jogadores e certos tipos de personagens, mesmo que se trate de um videojogo.

Por fim, a incorporação dos algoritmos de inteligência artificial como parte central na decisão de desafios e da altura correcta de os lançar. Apesar de não ter sido aprofundado um plano para o fazer, a estratégia passaria pela detecção de certos comportamentos e acções, seguindo uma estratégia de previsão de acções [1].

Por linhas gerais, seriam seguidos os passos seguintes:

1. Começa-se com a selecção e configuração de um conjunto de preferências do utilizador no ecrã inicial em formato de questionário, (ex: horário de

emprego ou de lazer) para realizar algumas previsões iniciais do seu comportamento;

2. Nos dias seguintes far-se-á o rastreio das horas em que a aplicação é utilizada, quais os desafios perdidos ou completados, gravação do contexto e da actividade do utilizador, entre outros, para a actualização dos padrões e previsões iniciais;

3. Ao longo dos dias, o passo anterior seria repetido de forma constante, aperfeiçoando a capacidade de previsão.

Para o rastreio destes padrões seriam utilizadas ferramentas de análise como o Answers [16] ou até o Firebase Predictions [18] que promete fornecer alguns destes dados já tratados.

No entanto, e apesar de ainda ser possível desenvolver muito trabalho neste jogo, por ser tão abrangente e estar em constante evolução, o trabalho feito mostra-se já bastante avançado.

Referências

[1] Shreenath Acharya, Asha Shenoy, Macwin Lewis, and Namrata Desai. Analysis and prediction of application usage in android phones. 2016. URL http://ieeexplore.ieee.org/document/7538346/?reload=truea. (p. 81)

[2] Android. Set or clear default apps. URL https://support.google.com/android/answer/6271667?hl=en. Visitado em 05.10.2017. (p. 39)

[3] Apple. ios. URL https://www.apple.com/pt/ios/ios-11/. Visitado em 05.10.2017. (p. 39)

[4] arturdm. jacoco-android-gradle-plugin. URL https://github.com/arturdm/jacoco-android-gradle-plugin. Visitado em 09.10.2017. (p. 68)

[5] CircleCI. Circleci. URL https://circleci.com. Visitado em 26.05.2017. (p. 77)

[6] Ward Cunningham. Manifesto para o desenvolvimento Ágil de software. URL http://agilemanifesto.org/iso/ptpt/manifesto.html. Visitado em 10.07.2017. (p. 73)

[7] Android Developers. Arquitetura da plataforma, . URL https://developer.android.com/guide/platform/index.html. Visitado em 15.06.2017. (p. 6)

[8] Android Developers. Instant apps, . URL https://developer.android.com/topic/instant-apps/index.html. Visitado em 06.10.2017. (p. 39)

[9] Android Developers. Launch-time performance, . URL https://developer.android.com/topic/performance/launch-time.html. Visitado em 15.09.2017. (pp. xvi e 41)

[10] Android Developers. Fused location provider, . URL https://developer.android.com/training/location/retrieve-current.html. Visitado em 10.06.2017. (p. 52)

[11] Android Developers. Picture-in-picture, . URL https://developer.android.com/guide/topics/ui/picture-in-picture.html. Visitado em 06.10.2017. (p. 39)

[12] Android Developers. Room persistence library, . URL https://developer.android.com/topic/libraries/architecture/room.html. Visitado em 10.06.2017. (p. 59)

[13] Android Developers. Widget, . URL https://developer.android.com/guide/topics/appwidgets/index.html. Visitado em 06.10.2017. (p. 39)

[14] Google Developers. Google awareness api : Antecipate and react, . URL https://developers.google.com/awareness/. Visitado em 10.06.2017. (p. 52)

[15] Evernote. Android-job. URL https://github.com/evernote/android-job. Visitado em 10.06.2017. (p. 54)

[16] Fabric. Answers. URL https://fabric.io/kits/android/answers. Visitado em 07.10.2017. (p. 82)

[17] FindBugs. Findbugs. URL http://findbugs.sourceforge.net. Visitado em 09.10.2017. (p. 69)

[18] Firebase. Firebase predictions. URL https://firebase.google.com/docs/predictions/. Visitado em 07.10.2017. (p. 82)

[19] Martin Fowler. Inversion of control containers and the dependency injection pattern, . URL https://martinfowler.com/articles/injection.html. Visitado em 15.09.2017. (p. 42)

[20] Martin Fowler. Testcoverage, . URL https://martinfowler.com/bliki/TestCoverage.html. Visitado em 10.10.2017. (p. 68)

[21] Martin Fowler. Testpyramid, . URL https://martinfowler.com/bliki/TestPyramid.html. Visitado em 10.10.2017. (pp. xvi, 65, e 66)

[22] Kaigan Games. Sara is missing. URL https://saraismissing.itch.io/sim. Visitado em 6.05.2017. (p. 9)

[23] Gartner.com. Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016, February 2017. URL http://www.gartner.com/newsroom/id/3609817. Visitado em 30.05.2017. (p. 6)

[24] Gideros. Gideros. URL http://giderosmobile.com. Visitado em 16.06.2017. (p. 8)

[25] Git. Git. URL https://git-scm.com. Visitado em 22.02.2017. (p. 76)

[26] GitFlow. Gitflow. URL http://nvie.com/posts/a-successful-git-branching-model/. Visitado em 22.02.2017. (p. 76)

[27] Github. Github. URL https://github.com. Visitado em 22.02.2017. (p. 76)

[28] Google. Dialogflow, . URL https://dialogflow.com. Visitado em 20.10.2017. (p. 81)

[29] Google. Dagger and android, . URL https://google.github.io/dagger/android.html. Visitado em 15.09.2017. (p. 44)

[30] Google. Espresso, . URL https://developer.android.com/training/testing/espresso/index.html. Visitado em 10.10.2017. (p. 68)

[31] Google. Google-java-format, . URL https://github.com/google/google-java-format. Visitado em 09.10.2017. (p. 70)

[32] Google. Ui automator, . URL https://developer.android.com/training/testing/ui-automator.html. Visitado em 10.10.2017. (p. 68)

[33] Gradle. Build flavors, . URL https://developer.android.com/studio/build/build-variants.html. Visitado em 10.10.2017. (p. 70)

[34] Gradle. Build flavors, . URL https://developer.android.com/studio/build/build-variants.html. Visitado em 10.10.2017. (p. 70)

[35] HaxeFlixel. Haxeflixel. URL http://haxeflixel.com. Visitado em 16.06.2017. (p. 8)

[36] Raph Koster. *Theory of Fun for Game Design*. O'Reilly Media, 2013. (pp. vii e ix)

[37] Annika Waern Markus Montola, Jaakko Stenros. *Pervasive Games: Theory and Design (Morgan Kaufmann Game Design Books)*. Morgan Kaufmann, 2009. (p. 8)

[38] Maven. Junit4. URL http://junit.org/junit4/. Visitado em 10.10.2017. (p. 66)

[39] Microsoft. Reactive extensions. URL https://msdn.microsoft.com/en-us/library/hh242985(v=vs.103).aspx. Visitado em 05.07.2017. (p. 40)

[40] Mockito. Mockito. URL http://site.mockito.org. Visitado em 10.10.2017. (p. 66)

[41] Niantic. Pokemon go. URL http://www.pokemongo.com. Visitado em 3.05.2017. (p. 9)

[42] Jussi Parikka and Jaakko Suominen. Victorian snakes? towards a cultural history of mobile games and the experience of movement. *Game Studies*, 6(1), Sep 2006. URL http://gamestudies.org/0601/articles/parikka_suominen. (p. 5)

[43] Android Studio Project. Android lint. URL http://tools.android.com/tips/lint. Visitado em 09.10.2017. (p. 69)

[44] G. A. Giraldi R. Silva, J. C. Oliveira. Introduction to augmented reality. Technical report, National Laboratory for Scientific Computation, 2003. URL http://lncc.br/~jauvane/papers/RelatorioTecnicoLNCC-2503.pdf. Visitado em 22.04.2017. (p. 8)

[45] ReactiveX. Rxjava: Reactive extensions for the jvm. URL https://github.com/ReactiveX/RxJava. Visitado em 10.06.2017. (p. 40)

[46] Roboletric. Roboletric. URL http://robolectric.org. Visitado em 10.10.2017. (p. 66)

[47] RobotiumTech. Robotium. URL https://github.com/RobotiumTech/robotium. Visitado em 10.10.2017. (p. 68)

[48] Olli Sotamaa. All the world's a botfighter stage: Notes on location-based multi-user gaming. Technical report, University of Tampere, Hypermedia Laboratory, 2002. URL http://www.digra.org/wp-content/uploads/digital-library/05164.14477.pdf. Visitado em 18.04.2017. (p. 6)

[49] Square. Retrofit. URL http://square.github.io/retrofit/. Visitado em 10.06.2017. (p. 59)

[50] Trello. Trello. URL http://www.trello.com. Visitado em 15.02.2017. (p. 75)

[51] Unity. Sobre nós. URL https://unity3d.com/pt/public-relations. Visitado em 16.06.2017. (p. 8)

[52] YoyoGames. Gamemaker studio. URL https://www.yoyogames.com/gamemaker/features. Visitado em 16.06.2017. (p. 8)

[53] Artem Zinnatullin. Git ftfy: branching model + ci [continuation of "git flow considered harmful"]. URL https://artemzin.com/blog/git-ftfy-branching-model-continuation-of-git-flow-considered-harmful/. Visitado em 11.04.2017. (p. 76)

Divulgação e Promoção

Apesar de estar fora do âmbito do projecto, decidiu-se definir uma estratégia de marketing e promoção do projecto de forma a tentar entrar e ganhar tracção no mercado das aplicações móveis.

Para isso foram criados vários canais públicos online do jogo, tendo em vista a sua disseminação em todas as vertentes, desde a promoção do produto em si, à interacção e extensão do jogo e das personagens fora da aplicação, até à demonstração e documentação do seu desenvolvimento.

A.1 Twitter

Existe um canal oficial no twitter que será utilizado para lançar várias mensagens enigmáticas e dúbias, como extensão ao jogo e onde eventualmente poderão estar a resposta a alguns desafios. Está também previsto que sirva de ferramenta de comunicação com alguns dos utilizadores, seja por via manual ou de forma automática através da api.ai.

A conta está disponível para ser seguida em *twitter.com/blvgame*.

A.2 Medium

A conta oficial da plataforma medium.com servirá como canal com várias publicações do ponto de vista dos seus criadores, com a descrição sobre o desenvolvimento do jogo e os desafios que foram ultrapassados.

Servirá maioritariamente para dar uma amostra mais técnica sobre todo o processo de desenvolvimento e poderá ser consultada em *medium.com/believe-game*.

A.3 Publicação e Promoção

A aplicação estará disponível na Play Store da Google de forma a conseguir distribuir num mercado global e de forma eficiente, com distribuições faseadas a cada nova versão.

Será criado um texto de promoção com *screenshots* para publicar no mercado de aplicações e reutilizado em fóruns de especialidade tecnológica, como a *Zwame.com* e em portais de noticias como o *ProductHunt.com* (internacional) ou o *pplware.com* (nacional).

Paralelamente, serão feitas várias apresentações em conferências com tópicos e características técnicas do jogo, por forma a ajudar a divulgar a sua existência.