INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

Jorge Amadeu Alves Pereira da Silva

Geração de código MVC para Android, a partir de Modelos em XML

Mestrado em Engenharia de Software

Trabalho de Projeto efetuado sob a orientação do
Doutor António Miguel Cruz

Fevereiro de 2017

RESUMO

Este trabalho explora a transformação de modelos de domínio e de casos de uso, descritos em XML, em código fonte de aplicações para a plataforma Android. A riqueza semântica do UML, e o rigor imposto por algumas regras convencionadas, permitem gerar modelos a partir dos quais é possível a geração de aplicações. Contudo, estas não podem ser consideradas como aplicações finais, pois os diagramas utilizados, ainda que complementados por convenções, não conseguem captar todas as restrições necessárias para à geração de uma aplicação final. Desta forma, o processo apresentado visa sobretudo a produção de protótipos funcionais. Estes podem ser úteis em fases de levantamento de requisitos, uma vez que ao serem gerados em questão de minutos, permitem uma exploração rápida das funcionalidades pretendidas numa futura aplicação, e podem mesmo servir de embrião para a produção da aplicação final, carecendo, neste caso, de desenvolvimentos posteriores. O processo de geração estabelecido neste projeto de mestrado permite a persistência de dados numa base de dados SQLite. Esta solução pode ser restritiva, pois isola a aplicação móvel em cada dispositivo onde é instalada. São apresentadas propostas de trabalho futuro que visam resolver este problema, adequando a aplicação gerada a um maior número de cenários de utilização de uma aplicação móvel.

Fevereiro de 2017

ABSTRACT

In this work, the possibility of transformation of domain models and use case models, expressed in XML, into application code to the android platform is explored. The semantic richness and rigor of the UML allows for the creation of models from which the generation process is possible. But those are not complete, because the diagrams used in the modelling process don't allow capturing all the needed restrictions. In this context, the proposed process only allows the generation of functional prototypes. These can be useful during requirements gathering and elicitation phase, because they can be produced in a few minutes and allow the fast exploration of multiple solutions and functionalities. In the present moment, the generation process only makes data persistence in SQLite databases. This solution may not be useful for many scenarios of mobile application use, because it isolates the mobile application in each device it is installed on. Future work proposals are presented to mitigate this problem, making the generated application fit for a greater number of a mobile application’s usage scenarios.

February 2017

CONTEÚDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

INDICE DE LISTAGENS

1. INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Os dispositivos móveis e em particular os smartphones registaram um franco crescimento nos últimos anos, tornando-se o equipamento com capacidade computacional de mais fácil acesso a um utilizador. Neste quadro o desenvolvimento de aplicações para este mercado aumentou significativamente de importância. Além disso, a presença quase ubíqua destes equipamentos cria a oportunidade para migrar algumas aplicações tradicionalmente presentes em sistemas desktop.

O crescimento da presença de dispositivos móveis tem associada alguma fragmentação resultante da existência de plataformas distintas como o Android, o iOS, o WindowsPhone além de outras menos representativas. Cada uma destas plataformas tem o seu ambiente de desenvolvimento e as suas ferramentas. Por outro lado, numa tentativa de conquista de mercado, procura-se a diferenciação pela introdução de novas funcionalidades, o que gera um ambiente de rápida mutação, por vezes com alterações significativas. Entre estas encontra-se a recente introdução do swift em substituição do objective-C como linguagem de programação para iOS, ou novas formas de interação resultantes da capacidade para o ecrã do dispositivo para detetar diferentes níveis de pressão. Todos estes fatores dificultam a migração de aplicações entre as diferentes plataformas e aumentam custos de produção por poderem exigir equipas de desenvolvimento distintas e especializadas.

Noutra frente, o desenvolvimento de software guiado por modelos permite a criação de processos de desenvolvimento em que, num último passo, é feita a geração automática de código, para plataformas específicas, a partir de um único modelo abstrato do sistema a desenvolver. Este processo envolve vários passos intercalados de modelação, por parte do engenheiro de software, e de transformação de modelos, envolvendo ferramentas automáticas ou semi-automáticas, culminando na geração de código para cada plataforma específica.

Neste trabalho pretende-se apresentar um processo e ferramentas para desenvolvimento guiado por modelos de aplicações para plataformas móveis.

1.2 MOTIVAÇÃO

O principal problema que se pretende analisar é o da possibilidade de gerar uma aplicação móvel partindo de uma descrição do sistema através de modelos de domínio e de casos de uso.

A utilização de modelos como ponto de partida para a geração dos artefactos específicos da plataforma pode ter o potencial de reduzir a necessidade de um conhecimento muito

especializado de uma dada plataforma, para a qual se está a desenvolver. Dessa forma podem-se reduzir custos de formação e atualização da equipa de desenvolvimento. Com a possibilidade de geração de código de uma forma automatizada, pode-se obter uma diminuição do tempo de desenvolvimento com a consequente diminuição de custos.
Uma parte significativa das aplicações empresariais disponíveis para plataformas móveis envolve o preenchimento de formulários, a listagem de itens ou a consulta de um detalhe. A estas tarefas está normalmente associada a criação, edição, leitura, eliminação e listagem de registos armazenados localmente ou disponibilizados através de serviços. Estas correspondem às operações CRUD às quais se associa uma operação de listagem que permite a pesquisa num conjunto de registos. Estes tipos de operações podem facilmente ser identificados em algumas das aplicações mais usadas, como por exemplo um cliente de email.
O projeto desenvolvido explora a possibilidade de gerar sequências de componentes da *User Interface* que suportem operações CRUD e de Listagem determinadas pela sequência de casos de uso atribuídos a um Actor. As entidades sobre as quais são feitas essas operações são definidas num modelo de domínio e associadas aos casos de uso apropriados.

1.3 Objetivos

Os objetivos do projeto desenvolvido são:

- Explorar a possibilidade de geração de aplicações móveis para a plataforma android a partir de modelos de domínio e de casos de uso expressos em XML.
- Explorar simplificações, que restringindo os casos de uso, facilitem a sua descrição e padronizem as operações possíveis, com vista a uma geração automática de código.
- Explorar o significado dos diagramas de casos de uso para a geração de layouts e interfaces de utilizador.
- Identificar limitações dos modelos que possam ter impacto na aplicação gerada.
- Explorar a possibilidade de utilização do modelo de domínio com vista à geração de soluções de persistência de dados

1.4 Estrutura do documento

O remanescente deste relatório de projeto de mestrado encontra-se estruturado da seguinte forma:

- No capítulo 2 são apresentadas algumas definições e conceitos úteis para a leitura deste relatório, nomeadamente as abordagens ao desenvolvimento de aplicações, as características da plataforma android e o seu padrão de desenvolvimento.
- No capítulo 3, explora-se o estado da arte, cobrindo abordagens que se centram na produção de *User Interfaces*, no desenvolvimento de serviços que possam ser

consumidos por plataformas móveis e no desenvolvimento de aplicações completas. Procura-se, ainda, fazer uma comparação entre as soluções apresentadas nas diferentes abordagens.

- No capítulo 4, aborda-se o processo de geração de aplicações baseadas em formulários a partir de modelos descritos em XML
- No capítulo 5, abordam-se as vantagens e limitações do processo apresentado desde a criação dos modelos até à geração das aplicações.
- No capitulo 6, apontam-se possibilidades de evolução futura deste trabalho

2. Definições e Conceitos

2.1 Introdução

O desenvolvimento de uma aplicação é uma tarefa complexa, que pode merecer diferentes tipos de abordagens. A equipa de desenvolvimento pode concentrar-se no desenvolvimento do seu código. Este constitui praticamente o único artefacto produzido e é o repositório da informação mais recente e atualizada do sistema, mas cuja leitura e análise pode ser complexa no contexto da manutenção do sistema. A utilização de modelos eleva o nível de abstração do conhecimento do sistema, podendo ser um facilitador do seu entendimento. Todavia, utilidade dos modelos não se restringe a este aspeto, a partir deles pode-se obter um sistema funcional, quer gerando o código correspondente a uma plataforma alvo ou interpretando-os em tempo de execução. Estas possibilidades exigem um maior rigor e menor ambiguidade dos modelos produzidos, quando comparados com os modelos que têm apenas por objetivo facilitar a comunicação entre membros de uma equipa. Além deste aspeto, é necessário definir as regras de transformação ou de interpretação dos modelos que conduzam a sistemas com o comportamento previsto. Neste capitulo explora-se o desenvolvimento orientado por modelos, a semântica dos modelos de domínio e de casos de uso e as características de uma aplicação para a plataforma android.

2.2 Desenvolvimento guiado por modelos

A capacidade para um ser humano lidar com a complexidade é limitada. A criação de modelos é uma forma de ultrapassar essas limitações. Os modelos permitem a criação de abstrações sobre um dado problema. Desta forma destaca-se os aspetos essenciais de uma dada visão sobre o espaço do problema e oculta-se detalhes de menor relevância. Com estas propriedades o modelo facilita a análise do problema, focando a atenção nos aspetos relevantes de uma dada visão e reduzindo-se a complexidade a níveis compatíveis com a capacidade de raciocínio humanas.

Neste contexto o desenvolvimento de uma aplicação pode ser entendido como um refinamento progressivo de modelos. Durante esse processo é acrescentado detalhe, reduz-se o nível de abstração tomando os modelos uma forma cada vez mais próxima da implementação do sistema.

Ao considerar níveis de abstração distintos é comum agrupar os modelos em diferentes categorias. Nas primeiras iterações de modelação/desenvolvimento são produzidos modelos independentes da computação (*computation independent model* – CIM), os quais definem funcionalidade de forma declarativa e independente de algoritmos específicos que possam levar a cabo essa funcionalidade (Díaz *et al.*, 2015), e/ou modelos independentes da plataforma (*platform independent model* – PIM), que modelam o sistema de forma independente de toda e qualquer plataforma tecnológica de

implementação. A redução do nível de abstração tornará os modelos dependentes das soluções tecnológicas escolhidas pelo que passarão a ser considerados Modelos específicos da plataforma (platform specific model – PSM). Este processo, denominado pela OMG (*Object Management Group*) como Model-Driven Architecture (Truyean, 2006), e mais genericamente como desenvolvimento guiado por modelos, baseia-se na criação de modelos a um nível de abstração elevado, e sua transformação em modelos num nível de abstração mais reduzido. Para isso é necessária uma linguagem adequada com algum grau de formalidade. A *Unified Modeling Language* (UML) foi definida pela OMG como uma linguagem para modelação de sistemas, sendo hoje a linguagem mais utilizada para modelar sistemas de software (OMG, 2015).

2.2.1. A MODELAÇÃO DE UM DOMÍNIO E OS DIAGRAMAS DE CLASSES

A utilização do UML permite desenvolver modelos, que no contexto de um domínio específico, capturam a sua estrutura, permitem a análise do problema e posterior evolução para a estrutura da solução. Uma dessas ferramentas é o diagrama de classes que permite a elaboração e refinamento desses modelos. Através de cada classe é possível especificar uma classificação para um conjunto de objetos e definir as suas propriedades e os seus comportamentos (OMG,2015). As propriedades de uma classe correspondem aos seus atributos, que podem corresponder a terminações de associações binárias (OMG,2015). Estas correspondem a valores que cada objeto da classe tem que possuir e com os quais é instanciado, quer eles sejam explicitamente definidos ou determinados (OMG,2015). Os comportamentos são definidos através de operações que podem ser invocadas nas instâncias da classe. Uma operação especifica o nome, o tipo, os parâmetros e as condições associadas à evocação (OMG, 2015). Neste trabalho o leque de operações que são realizáveis sobre as entidades de um modelo foi limitado a um conjunto padrão cuja especificação é feita na definição dos casos de uso que são associados a cada uma das classes do modelo.

Uma associação estabelece uma relação entre um conjunto de objetos (OMG,2015), que no contexto de um diagrama de classes, são definidos através de classes. Através de uma associação podem-se definir com rigor aquelas propriedades que resultam de relacionamentos entre classes.

Fazendo uso deste instrumento pode-se capturar o conhecimento sobre um domínio, identificando as suas entidades, as propriedades das mesmas, bem como as suas relações.

2.2.2. ESPECIFICAÇÃO DE COMPORTAMENTOS E OS DIAGRAMAS DE CASOS DE USO

As ações que é possível realizar num sistema podem ser modeladas através de diagramas de casos de uso. Nestes diagramas um caso de uso representa um comportamento ou interação com o sistema do qual se obtém um resultado observável e de valor para o Actor ou outras partes interessadas(OMG,2015). Neste contexto um Actor modela o papel desempenhado por uma entidade quando interage com os casos de uso, não tendo

obrigatoriamente correspondência com uma entidade física, pois a mesma entidade pode desempenhar papeis diferentes em contextos diferentes ou o mesmo papel pode estar atribuído a entidades distintas (OMG,2015).

A funcionalidade associada a um caso de uso tem que ser especificada, podendo tal envolver a definição de interações, atividades, pré-condições e pós-condições de uma forma rigorosa ou em linguagem natural (OMG,2015). No contexto deste trabalho o âmbito dos casos de uso está limitado às ações de criação (*create*), leitura(*read*), atualização (*update*), eliminação (*delete*) e listagem (*list*) de registos, isto é, às operações CRUD e de listagem, pelo que a especificação dos mesmos foi muito simplificada.

Os casos de uso podem pertencer a uma classe e dessa forma representam o tema a que estão associados (OMG,2015). Neste trabalho os casos de usos podem estar associados a uma ou duas classes. Quando associados a uma classe representam operações CRUD e de listagem sobre uma entidade do domínio. Quando associados a duas classes essas operações fazem-se sobre entidades relacionadas em que uma é um detalhe da outra.

Os casos de uso podem ser associados através de relacionamentos que a especificação do UML dá diferentes significados. Um relacionamento de ***extend*** entre dois casos de uso é um relacionamento dirigido que liga um caso de uso de origem a um caso de uso de destino (OMG,2015). Através desse relacionamento a funcionalidade associada ao segundo é ampliada através da opção de execução da funcionalidade associada ao primeiro (OMG,2015). Os dois casos de uso são definidos de forma atómica e independente e a realização de um caso de uso não implica a obrigatoriedade de realização do caso de uso que amplia. Essa independência e atomicidade permite a associação do caso de uso que ampliam funcionalidades a diferentes casos de uso cuja funcionalidade é ampliada (OMG,2015). No contexto deste trabalho um relacionamento de ***extend*** pode ser utilizado para encadear operações CRUD e listagem que são opcionais para o Actor, como por exemplo visualizar uma listagem de registos e a partir dela ter a possibilidade de realizar uma das operações CRUD, ou visualizar um determinado registo e a partir dele ter a possibilidade de aceder a uma listagem de registos de uma entidade relacionada. A relação ***extend*** prevê a existência de pontos de extensão, estes correspondem ao momento da execução da funcionalidade em que esta pode ser ampliada pelo caso de uso associado (OMG,2015). Este conceito poderia ser explorado para definir quando uma determinada operação CRUD e de listagem pode ser lançada a partir de uma prévia. Por exemplo em que condições durante a visualização de um registo é possível ampliar essa funcionalidade para aceder a uma lista de registos de uma entidade relacionada. No presente trabalho os casos de uso associados através de relacionamentos ***extend*** estão imediatamente disponíveis, ou seja, a condição correspondente ao ponto de extensão é sempre avaliada como verdadeira. Este comportamento permite, por exemplo pedir a listagem de entidades relacionadas, mesmo quando estas ainda não foram criadas. Nestes casos usa-se a funcionalidades dos layouts da plataforma Android para fornecer dar um feedback ao Actor.

A especificação UML prevê um relacionamento ***includes*** entre casos de uso. Este é também um relacionamento dirigido em que o comportamento definido num caso de uso é incluído no comportamento de outro caso de uso, que é através dele obrigatoriamente ampliado (OMG,2015). Este relacionamento é usado quando existem porções de comportamento que surgem associados a casos de uso diferentes e que desta forma podem ser segregados para que possam ser reutilizados (OMG,2015). O relacionamento ***includes*** permite também a composição hierárquica de casos de uso em que a realização de um caso de uso só está concluída depois de realizados e concluídos todos os casos de uso da hierarquia, em que cada um destes, uma vez iniciado só retornam o controlo ao caso de uso ampliado quando se verificar a sua conclusão bem como a de todos os que a ele estiverem associados por este tipo de relacionamento (OMG,2015). Este tipo de relacionamento pode ser usado para modelar comportamentos complexos compostos por vários passos cuja conclusão final depende do resultado dos passos intermédios. No atual estado de desenvolvimento deste trabalho não se exploraram situações de maior complexidade do que operações CRUD e de listagem atómicas pelo que não é explorada a semântica do relacionamento ***includes*** entre casos de uso.

2.2.3. Transformação de modelos como forma de desenvolvimento

O desenvolvimento orientado por modelos (MDD) prescreve a construção de modelos que são refinados ou transformados reduzindo o nível de abstração desde o modelo independente da plataforma (PIM) até a um modelo específico da plataforma (PSM) que finalmente é convertido em código executável (Silva, Paiva, & Cruz, 2016) (figura 1)

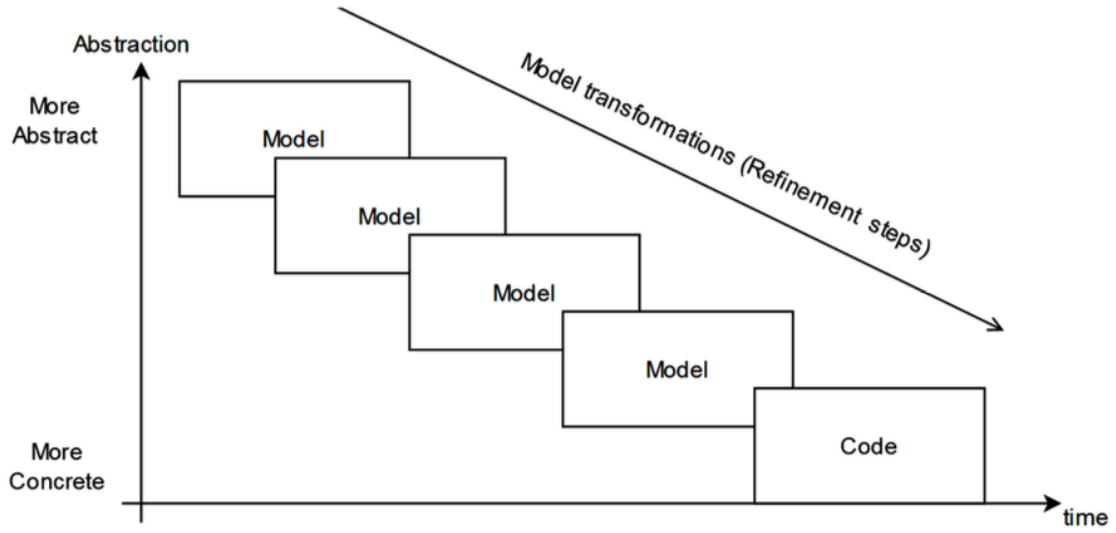

Figura 1: Transformações de modelos através de um processo de redução do nível de abstração.

Os primeiros modelos produzidos são normalmente independentes da plataforma (PIM) e podem ser constituídos por diversos artefactos, que permitem diferentes perspetivas

sobre o sistema. O PIM pode constituir o input de um processo de transformação que origine um novo modelo (model-to-model transformation - M2M). Uma vez definido, esse processo, permite transformar um PIM num modelo de um nível de abstração diferente como um modelo específico de uma plataforma (PSM) (Cruz, 2010; Silva, Paiva, & Cruz, 2016). O código executável numa plataforma pode resultar de um processo automático de geração a partir do PSM através de um processo de transformação do modelo para código (M2C – model-to-code) (Cruz, 2010; Silva, Paiva, & Cruz, 2016). Um processo equivalente, de transformação de modelo para texto (M2T – model-to-text) pode ser usado para gerar documentação (Cruz, 2010; Silva, Paiva, & Cruz, 2016). Considerando que o código é uma forma de texto as designações M2C e M2T podem ser tidas como equivalentes (Cruz, 2010).

Esta metodologia permite preservar sob a forma de modelos o conhecimento sobre um domínio ou sistema. Estes mantêm a sua utilidade pois podem ser objeto de adaptação e transformação aquando de alterações no domínio ou nas tecnologias (Petrov & Buchmann, 2008).

Ao produzir modelos que vão ser objeto de transformações automatizadas é necessário garantir que aqueles são rigorosos, completos e não ambíguos (Cruz, 2010) se o resultado final pretendido for a produção de uma aplicação final. Todavia modelos incompletos e com algum grau de ambiguidade podem ser usados para gerar protótipos com um certo nível de funcionalidade, que permitam aumentar o conhecimento do problema, a exploração de diferentes soluções ou melhorar os modelos existentes.

Os processos de transformação M2M e M2T têm objetivos diferentes. Os primeiros são adequados ao refinamento de modelos, ao seu refactoring, à engenharia reversa, à aplicação de padrões, à geração de novas perspetivas de um modelo (PIM-to-PIM), à transformação de PIM para PSM ou a qualquer atividade que possa ser automatizada sobre modelos. Estas técnicas podem compreender (Jézéquel, 2005; Cruz, 2010; Silva, Paiva, & Cruz, 2016):

- Linguagens genéricas de programação (como Java ou C#)
- Ferramentas genéricas de transformação (como o XSLT)
- Ferramentas CASE e as suas linguagens de scripting (como o Rose)
- Ferramentas dedicadas de transformação de modelos (como ATL & MTL (INRIA), QVTEclipse)
- Ferramentas de Meta-modelação (como MetaCase e Kermeta)

As transformações M2T destinam-se à geração de código ou documentação e podem ser classificadas em (Jézéquel, 2005; Cruz, 2010; Silva, Paiva, & Cruz, 2016):

- **Abordagens baseadas em visitas (visitor-based)**: em que a representação do modelo é atravessada e o código é gerado para um stream de texto.
- **Abordagens baseadas em Templates (template-based)**: baseados na construção de Templates, que consistem no texto alvo com meta-código para acesso à informação do modelo fonte

A aplicação de técnicas de transformação no processo de desenvolvimento procura resolver os seguintes problemas (Cruz, 2010; Warmer et al., 2003; Silva & Videira, 2008):

O problema da produtividade: quando as principais atividades produtivas no desenvolvimento de software são a produção de código e os testes, as equipas tendem a despender pouco tempo em atividades de modelação e de documentação dos sistemas. O problema surge quando as equipas são desmanteladas ou novos membros assumem responsabilidades na manutenção do software, corrigindo erros ou melhorando funcionalidades. O contributo do MDD resulta da transferência da maioria das atividades para a modelação e para processos automáticos de transformação M2M e M2T. Os modelos passam a ser o principal produto e tendo um nível mais elevado de abstração são de mais fácil entendimento pelos novos membros e contribuindo para reduzir o esforço e custo do processo de manutenção.

O problema da portabilidade: a rápida renovação tecnológica acaba por impor um esforço de migração para as novas plataformas e tecnologias. O MDD tende a preservar o valor do investimento anterior ao basear a produção de código em processos automáticos de transformação M2C.

O problema da interoperabilidade: as organizações dependem cada vez mais de vários sistemas de software para a qual a interoperabilidade é fundamental. Neste contexto a definição de componente pode ocorrer ao nível dos modelos interoperacionais. Ao gerar os sistemas a partir dos respetivos modelos a interoperacionalidade ao nível dos modelos pode ser transportada para o nível do código.

O problema da manutenção e documentação: A produção e manutenção da documentação envolve esforço e tende a ser uma tarefa negligenciada. Quando tal acontece a base documental fica desatualizada o que provoca maiores dificuldades e custos na manutenção do software. A produção automatizada de documentação a partir do código não garante a atualização da documentação de alto nível. A geração de automatizada de documentação a partir dos modelos é a forma que o MDD propõe para a sua resolução.

2.3 A PLATAFORMA ANDROID – ESTRUTURA E ORGANIZAÇÃO DAS SUAS APLICAÇÕES.

A plataforma Android foi desenvolvida para dar suporte a plataformas móveis como smartphones e tablets. Inclui um sistema operativo baseado no Linux, sobre o qual corre um conjunto de bibliotecas nativas e o *runtime Android*. Sobre esta camada opera a Application Framework do qual depende a camada correspondente às aplicações com que o utilizador interage (Android API Guide, 2017).

As aplicações em Android existem num espaço controlado que garante segurança e o controlo dos recursos da plataforma. Assim a aplicação existe em isolamento das restantes aplicações pois é executada na sua própria máquina virtual e recebe o seu próprio utilizador e processo. Ao gerir a execução da aplicação o sistema garante o

controlo dos recursos e a sua disponibilização adequada às aplicações de forma a proporcionar a melhor experiência de utilização. Esta propriedade implica que a aplicação seja desenhada de forma a ceder o controlo da sua execução e a recupera-lo conforme as necessidades imposta pelo sistema. Assim os componentes de uma aplicação têm ciclos de vida cujas etapas permitem ao programador conjugar as necessidades da sua aplicação com o controlo da sua criação e destruição pelo sistema (Andoid API Guide, 2017).
Uma aplicação Android pode ser constituída por quatro tipos diferentes de componentes. Eles podem ser Activities, Services, Content Providers ou Broadcast Receivers que têm diferentes propósitos e ciclos de vida no contexto da aplicação (Andoid API Guide, 2017).

2.3.1. COMPONENTES DE UMA APLICAÇÃO ANDROID

O espaço de interação é representado por uma ou mais Activity. Através deste componente o utilizador ganha acesso aos elementos que lhe permitem realizar uma tarefa no contexto da aplicação. O encadeamento de tarefas é possível através do encadeamento de diferentes Activities, o que possibilita a realização de operações complexas.
Um aspeto relevante de uma Activity é que estes espaços de interação podem ser isolados e independentes uns dos outros. Assim é possível, desde que a aplicação o permita, que uma outra aplicação crie Activities que não lhe pertencem para que uma tarefa seja executada. Por exemplo é possível uma aplicação que um comercial no terreno usa para colocar ordens de encomenda, aceda a uma Activity da aplicação de email para que seja possível compor uma mensagem de confirmação para um cliente (Android API Guide 2017).
Através das Activities o sistema consegue de forma eficiente manter a interação com o utilizador. A Activity que representa o ecrã que o utilizador manipula é prioritária e o seu processo é mantido. No entanto as Activities que o utilizador utilizou anteriormente têm uma elevada probabilidade de voltarem a ser necessárias, pois o utilizador pode voltar a elas após completar a tarefa em mãos ou fazendo *back*. Procurando garantir a melhor experiência de utilização esses processos são mantidos dentro das limitações do dispositivo concreto que está a ser utilizado. Não sendo tal possível, as Activities têm que ter meios para serem recriadas com o estado anterior recuperado, quando o utilizador a elas regressa depois de o sistema ter eliminado os processos que as suportavam (Andorid API Guide, 2017).
Um Service corresponde à possibilidade de uma aplicação se manter em funcionamento em background, por exemplo para realizar uma operação demorada que poderia comprometer a experiência de utilização ou aprisionar o sistema. Um Service não fornece uma interface de utilizador, mas pode ser criado ou usado por uma Activity.
Tal como uma Activity o ciclo de vida de um Service é gerido pelo sistema e tem de disponibilizar formas que permitam estabelecer prioridade de execução e meios de

destruição e recriação de forma controlada e adequada à realização com eficiência da operação por que é responsável (Android API Guide, 2017).

Um Broadcast Receiver é um componente que permite à aplicação receber eventos do sistema ou de outras aplicações, mesmo quando a aplicação não está em funcionamento. Desta forma uma aplicação pode ser notificada de um dado evento, como por exemplo a conclusão de um download, ou que é o momento para realizar uma tarefa agendada (Android API Guide, 2017).

Um Content Provider faz a gestão de um conjunto de dados partilhados. Através dele e com a sua permissão, várias aplicações podem usar um mesmo conjunto de dados. No sistema Android existe um *content provider* para os contactos do utilizador. Assim esta informação fica disponível para todas as aplicações que tenham permissão de acesso (Android API Guide, 2017).

A ativação dos componentes Activity, Service e Broadcast Receivers é feita através de mensagens assíncronas designadas de Intent. Um Intent é usado para associar componentes no sentido de obter a funcionalidade desejada através da colaboração de componentes que podem pertencer a aplicações diferentes. Estes podem ser explícitos quando usados para ativar um componente em particular ou implícitos quando é apenas especificado o tipo de componente a usar (Android API Guide, 2017).

Os content providers são ativados de forma distinta, fazendo uso de um pedido sob a forma de um ContentResolver. Este consegue gerir a transação com o content provider evitando que o componente que fez o pedido tenha que lidar diretamente com o content provider.

2.3.2. O ANDROID MANIFEST

O android manifest é a forma de comunicar ao sistema quais os componentes da aplicação, pelo que eles têm de ser declarados neste ficheiro. Além desta função o manifesto serve para identificar as permissões que a aplicação requer, o nível mínimo da API que a aplicação suporta, requisitos de hardware e software necessários ao funcionamento da aplicação e bibliotecas que a aplicação utiliza (Android API Guide, 2017).

De uma forma geral só os componentes declarados no manifest estão visíveis para o sistema android e podem ser lançados. Uma exceção a esta regra existe para os broadcast receivers que podem ser criados e registados de forma dinâmica no código.

Como referido, a ativação de activities, services e broadcast receivers é feita através de um intent explicito ou implícito. A capacidade para um dado componente de uma aplicação responder a um intent implícito lançado por outra aplicação tem que ser definida no manifest. Quando esta capacidade é definida o sistema possibilitará ao utilizador a seleção do componente com o qual ele pretende executar a tarefa que iniciou na aplicação que lançou o intent.

2.3.3. Recursos da Aplicação

Uma aplicação depende de mais do que do código java para a sua execução. Normalmente depende de imagens, como por exemplo ícones, e de layouts da *user interface*, definidos em ficheiros XML. Todos esses recursos são identificados de forma unívoca num projeto android. Os recursos são disponibilizados de forma independente do código permitindo que a mesma base de código possa ser usada em dispositivos com características diferentes, usando para tal, diferentes versões de um dado recurso, como por exemplo um layout. Para possibilitar ao sistema android a escolha do conjunto de recursos adequados a um dado dispositivo, estes são estruturados num conjunto de pastas com designações que permitem identificar o recurso e a configuração para que são adequados (Android API Guide, 2017).

2.3.4. O ciclo de vida de uma Activity ou o casamento entre as necessidades do utilizador e o controlo das activities pelo sistema

Uma activity é uma unidade de interação com o utilizador. Ao realizar uma tarefa no sistema o utilizador pode navegar ao longo de diferentes activities de uma ou várias aplicações distintas. Esta navegação não é obrigatoriamente de sentido único. O utilizador pode voltar a uma activity já visitada ao regressar depois de realizada uma dada interação. Por exemplo pode voltar a uma activity que representa uma nota de encomenda, depois de ter escrito e enviado um mail de confirmação da mesma. Uma activity é assim um elemento com grande independência relativamente às restantes activities de uma aplicação e cujo controlo e manutenção do seu processo é da responsabilidade do sistema. Esse controlo garante segurança e o uso eficiente de recursos. Do ponto de vista do utilizador esse controlo poderia, se não fosse implementado de forma conveniente, gerar confusão, insatisfação e uma experiência de utilização frustrante. Para permitir controlo sobre os momentos em que o sistema gere o processo de uma activity a plataforma android implementa a ideia de ciclo de vida. Este ciclo corresponde a um conjunto de etapas que uma activity pode atravessar desde o seu lançamento até à sua destruição. Cada uma das transições pode ser gerida através de métodos que permitem ao programador a implementação de funcionalidades que garantam a qualidade da interação ou a realização de ações adequadas à preservação de recursos do sistema, como por exemplo a desativação do sistema de geolocalização ou o fecho de uma ligação a uma base de dados. O ciclo de vida de uma Activity é governado por um conjunto de métodos cuja correta implementação é da responsabilidade do developer, através da figura 2, proveniente da documentação da API da plataforma, podemos identifica-los e visualizar os eventos que determinam as suas possibilidades de encadeamento (Android API Guide, 2017).

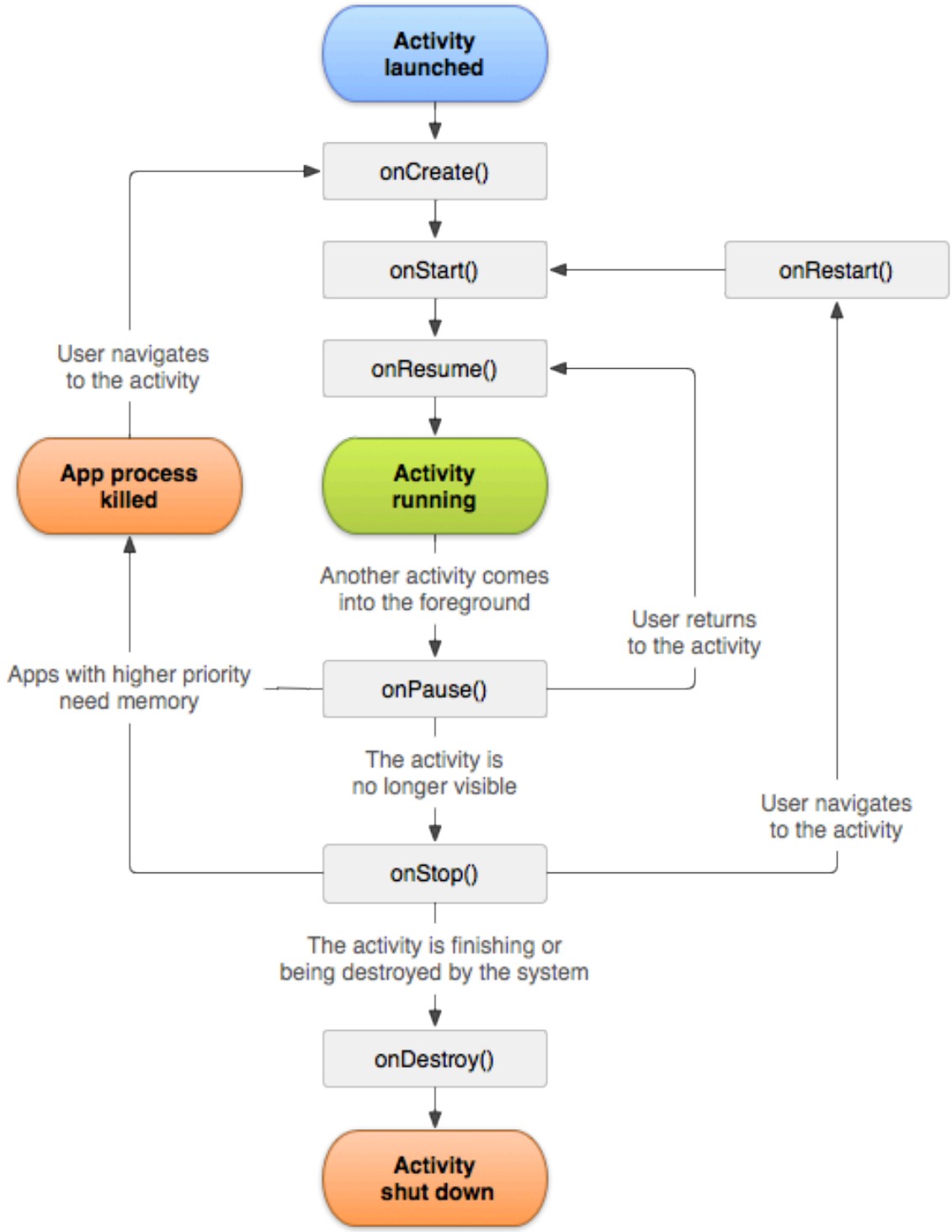

Figura 2: Ciclo de vida de uma Activity ilustrando os métodos que o governam, bem como os eventos que determinam a sua evocação. Figura proveniente da documentação online da API da plataforma (Android API Guide, 2017).

Através dos métodos onCreate(), onStart(), onResume(), onPause(), onStop(), onRestrat() e onDestroy() o programador tem oportunidade de implementar as medidas necessárias a cada uma das etapas que a activity pode atravessar.

O método onCreate() é chamado pelo sistema sempre que uma activity é lançada. Por este motivo a sua implementação é obrigatória. Será neste método que é feita a

inicialização da activity como por exemplo atribuição de variáveis usadas em diferentes métodos.
O método onStart() é invocado quando a activity se prepara para ser apresentada ao utilizador e a interação começa a ser possível.
O método onResume() é invocado imediatamente depois do onStart() ou quando o a ação do utilizador o faz retornar a uma activity que estivesse em pausa. É após a execução deste método que a interação com utilizador ocorre. Como este método é sempre invocado quando se regressa a uma activity que esteve em estado de pausa, deve ser usado para reestabelecer recursos e componente libertados ao entrar nesse estado.
Quando o utilizador abandona uma activity ou algum evento interrompe a execução da aplicação, como por exemplo uma chamada, a apresentação de um dialogo ou o apagar do ecrã do dispositivo, é invocado o método onPause(). Como referido anteriormente este será o momento para libertar recursos cuja atividade só faz sentido quando a activity tem o foco do utilizador, e que noutra circunstância não têm utilidade, e podem ter impacto significativo na autonomia. A execução deste método dever ser breve, pelo que não deve ser usado para fazer operações potencialmente demoradas como por exemplo o guardar de dados da aplicação (Android API Guide, 2017).
Quando a activity deixa de estar visível é invocado o método onStop(). Através dele o programador deve libertar praticamente todos os recursos utilizados pela aplicação. É a última oportunidade para libertar recursos com garantia de execução dessa operação pois é possível que o sistema destrua o processo onde corre a Activity sem invocar o método onDestroy(), pelo que todos os recursos que possam afetar o desempenho da plataforma deve ser libertados. O método onStrop() é também a oportunidade para realizar tarefas mais exigentes como por exemplo realizar alguma operação de persistência de dados. Todavia neste método não é necessário guardar a informação contida em objetos como por exemplo instâncias de EditText, porque essa é preservada pelo sistema e recuperada se o utilizador regressar à Activity.
A invocação do método onDestroy() ocorre quando a Activity vai ser destruída e é a última que receberá. Assim será o momento de libertar todos os recursos para os quais essa operação ainda não tenha sido realizada.

2.3.5. Funcionalidade de uma aplicação e a sua relação com a navegação entre Activies

As funcionalidades de uma aplicação e a interação com o utilizador implicam a transição entre Activities.
A navegação para uma nova Activity implica a criação de uma instância de Intent. Neste objeto é possível especificar o nome do componente que se pretende iniciar. Quando tal é feito cria-se um Intent explicito e ele apenas é disponibilizado a esse componente. Em alternativa o nome do componente pode ser omitido. O Intent é assim implícito e o sistema procurará encontrar um componente com capacidade para responder.

Um Intent pode passar dados para o componente que vai ser iniciado. Isto pode ser feito através de uma referência URI para os dados ou através de um conjunto de pares chave-valor. Para a primeira opção o Intent disponibilizaos métodos setData() para definir apenas a referência, setType() para definir apenas o tipo MIME dos dados e setDataAndType() para as situações em que é necessário definir explicitamente dados e tipo. Quando os dados são passados sob a forma de pares chave-valor, estes podem ser adicionados ao Intent através de métodos putExtra() aos quais é passado uma String com a chave e o respetivo valor. Existindo métodos para valores de diferentes tipos de dados. Definido o Intent ele é passado ao método que iniciará o componente pretendido, por exemplo startActivity() para iniciar uma Activity permitindo a navegação entre janelas de uma aplicação (Android API Guide, 2017).

2.3.6 User Interface na plataforma Android

A *user interface* de uma aplicação android pode ser vista como um arranjo de elementos que colocam algo no ecrã e de elementos que agrupam outros elementos. Na terminologia da plataforma, os primeiros são denominados de View, e os últimos de ViewGroup. Uma View pode ser um local onde é colocado texto editável, por exemplo um EditText. Um ViewGroup é uma estrutura que pode conter vários View e ViewGroup, permitindo a sua organização, por exemplo um LinearLayout . A organização da *user interface* é assim uma estrutura hierárquica cuja raiz é sempre um ViewGroup dentro do qual podem existir instâncias de View e de ViewGroup (Android API Guide ,2017). A figura 3 mostra um possível arranjo destes elementos na construção de uma *user interface*.

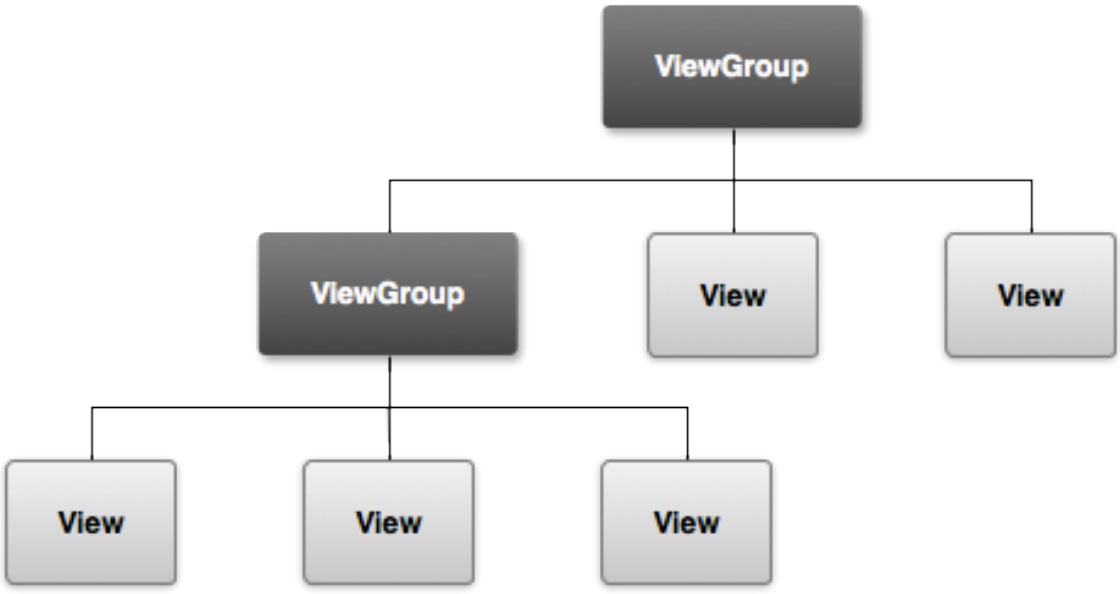

Figura 3: Ilustração de uma possível organização hierárquica de elementos para uma *user interface* na plataforma android. Figura proveniente da documentação online da API da plataforma (Andoid API Guide, 2017).

A plataforma oferece também componentes específicos que permitem desenvolver uma aplicação com aspeto coerente e familiar para o utilizador da plataforma. Esses componentes podem ser diálogos ou uma App Bar (action bar).
A estrutura e organização dos elementos de uma *user interface* é definida num layout. Este pode ser instanciado em runtime, a partir do código da aplicação ou definido e declarado num ficheiro XML independente do código da aplicação. A segregação do layout permite que uma mesma base de código possa ser usada em diferentes layouts, por exemplo para dar suporte a dispositivos com características físicas distintas ou adaptar facilmente a aplicação a utilizadores de diferentes nacionalidades. Esta definição do layout é compilada num recurso que pode ser pode ser carregado pela Activity aquando da invocação do método onCreate() através do método setContentView() (Android API Guide, 2017).
Cada View ou ViewGroup de um layout, pode ter uma série de atributos, através dos quais se definem as suas propriedades. Um desses atributos é o ID, que identifica a View de forma única na árvore que define o layout. Dessa forma é possível obter uma referência para a View no seio do código da aplicação possibilitando ações sobre a mesma. Além do ID existem atributos que podem ser usados para definir o comportamento e posição de cada elemento da *user interface* contexto do ViewGroup que os contem. Estes são genericamente designados de parâmetros de layout e surgem na definição XML como elementos com o prefixo "layout_". Numerosos outros atributos permitem definir aspetos específicos de uma View, que podem ir desde o estado de uma CheckBox ao tipo de dados que pode ser inserido num EditText (Android API Guide, 2017).
A plataforma disponibiliza uma série de ViewGroups com comportamentos predefinidos quanto à disposição dos elementos que contém (LinearLayout, RelativeLayout etc)++++ São disponibilizadas também layouts específicos que são alimentados com dados através de um Adapter em runtime. Os mais comuns são o ListView e o GridView e são subclasses de AdapterView. Neste tipo de layout o Adapter acede a uma fonte de dados, com a qual produz views que alimentam o layout. Para diferentes fontes de dados existem Adapters específicos, podendo o programado/developer criar novos se tal se justificar (Android API Guide, 2017).

2.3.6.1. Controlos de Input

Entre as View da plataforma android existem um conjunto de elementos que permitem que o utilizador introduza informação e interaja com a aplicação. Estes correspondem a controlos de input (input controls) e a cada um deles corresponde um elemento em XML na descrição do layout.
Entre os demais elementos de interação da plataforma encontram-se os botões (Button), os campos de texto (EditText ou AutoCompleteTextView), as checkboxes (CheckBox), os radio button (RadioGroup e RadioButton), os toggle button (ToggleButton), os spinner (Spinner) e os pickers (DatePicker, TimePicker). A cada um destes está associada uma ação sobre a interface.

A um botão está associada a ordem para a realização de uma ação, como por exemplo a gravação de dados, o envio de um email ou a ação de cancelamento. Um campo de texto permite a inserção e edição de texto, como por exemplo o preenchimento de um campo de um formulário. A atuação sobre o botão faz-se com um click. Esta gera um evento que pode ser gerido através de uma função identificada no layout através do atributo android:onClick="handlerMethod". A implementação do método, que neste caso seria handlerMethod faz-se na Activity que utiliza o layout. Em alternativa o método que é responsável pelo evento pode ser totalmente definido programaticamente no código da Activity atráves do método Button.setOnClickListener().
Uma checkbox pode ser usada para assinalar um valor que pode ter dois valores possíveis, como por exemplo verdadeiro ou falso, masculino ou feminino, “em stock” ou esgotado. Um conjunto destas estruturas permite criar um espaço de interação onde o utilizador pode realizar múltiplas seleções que não sejam mutuamente exclusivas. Tal como num botão os eventos gerados pelos clicks também podem ser capturados e tratados por métodos definidos no ficheiro de layout, através do atributo android:onClick desta View.
Através de grupos de botões de rádio (RadioGroup) é possível proporcionar um espaço de interação em que apenas é possível fazer uma de entre várias opções. Cada uma destas é apresentada sob a forma de um botão de rádio.
Um Toogle Button, que herda da classe CompoudButton, tem um funcionamento semelhante a um interruptor e pode ser usado para ativar ou desligar uma opção ou funcionalidade, por exemplo ativar ou desligar a rede wifi do dispositivo. O tratamento do evento correspondente à alteração do estado do botão, é definido programaticamente através da implementação dum listener associado à View através do método CompoudButton.OnCheckedChangeListener(), através do qual se pode fazer a transição entre os dois estados associados ao botão.
Quando há necessidade de fazer a seleção de um elemento de entre um conjunto de opções, pode ser usado um spinner através do qual se escolhe um dos elementos de uma lista que permanece escondida enquanto o utilizador não interage com o controlo. Assim normalmente só é apresentado o valor selecionado, limitando-se o espaço de ecrã utilizado. Este aspeto pode ser um fator importante a ter em consideração no momento do desenho da interface, na opção por este controlo em comparação com outros que oferecem funcionalidade semelhante como por exemplo um RadioGroup e respetivos RadioButton.
Um picker permite escolher um valor através de gestos no ecrã sobre o controlo. É normalmente usado para selecionar datas (DatePicker) ou informação horária (TimePicker) (Android API Guide, 2017).

2.3.6.2. Menus

Os menus são elementos da interface comuns numa grande diversidade de plataformas e aplicações, através deles são disponibilizadas opções de atuação sobre o sistema que o utilizador pode selecionar.

A plataforma android disponibiliza três tipos fundamentais de menus que podem ser utilizados em contextos de interação diferentes. O options menu é o principal menu de uma Activity. Este pode ser acedido através do botão menu do dispositivo, quando este está disponível (o que não é requisito desde a versão 3.0 do sistema) ou através da app bar (ou action bar). Este menu é o local para colocar as opções que permitem aceder a ações que tenham um caracter global no contexto da aplicação ou Activity. O context menu permite disponibilizar opções de ação sobre um elemento específico. Este é apresentado como um menu flutuante, sobre a Activity, quando o utilizador realiza um long-click sobre o elemento a que está associado. O context menu deve ser usado para realizar ações que afetam o conteúdo do item selecionado. O terceiro tipo de menu é o popup, através do qual é disponibilizada uma lista de opções ancorada numa view. A utilização pretendida para este último tipo é a de proporcionar um suporte para colocar ações extra relacionadas com a view selecionada, mas que não afetam o seu conteúdo. Por exemplo um popup menu sobre uma view que represente um contacto poderia proporcionar a opção de realizar uma chamada, enviar uma mensagem ou um email.
Na plataforma android os menus são descritos em ficheiros XML, que constituem um dos recursos da aplicação. Esses ficheiros especificam o menu e todos os seus itens. Este recurso é tornado acessível a partir do código da Activity para que as respetivas funcionalidades sejam implementadas. A associação entre uma Activity e os respetivos menus é feito através da implementação de funções callback. Para um options menu esse método é onCreateOptionsMenu() que permite identificar o respetivo recurso. A resposta aos eventos de seleção de itens do menu é feita através da implementação do método onOptionsItemSelected() através do qual é possível identificar o item selecionado e desencadear a resposta adequada. Um options menu todo também ser modificado em runtime através da implementação do método onPrepareOptionsMenu(), possibilitando por exemplo a ativação ou desativação de itens. Um context menu é muitas vezes usado em conjugação com uma ListView ou GridView para permitir a escolha de uma opção relativa a um dos elementos da mesma. Para gerar um destes menus é necessário registar a view para o efeito através do método registerForContextMenu(). O acesso ao layout do menu é conseguido através do método de callback onCreateContextMenu(). A implementação do código que responde à seleção de cada um dos itens é feita através do método onContectItemSelected(). A plataforma oferece muitas mais possibilidades de interação além das descritas para estes dois tipos de menus, todavia elas não foram usadas durante o processo de geração de código (Android API Guide, 2017).

2.3.7 Persistência de dados

A plataforma oferece várias opções para persistência de dados cuja escolha é dependente das necessidades da aplicação. Entre elas encontram-se as Shared Preferences em que os dados são armazenados sob a forma de pares chave-valor. Estes dados são privados da aplicação. Através do armazenamento interno (Internal Storage) é possível guardar dados privados na memória do dispositivo. O armazenamento externo (External Storage)

permite gravar os dados para um suporte externo, podendo desta forma os dados ser partilhados. As bases de dados SQLite estão disponíveis para o armazenamento de dados estruturados. Estes são diretamente acessíveis pela aplicação que cria a base de dados e dessa forma são dados privados da aplicação. No entanto a utilização de componentes Content Provider permite disponibilizar os dados gravados na base de dados a outras aplicações. Finalmente os dados podem ser gravados através da rede para um servidor externo (Android API Guide, 2017).

2.3.7.1. Utilização de bases de dados SQLite

A utilização de uma base de dados SQLite depende da sua criação na memória disponível no dispositivo. Esta estará disponível para as classes da aplicação, mas não será diretamente acessível fora do seu contexto.

A sua criação faz-se através da implementação de uma subclasse de SQLiteOpenHelper. Nesta a criação da base de dados faz-se através de uma implementação do seu método onCreate(). As operações de escrita e leitura são possíveis obtendo um objeto que representa a base de dados através dos métodos getWritableDatabase() e getReadableDatabase(). Na posse desse objeto é possível executar queries à base de dados. As queries retornam um Cursor através dos quais é possível aceder e navegar nos dados resultantes da sua execução (Android API Guide, 2017).

2.4. O PADRÃO MODEL-VIEW-CONTROLLER (MVC)

A arquitetura de uma aplicação afeta o seu desempenho e a possibilidade de fácil manutenção. No contexto do desenvolvimento de aplicações com uma interface grágica de utilizador (GUI) o padrão Model-View-Controller (MVC) organiza a aplicação em camadas, separado a apresentação e a interação, dos dados do sistema (Sommerville, 2011). Nesta arquitetura um Modelo (Model) é o componente que gere os dados do sistema e as operações com eles associados. Uma View é o componente responsável pela apresentação dos dados ao utilizador. Por fim o Controller lida com os aspetos da interação com o utilizador. Assim é responsável por estabelecer o mapeamento entre as ações do utilizador e a alterações dos dados ou por garantir a navegação entre diferentes apresentações dos diferentes modelos (Sommerville,2011).

Este padrão arquitetural ao permitir a separação entre os dados e a sua apresentação, garante a possibilidade de modificação independente de cada uma das camadas (Sommerville, 2011). Assim é possível a apresentar de forma diferente o mesmo conjunto de dados ou fazer com que uma alteração dos dados seja facilmente refletida em várias vistas (Sommerville, 2011).

A separação dos componentes em camadas aumenta a complexidade da aplicação, no entanto, fornece como retorno uma maior facilidade de intervenção sobre componentes sem repercussões sobre todo o sistema. Este aspeto é particularmente útil para a manutenção e evolução de um sistema (Sommerville, 2011).

2.5 Conclusões

A modelação pode ser encarada como uma etapa fundamental no processo de desenvolvimento de aplicações, podendo mesmo assumir um papel central. Estão atualmente disponíveis instrumentos que permitem capturar a estrutura e comportamento pretendido para o futuro sistema, entre os quais se encontram os modelos de domínio e de casos de uso. Através de um processo de transformação de modelos é possível gerar aplicações, por refinamentos sucessivos, para plataformas específicas. Tal depende do estabelecimento de mapeamentos e regras de transformação que convertam a informação contida nos modelos em artefactos específicos de uma plataforma, aderindo a uma arquitetura que facilite a sua evolução e manutenção.

3. ANÁLISE DO ESTADO DA ARTE

3.1 INTRODUÇÃO

Neste capítulo é feita uma análise ao estado da arte no que respeita a abordagens de desenvolvimento de software guiadas por modelos, focando inicialmente abordagens que procuram responder à criação de uma *User Interface* (UI), e depois à criação de serviços que possam ser consumidos por plataformas móveis. Depois, aborda-se abordagens mais completas que procuram dar resposta aos diferentes aspetos que conduzem à criação de uma aplicação completa e funcional, sendo exclusivamente focadas abordagens que respeitam à criação de aplicações móveis. Por fim, procura-se fazer uma comparação entre as soluções apresentadas nas diferentes secções.

3.2 ABORDAGENS COM ENFÂSE NO DESENVOLVIMENTO DE *USER INTERFACES*

A *User Interface* é a componente do sistema através do qual se realiza a interação entre o homem e a máquina. A sua construção é fundamental para que um objetivo ou tarefa do utilizador seja realizável através de um sistema.

A especificação da *User Interface* pode começar a ser realizada a níveis diferentes de abstração e em associação com a construção de outros modelos do sistema a desenvolver. Neste campo existem linguagens que permitem a sua descrição para posterior transformação. No campo das linguagens de descrição de interfaces existe a UsiXML que permite a especificação da *User Interface* a vários níveis de abstração. É então possível uma especificação desde um nível de tarefas e conceitos até ao nível concreto de objetos de interface, como por exemplo botões para diferentes plataformas (Limbourg & Vanderdonckt, 2004). A transformação entre os diferentes níveis é possível quer no sentido de uma maior abstração ou no sentido inverso, bem como entre modelos de níveis de abstração equivalentes (Limbourg & Vanderdonckt, 2004). No nível mais concreto é possível gerar interfaces Web que podem ser consumidas por dispositivos móveis. A utilização da UsiXML pode envolver a criação ou transformações de/para Modelos de Domínio, Modelo de Tarefas (Task Model), Modelo Abstrato da *User Interface* (auiModel), Modelo Concreto da *User Interface* (cuiModel), Modelo de Mapeamentos (mappingModel), Modelo de Contexto (contextModel) e um Modelo de transformações (transformationModel).

A MARIA XML (Paternó, Santoro & Spano, 2009) é uma proposta que inclui um modelo de dados para permitir uma maior relação entre a interface e os dados subjacentes. Inclui ainda um modelo de eventos para especificar como a interface responde a eventos gerados pelo utilizador, como por exemplo eventos de ativação ou de alteração. A linguagem fornece também capacidade para especificar partes dinâmicas da interface que vão da distribuição de elementos dentro de uma vista, à navegação entre vistas.

A linguagem permite definir a interface a um nível abstrato e independente da plataforma ou a um nível concreto dependente da plataforma, mas independente da linguagem de implementação.
A MARIA XML permite ainda gerar anotações que facilitam a integração de Web services com a UI. Disponibiliza um ambiente que permite definir transformações diretas e reversas entre modelos, permitindo navegar entre níveis de abstração diferentes. (Paternó, Santoro & Spano, 2009). A interface gerada é uma interface Web que pode ser consumida por dispositivos móveis.
A abordagem apresentada em (Cruz, A.M., 2010; Cruz, A.M. & Faria, J.P., 2010; Cruz, A.M., 2015) baseia-se na geração de um modelo da UI a partir de modelos com a visão estrutural do sistema (modelo de Domínio) e com a visão da funcionalidade (modelo de Casos de Uso). O modelo da UI é gerado com base em padrões de Casos de Uso, e na sua ligação ao modelo de domínio (Cruz, A.M., 2014). Após gerado o modelo da UI, o engenheiro de software pode alterar este desde que mantenha a ligação ao modelo de domínio. A geração da UI é então possível a partir dos modelos de domínio e da UI, sendo atualmente gerado código XUL (Cruz, A.M., 2010; Cruz, A.M. & Faria, J.P., 2010) e um protótipo aplicacional em Javascript.
Atingir plataformas móveis através de interfaces Web é também a proposta de Brambilla, Mauri & Umuhoza (2014), através de uma extensão à linguagem Interaction Flow Modeling Language (IFML). O código gerado a partir dos modelos descritos naquela linguagem é adaptado à framework Cordova da Apache. Esta framework tem no PhoneGap da Adobe uma das suas implementações mais conhecidas. Usando o PhoneGap é possível obter o ficheiro apk para Android ou ipa para iOS partindo do código que produzia a interface Web (Brambilla, Mauri & Umuhoza , 2014).
A produção de interfaces multimodais capazes de se adaptarem a diferentes tipos de dispositivos é proposto por Porta (2010). Este autor apresenta uma metodologia para a geração dessas interfaces, pondo a enfâse nos serviços associados a processos de negócio. Nestes é comum a participação de vários intervenientes cujo papel pode ser descrito através de casos de uso e diagramas de atividade. Estes são usados para obter um Modelo da Tarefa (Task Model) que descreve a decomposição das tarefas e a colaboração entre os Atores. Partindo do Modelo da Tarefa e da descrição das interações em DiaMODL seria possível, através transformações sucessivas, obter uma *User Interface*, consumível por um browser, e uma camada de software, que a suporta. Esta última distribui-se entre o dispositivo cliente e a camada de serviços.

3.3 Abordagens Com Enfâse no Desenvolvimento de Serviços

Uma das características das novas categorias de dispositivos móveis, nomeadamente dos smartphones é a facilidade em aceder a dados na Internet. Esta capacidade está muitas vezes disponível através de diferentes tecnologias. Em virtude deste facto muitas das aplicações preveem no seu funcionamento o acesso a dados externos.

Uma forma de obter esses dados externos é através de Web services e estes podem não ter sido concebidos para dar resposta adequada a dispositivos móveis. Os motivos podem ser vários e estar relacionados com o momento em que foram concebidos ou com a necessidade de dar respostas a outros tipos de clientes. Os smartphones impõem algumas restrições a um Web service. O ecrã tem normalmente reduzidas dimensões o que não o torna adequado à apresentação de grande quantidade de informação, mas útil para consumir informação resumida e simples. A criação de Web services flexíveis pode garantir que estes sejam consumidos por diversos tipos de clientes incluindo dispositivos móveis (Ortiz & Prado, 2010). Para averiguar desta possibilidade, Ortiz & Prado (2010) inventariam várias soluções possíveis, que analisam através de uma grelha que considera cinco itens:

1. A transparência para o criador do serviço quanto ao conhecimento da plataforma ou plataformas clientes.
2. A consistência de funcionamento do serviço entre versões anteriores e posteriores à adaptação para responder a clientes móveis.
3. A duplicação de código do lado serviço para que ocorra a sua adaptação a diferentes tipos de clientes.
4. A necessidade que o cliente tem de conhecer características do serviço de forma a invocar o método correto para obter a resposta mais adaptada ao tipo de dispositivo.
5. A necessidade de introdução, do lado do serviço, de código não relacionado com a sua função.

Desta análise resulta a proposta de um serviço SOAP cujo *header* é modificado para incluir informação sobre o cliente móvel. Este tipo de serviços seria modelado num modelo independente da plataforma (PIM), que incluiria um conjunto de estereótipos para assinalar as classes que representam o Web service e os métodos que produzem respostas distintas para clientes diferentes. A existência desses métodos implicaria a distinção entre os atributos, pois existiria um conjunto que faria parte da resposta a todos os dispositivos e outros atributos que surgiriam apenas nos dispositivos cujas características permitissem o consumo de uma informação mais detalhada. Assim os atributos correspondentes à informação mais resumida e consumível por dispositivos móveis seria assinalada com um estereótipo. A geração do Web service é atingida por um conjunto de transformações partindo do modelo inicial.

3.4 Abordagens para o Desenvolvimento de Aplicações Completas

A principal vantagem do desenvolvimento orientado por modelos é que através da abstração das plataformas concretas se isola o *developer* da sua variabilidade e especificidade. Os mecanismos de transformação automatizada ou semi-automatizada lidam com os detalhes da plataforma permitindo que o *developer* dedique mais do seu

tempo à funcionalidade pretendida na aplicação a desenvolver. Esforços no sentido de um isolamento dos detalhes de implementação em diferentes plataformas móveis têm sido feitos dos quais se destacam os seguintes.

Heitkötter, Majchrzak e Kuchen, 2013 propõem um processo em três etapas, sendo que destas apenas a modelação, que correspondente à primeira delas, envolve um trabalho cognitivo significativo por parte do *developer*. As duas seguintes a geração do código e sua compilação requerer menor empenho e são praticamente automatizadas. A *framework* proposta, o md2, é responsável pelas duas primeiras etapas. A modelação da aplicação é realizada através de uma linguagem específica do domínio (DSL), que faz parte integrante da *framework* e desenvolvida especificamente para o domínio das aplicações móveis. Além do código para o dispositivo móvel é gerado código para suportar por exemplo serviços de *back-end*. A compilação do código gerado em Objective-C para iOS ou Java para Android é usado nas respetivas plataformas de desenvolvimento obtendo-se através delas aplicações nativas para os dispositivos móveis com quota de mercado.

Parada e Brisolara , 2012 , propõem uma solução para a geração de aplicações para Android, em que o sistema é modelado usando UML. A visão estrutural do sistema é modelada com diagrama de classes e a visão comportamental através de diagramas de sequência. O modelo produzido é já um modelo específico para a plataforma.

Na visão estrutural as classes herdam de duas superclasses, Activity e Service, em que a primeira representa elementos com os quais o utilizador interagirá e a segunda representa processos que ocorrem em background, tendo uma correspondência direta com os elementos com o mesmo nome da plataforma Android. As subclasses, representam elementos de cada tipo na futura aplicação e especificam quais os métodos a serem adaptados. Estes podem correspondem ao métodos padrão da plataforma como o onCreate().

Na visão comportamental são usados diagramas de sequência. Estes podem especificar a utilização de componentes específicos da plataforma Android como os Intent ou Content Providers, ou interações, trocas de mensagens e outras condições típicas na modelação de um sistema. Os mesmos diagramas modelam também os métodos que são adaptados em cada uma das subclasses que herdam de Activity ou Service, como por exemplo o onCreate().

Durante a geração de código a visão estrutural dá suporte à criação dos ficheiros Java para cada uma das classes. A visão comportamental suporta a geração do código dos métodos.

A XIS-Mobile baseia-se a linguagem XIS, estendendo o UML através de perfis que permitem criar modelos a partir dos quais pode ser gerada uma aplicação móvel. Esses modelos dão quatro visões diferentes do sistema. Essas são a visão das Entidades, que envolve a visão do Domínio e a visão das BusinessEntities, permitindo definir as entidades relevantes para no contexto do problema. A visão dos Casos de Uso que detalham as operações que o utilizador pode realizar e que podem incluir as operações CRUD e a operação de pesquisa. A visão arquitetural que permite especificar as interações entre a

aplicação móvel e serviços que lhe são externos. Finalmente a visão da *User Interface* modela a navegação e os espaços de interação da aplicação (Ribeiro & Silva , 2014).

Vaupel et al. (2014) propõem uma abordagem que permite o desenvolvimento de uma aplicação móvel a diferentes níveis de abstração: modelação compacta de elementos standard da *app*, modelação detalhada de elementos individuais, e a possibilidade de modelos para necessidades específicas. A abordagem propõe um meta-modelo para especificar comportamento.

Kramer et al. (2010) desenvolveu o MOBDSL, uma linguagem específica do domínio (DSL) para especificar aplicações móveis de forma independente da plataforma. Esta abordagem não gera código nativo para as plataformas suportadas. Ela gera código para uma máquina virtual que tem que ser desenvolvida para cada plataforma.

Balagtas-Fernandez et al. (2010) desenvolveram Mobia modeler e framework, que permite a utilizadores sem formação técnica, normalmente especialistas de um domínio, modelar a *user interface* e funcionalidade através de componentes configuráveis de uma forma independente da plataforma (Mobia PIM). A utilização de um modificador transforma o Mobia PIM num modelo específico da plataforma, que é fundido com templates para gerar código para plataformas específicas.

Applause (Applause) propõem uma DSL para descrever aplicações móveis e um conjunto de geradores de código baseados no Eclipse e Xtext que utilizam essas descrições para gerar aplicações móveis nativas e centradas em dados para as principais plataformas (iOS, Android, Windows Phone). ModAgile Mobile (ModAgile) também propõem uma abordagem para geração de aplicações centradas em dados orientadas por modelos e permitindo um desenvolvimento multiplataforma eficiente. Todavia as duas exigem uma modelação muito detalhada da interface do utilizador.

3.5 Comparação das Diferentes Abordagens

Na comparação entre as diferentes abordagens optou-se pela sua realização dentro de cada uma das categorias analisadas. Assim na tabela 1 comparam-se as formas de produção de *User Interfaces*. Na tabela 2 sumaria-se a informação recolhida sobre a produção de Web services flexíveis. Finalmente na tabela 3 apresenta-se a comparação entre as metodologias que permitem a obtenção de aplicações nativas para dispositivos móveis.

Tabela 1: Comparação entre as diferentes abordagens para a geração de *User Interfaces* capazes de serem consumidas em dispositivos móveis.

Designação e Referência	Modelos utilizados	Transformações possíveis	Forma de acessibilidade por dispositivos móveis
UsiXML(Limbourg & Vanderdonckt , 2004)	UIModel constituido por: Domain Model, Task Model, auiModel , cuiModel, mappingModel, contextModel e transformationModel	No sentido de uma menor ou maior abstração ou e entre níveis de abstração equivalentes	Não nativa (Web)
MARIA XML (Paternó, Santoro, & Spano, 2009)	Modelo de dados, modelo de eventos, como são feitos	Configuráveis nos dois sentidos	Não nativa (Web)
IFML (extensão) (Brambilla , Mauri , & Umuhoza , 2014)	Parece-me uma linguagem de diagramas específica	No sentido de uma menor abstração	Não nativa (Web via apache Cordova) Possibilidade de gerar apk ou ipa através do PhoneGap
DiaMODL (Porta, 2010)	Modelo das interações (diaMODL) e Task Model	No sentido de uma menor abstração	Não nativa (Web)
(Cruz, A.M., 2010) (Cruz, A.M. & Faria, J.P., 2010) (Cruz, A.M., 2014) (Cruz, A.M., 2015)	Modelo de Domínio com a visão estrutural do sistema, modelo de Casos de Uso com a visão da funcionalidade	Geração de modelo da UI com a visão de apresentação do sistema, e daí geração de código (XUL)	Não nativa (XUL)

Tabela 2: Comparação entre as diferentes abordagens para a geração de Serviços passíveis de serem consumidas em dispositivos móveis.

Designação e Referência	Modelos Utilizados	Transformações possíveis	Forma de acessibilidade por dispositivos móveis
(Ortiz & Prado , 2010)	Modelo independente da plataforma do Web service apresentado usando diagramas de classes UML com estereótipos específicos.	Produção de um Web service a partir de modelos mais abstratos	Gerado um Web service que a partir da informação contida no header do pedido SOAP, consegue determinar o tipo de dispositivo e dessa forma envia uma resposta mais concisa ao dispositivo móvel.

Tabela 3: Comparação entre as diferentes abordagens para a geração de aplicações para dispositivos móveis.

Referência	Modelos Utilizados	Transformações possíveis	Forma de acessibilidade por dispositivos móveis
md2(Heitkötter, Majchrzak, & Kuchen, 2013)		Gerar a aplicação	Nativa iOS (objective-C) e Android
(Parada & Brisolara , 2012)	Structural View (usando diagrama de classes) Behavioral View (usando diagrama de sequência) (Modelos específicos da plataforma)	Gerar a aplicação	Nativa Android
XIS-Mobile (Ribeiro & Silva , 2014)	Perfil UML com Entity View (DomainView + BusinessEntitiesView), UseCase View, Architectural View, e *User Interfaces* View	Gerar a aplicação	Nativa Android e WindowsPhone

Referência	Modelos Utilizados	Transformações possíveis	Forma de acessibilidade por dispositivos móveis
Vaupel et al 2014	Modelação dos elementos standard da app; modelação detalhada de elementos individuais e possibilidade de modelos para necessidades específicas. Propõe meta-modelo para especificar comportamento	Gerar a aplicação	
Kramer et al 2010 MOBDSL,	DSL para especificar a aplicação móvel de forma independente da plataforma	Geração de uma aplicação	A aplicação é gerada para uma máquina virtual que é desenvolvida para cada plataforma. A aplicação não é por isso nativa para nenhuma das principais plataformas móveis.
Mobia modeler Balagtas-Fernandes et al 2010	Modelos de interface e de funcionalidade através de componentes configuráveis (Mobia PIM)	Gerar a aplicação	Geração de código para para plataformas especificas
Aplause	DSL para descrever aplicações móveis. Modelos detalhados da interface do utilizador	Gerar a aplicação	Principais plataformas (iOS, Android e Windows Phone)
ModAgile	Modelos detalhados da interface do utilizador	Gerar a aplicação	multiplataforma

3.6 Conclusões

A construção de um novo sistema orientado por modelos com subsequente geração do código pode seguir diferentes abordagens. Pode ter uma enfase no espaço de interação onde decorrerão as ações que conduzem à realização de tarefas ou objetivos do utilizador. Nestas abordagens o espaço de interação pode ser modelado de forma bastante concreta ou derivado a partir de modelos de maior nível de abstração. Entre estas possibilidades encontram-se modelos de casos de uso, diagramas de atividade, modelos de tarefas ou de modelos abstratos da *user interface*. Estes modelos de elevado nível podem ser transformados em *user interfaces* concretas com os quais se quer atingir a maior diversidade de plataformas. Neste sentido a opção pela geração para interfaces Web ou para frameworks como o Cordova permite ultrapassar a necessidade de geração de código nativo para cada uma das plataformas, embora à custa do sacrifício de algumas funcionalidades específicas ou mais recentes de cada uma.

Uma outra abordagem pode ser modelar o futuro sistema como uma plataforma de serviços que possam ser consumidos por um número alargado de clientes, entre os quais se encontram os dispositivos móveis. No entanto esta abordagem enfrenta a dificuldade da diversidade de meios de acesso pelo que se impõe a necessidade de modelar respostas diferenciadas e adequadas a diferentes clientes. O desenvolvimento de aplicações nativas para as plataformas móveis permite aproveitar todas as suas potencialidades. Neste caso os modelos sofrem um processo de transformação que gera código nativo. As abordagens que procuram explorar esta vantagem seguem estratégias de modelação diversas.

Numa primeira abordagem modelação e geração de aplicações nativas para dispositivos móveis suportados pela plataforma android o autor deste relatório procurou explorar a derivação de uma *user interface* funcional a partir de um modelo de domínio (Silva, Paiva, & Cruz, 2016) (figura 4).

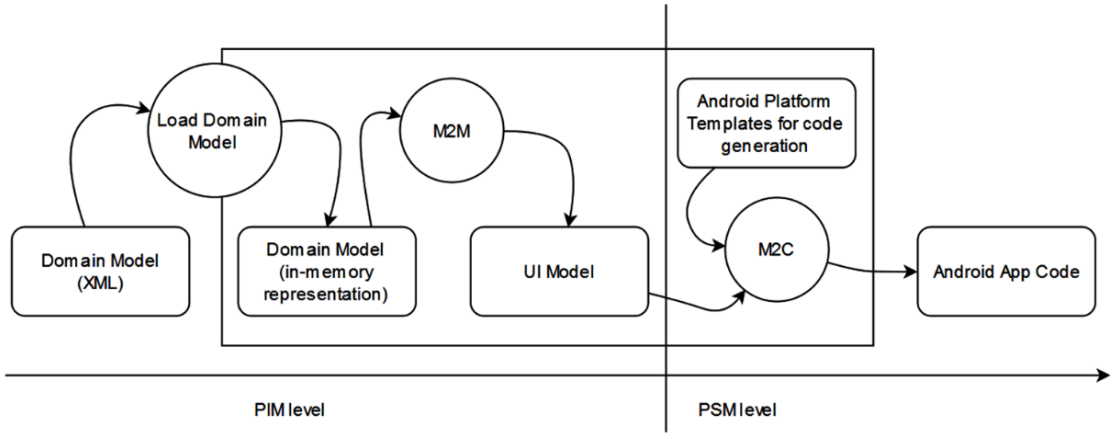

Figura 4: Transformações aplicadas, como refinamentos sucessivos, desde o modelo de domínio até à geração do código da aplicação. Retirado de Silva, Paiva, & Cruz, 2016.

O âmbito dessas aplicações foi limitado a sistemas centrados em dados manipuláveis através de formulários. Para tal utilizou-se uma simplificação e adaptação do meta-modelo de *user interface* proposto por Cruz & Faria, 2010 para gerar um modelo que

através de transformações M2T originavam uma descrição de layouts em XML e código java das respetivas Activities . Nesta abordagem inicial a persistência de dados foi simulada através de mapeamentos chave, valor. A geração do XML dos layouts bem como o código java a partir do modelo era baseada em Templates.
A abordagem então ensaiada foi posteriormente modificada e ampliada de forma a que a persistência de dados fosse feita em base de dados gerada a partir do modelo de domínio e a derivação da *user interface* fosse feita a partir de um modelo de casos de uso integrado com o modelo de domínio.

4. Desenvolvimento guiado por modelos de Aplicações Android

4.1 Introdução

A produção de uma aplicação a partir de modelos exige a definição dos modelos e do processo de transformação. Neste capítulo discute-se a utilização do editor AMALIA, produzido em projetos anteriores com a participação do autor deste documento, para gerar uma descrição de modelos de domínio e de casos de usos, e o processo da transformação desses modelos em código e texto (XML para a definição dos layouts das Activities) para a plataforma Android. Esse processo é apresentado de forma sumária na figura 5.

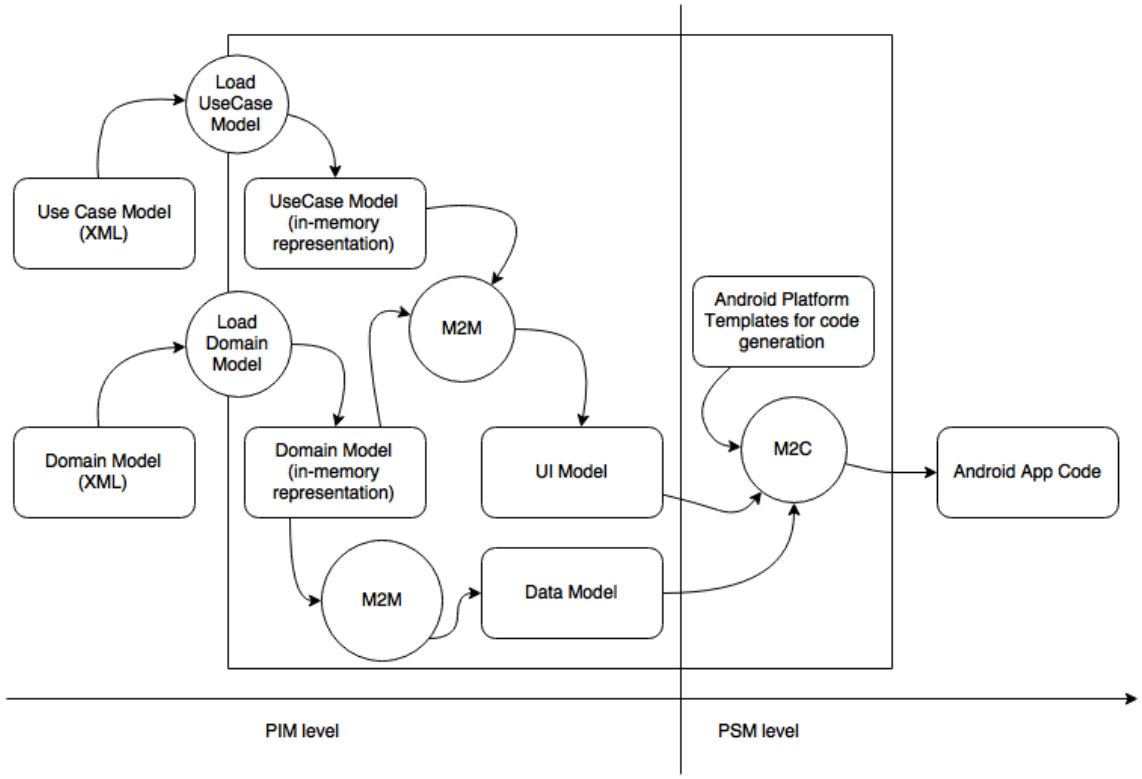

Figure 5: Representação sumária dos processos de transformação de modelos de domínio e de casos de uso numa aplicação para a plataforma Android.

Os exemplos apresentados usam o modelo de domínio e de casos de uso das figuras 6 e 7, respetivamente, apresentados tal como foram criados no editor AMALIA. O exemplo corresponde a um hipotético clube desportivo onde se praticam diferentes desportos (Sport). Cada desporto tem os seus atletas (Athlete), eventos (Event). As convocatórias (Call) associariam atletas a eventos. Exploram-se apenas casos de uso associados a um hipotético dirigente desportivo que tem por responsabilidade gerir os desportos e respetivos atletas.

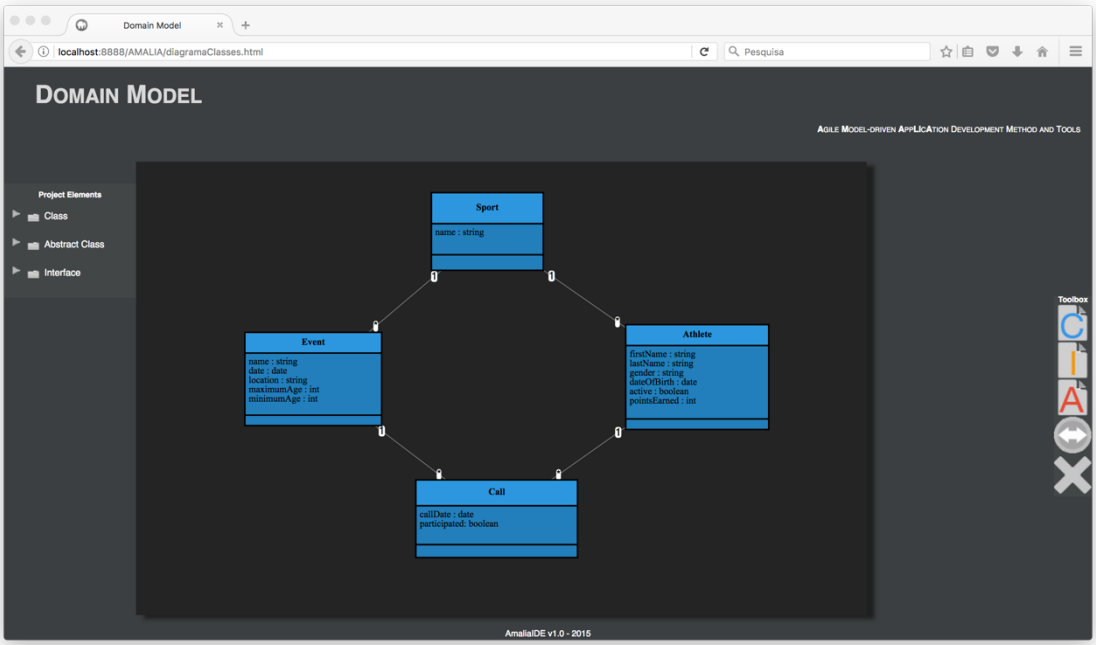

Figura 6: Modelo de domínio utilizado como exemplo na geração de uma aplicação Android.

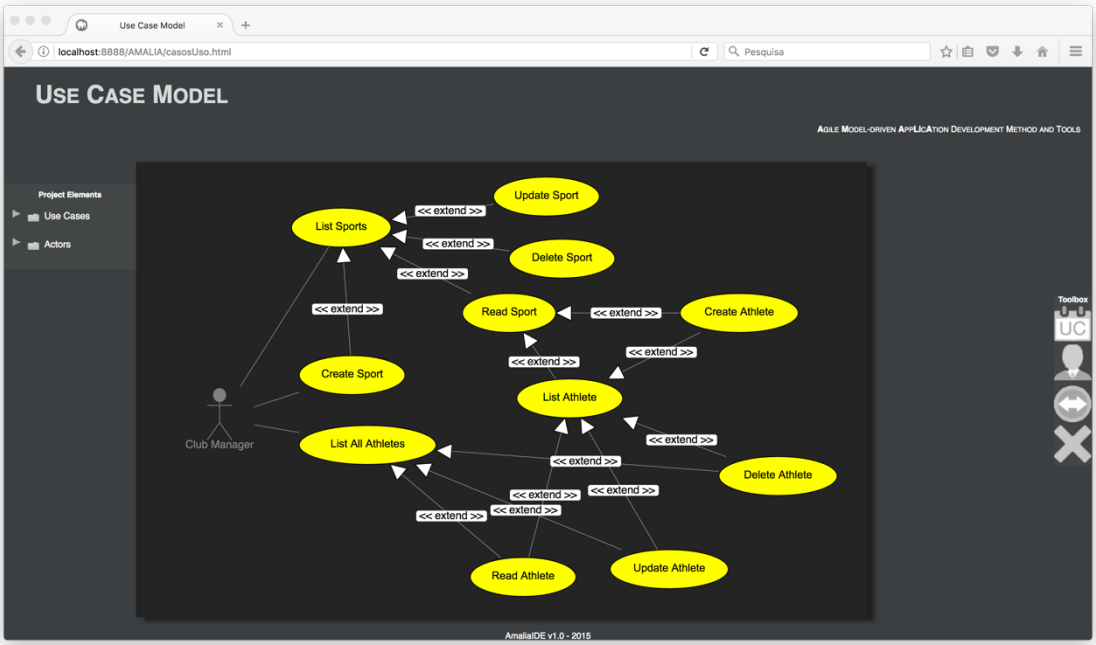

Figura 7: Modelo de casos de uso usado como exemplo na geração de uma aplicação Android e que define o conjunto de ações de um hipotético dirigente desportivo.

Uma implementação do processo representado na figura 5 foi feita em Java 8 para servir de prova de conceito e para testar possíveis soluções.

4.2 A FERRAMENTA AMALIA, SUA UTILIZAÇÃO PARA A ELABORAÇÃO DE UM PRIMEIRO CONJUNTO DE MODELOS E O SEU OUTPUT.

O editor Amália foi um projeto iniciado pelo autor deste relatório e posteriormente melhorado no âmbito de vários projetos desenvolvidos no decorrer da licenciatura em engenharia informática do IPVC, sob orientação do Professor Miguel Cruz. A sua utilidade resulta da possibilidade de produzir um modelo de uma aplicação de software composto por um conjunto integrado de (sub-)modelos de domínio e de casos de uso. A integração entre esses modelos reflete-se na possibilidade de poder explicitamente fazer referência a entidades do modelo de domínio, a partir do modelo de casos de uso. O sistema de software modelado no Amália tem, assim, duas visões integradas, modeladas através dos dois modelos referidos: uma visão estrutural, modelada através do modelo de domínio, a qual representa a estrutura informacional do sistema; e, uma visão da funcionalidade do sistema, modelada através do modelo de casos de uso. O modelo de casos de uso é, assim, visto como um modelo da funcionalidade ou features apresentadas pelo sistema aos seus utilizadores (modelados como atores).

No modelo de casos de uso é apenas possível definir um conjunto limitado de funcionalidades/operações. Estão previstas as operações de criação, leitura, atualização, apagar, ou seja, as operações CRUD, às quais se associa a operação de listar. Todavia este conjunto limitado de operações está na base da maioria das features das aplicações baseadas em formulários, pelo que a ferramenta se tornou o ponto de entrada para o desenvolvimento deste projeto.

Da ferramenta AMALIA é utilizado o seu output em formato XML, que corresponde à descrição textual dos modelos criados graficamente. Neste output o modelo de domínio é representado sob a forma de uma lista de elementos representados pelo tag <element> em que cada um corresponde a uma classe do modelo, tal como exemplificado na listagem1. O nome da classe é definido no tag<name>. Através do tag <atribute> é possível definir o nome, visibilidade e tipo de dados dos atributos de uma entidade. Por defeito a visibilidade é *private* e o tipo de dados é *string*. O editor permite a definição de outros tipos de visibilidade e de dados, a responsabilidade pela sua correta definição é da responsabilidade do modelador.

Listagem 1: Exemplo da definição das entidades do modelo de domínio. Cada entidade pode conter atributos definidos através de <atribute>. Por defeito os atributos têm visibilidade *private* e tipo *string*, sendo, no entanto, possível definir outra visibilidade ou tipo de dados.

```
<domain_model>
        <elements>
                <!--  conteúdo omitido-->
                <element id="3dd0f67f-4654-4807-aab9-49f7bb47b711">
                        <type>class</type>
                        <name>Athlete</name>
                        <attributes>
                                <attribute visibility="private" type="string">firstName</attribute>
                                <attribute visibility="private" type="string">lastName</attribute>
                                <attribute visibility="private" type="string">gender</attribute>
                                <attribute visibility="private" type="date">dateOfBirth</attribute>
```

```
                    <attribute visibility="private" type="boolean">active</attribute>
                    <attribute visibility="private" type="int">pointsEarned</attribute>
                </attributes>
            </element>
        </elements>
        <!-- conteúdo omitido -->
</domain_model>
```

As relações entre as classes do modelo de domínio são registadas numa lista de elementos representados pelo tag <connection> em que cada um representa uma relação entre duas entidades, tal como exemplificado na listagem2. Nessa especificação o tag<type> permite definir o seu tipo. Os tag<source_id> e <destination_id> permite definir os extremos da relação através do identificador gerado pelo AMALIA para entidades envolvidas. Os tag<cardinality_destination> e <cardinality_source> permitem definir as respetivas cardinalidades.

Listagem 2: Exemplo da representação de uma relação entre duas entidades do modelo de domínio. O tipo da relação é definido no tag<type> e pode ser association, aggregation, composition ou generalization. As extremidades da relação são representadas pelos identificadores gerados pelo AMALIA. A cardinalidade de cada uma das extremidades é igualmente especificada

```
<domain_model>
        <!-- conteúdo omitido -->
        <connections>
            <connection id="95f473ee-0a80-44bd-849a-29fbc982c1fa">
                <type>association</type>
                <cardinality_destination>1</cardinality_destination>
                <cardinality_source>*</cardinality_source>
                <source_id>3dd0f67f-4654-4807-aab9-49f7bb47b711</source_id>
                <destination_id>96adc707-d445-48fb-b5d6-f6d036bbc6ad</destination_id>
            </connection>
            <!-- conteúdo omitido -->
        </connections>
</domain_model>
```

De forma semelhante a representação do modelo de casos de uso é feita através de duas listas. Uma usando o tag <element> para representar atores e casos de uso e outra usando o tag <connection> para representar cada uma das relações entre esses elementos. A estrutura da representação dos Atores e casos de uso é exemplificada na listagem 3, onde é visível a utilização do tag <type> para a identificação do tipo de elemento e do tag<name> para a especificação do respetivo nome.

No AMALIA a descrição dos casos de uso é restringida a um conjunto limitado de operações que envolvem no máximo duas entidades do modelo de domínio. Essa restrição é visível no diálogo do AMALIA em que a especificação do caso de uso é feita (figura8).

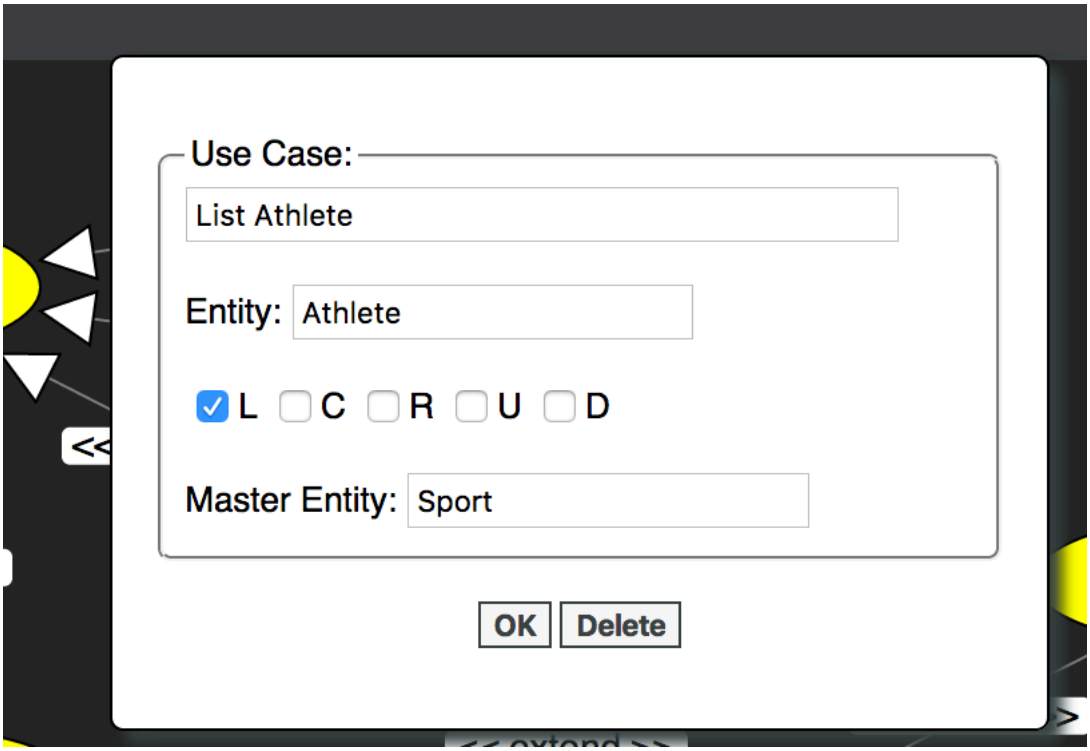

Figura 8: Dialogo de especificação de um caso de uso. Através dele é possível definir o nome do caso de uso, a entidade sobre a qual se realizarão as operações definidas (CRUD ou Listar). É igualmente possível definir uma entidade (master entity) com a qual a primeira está relacionada, permitindo assim a especificação de um padrão master-detail.

Assim na descrição XML de um caso de uso o tipo elemento - <type> - é *use case* (casos de uso). A operação é especificada através do tag<operation> e a entidade sobre a qual é executada é identificada com o tag<entity>. Quando a operação é realizada sobre uma entidade relacionada com outra o tag<masterEntity> permite fazer a sua referenciação.

Listagem 3: Exemplo da especificação de elementos de um modelo de casos de uso. Os elementos podem ser do tipo Actor ou Use Case. Um Use Case deve ter associada no mínimo uma entidade e no máximo duas (*entity* e *master entity*). A operação (*operation*) realizar-se-á sempre sobre a entidade especificada em *entity*.

```
<use_case_diagram empty="false">
        <elements>
                <element id="4b333f7e-b27b-4462-86a0-cf8e3d2ca517">
                        <type>actor</type>
                        <name>Club Manager</name>
                </element>
                <element id="68d79451-f20c-47f3-984e-87a0f7ba95c5">
                        <type>use case</type>
                        <name>List Sports</name>
                        <operations>
                                <operation>list</operation>
                        </operations>
                        <entity>Sport</entity>
                </element>
        <element id="fb3f6a23-72b0-4f6c-8d90-b195df346f70">
                        <type>use case</type>
                        <name>List Athlete</name>
```

```
                    <operations>
                        <operation>list</operation>
                    </operations>
                    <entity>Athlete</entity>
                    <masterEntity>Sport</masterEntity>
                </element>
                <!-- conteúdo omitido -->
        </elements>
</use_case_diagram>
```

A listagem 4 exemplifica a representação das relações entre os elementos do modelo de casos de uso. O tipo de relação é definido através do tag<type> e os dois elementos envolvidos na relação são identificados através dos tag<source_id> e <destination_id>, através de um identificador gerado pelo editor AMALIA.

Listagem 4: Exemplificação da especificação de relacionamentos entre elementos de um modelo de casos de uso. Os tipos de relacionamento são especificados através do tag<type> e podem ser do tipo association, include ou extend. Os extremos do relacionamento são identificados pelos identificadores gerados pelo AMALIA.

```
<use_case_diagram empty="false">
        <!-- conteúdo omitido -->
        <connections>
                <connection id="affaf114-5908-4ac7-82aa-5c2cea1d23b0">
                        <type>association</type>
                        <source_id>68d79451-f20c-47f3-984e-87a0f7ba95c5</source_id>
                        <destination_id>4b333f7e-b27b-4462-86a0-cf8e3d2ca517</destination_id>
                </connection>
                <connection id="1cd9c5f7-2d67-4151-a63d-fca68069faca">
                        <type>extend</type>
                        <source_id>49458bd9-5d57-4cac-a0d7-ad351b888f25</source_id>
                        <destination_id>fb3f6a23-72b0-4f6c-8d90-b195df346f70</destination_id>
                </connection>
        <!-- conteúdo omitido -->
        </connections>
</use_case_diagram>
```

Nessa representação, ligações do tipo *association* são usadas para ligar casos de uso a Atores e ligações do tipo *extend* ou *include* são utilizadas para a associação entre casos de uso.

Apesar de, no estado atual desenvolvimento, o editor AMALIA garantir alguns aspetos de coerência entre modelos de domínio e casos de uso, ao só permitir a associação de casos de uso a entidades existentes no modelo de domínio, existe ainda um longo caminho de desenvolvimento da ferramenta para garantir o rigor e a eliminação de ambiguidades através de um mecanismo eficiente de validação e de implementação de restrições. Por exemplo atualmente o editor não garante que apenas seja possível associar duas entidades a um caso de uso se existir um relacionamento entre elas. Outro exemplo tem a ver com o facto de não ser possível impor restrições aos atributos de uma entidade. Por

este motivo, a validade dos modelos produzidos e o seu rigor são responsabilidades do modelador. Além disto a incapacidade para impor restrições aos atributos, e a impossibilidade de definir atributos calculados, entre outros aspetos, impossibilita a geração completa de aplicações através de um processo de transformação puramente automatizado.

4.3 Importação dos Modelos Gerados Pela Ferramenta Amália e a sua Representação em Memória

A ferramenta de geração de código desenvolvida no âmbito deste trabalho de projeto de mestrado, partindo do output em XML de um modelo gerado no AMALIA, gera em memória duas representações do mesmo. Uma correspondente ao modelo de domínio e outra para o modelo de casos de uso. Em memória o modelo de domínio (DM) corresponde a uma coleção de objetos da classe ClassRepresentation e outra de objetos ClassConnections. Cada um desses objetos corresponde respetivamente a uma entidade definida através do tag <element> ou a uma associação, definida através do tag <connetion>. As classes usadas para fazer a representação do DM relacionam-se da forma representada na figura 9.

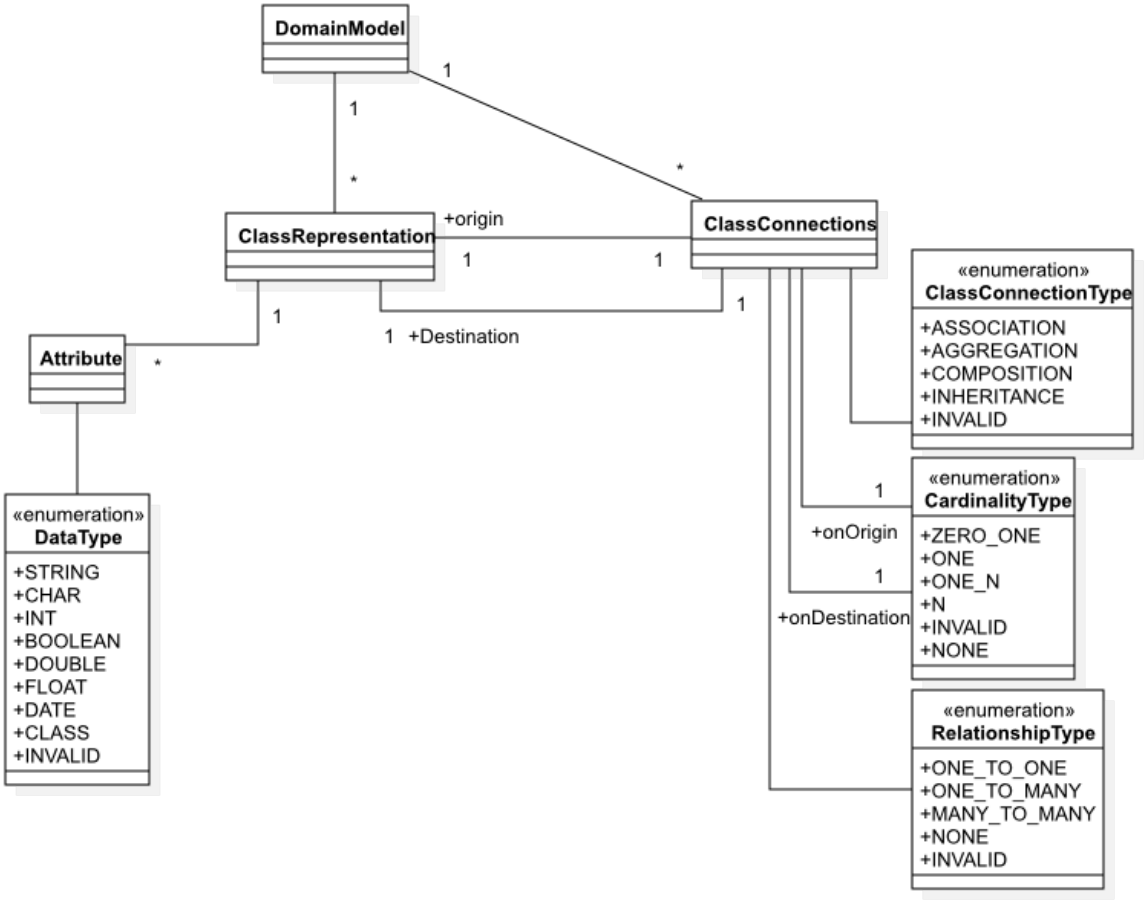

Figura 9: Representação da estrutura em memória do modelo de domínio, depois da sua importação a partir do ficheiro XML produzido pelo editor de modelos AMALIA

Este processo é desencadeado na instanciação da classe que irá representar o modelo de domínio (DomainModel) tal como apresentado na listagem 5.

Listagem 5: Construtor da classe DomainModel evidenciando a leitura do XML para a instanciação da lista de entidades do modelo de domínio e das respetivas ligações.

```
public DomainModel (AmaliaFileReader reader)
                                        throws ParserConfigurationException, SAXException,
IOException{
    modelClasses = reader.getDomainClassesList();
    allModelConnections = reader.getClassConnectionsList();

    //restante código omitido
    }
```

A estrutura corresponde assim a uma tradução quase literal do ficheiro XML do AMALIA. Assim, cada tag <element> do tipo classe, do modelo de domínio, é transformado numa instância de ClassRepresentation, e cada um dos seus atributos dá origem a uma instância de Attribute, onde é guardada a informação relativamente ao nome, tipo de dados e visibilidade. Por sua vez as relações entre as entidades do modelo de domínio dão origem a uma lista de instâncias de ClassConnection. Cada instância guarda a informação relativa à origem e destino da relação, ao tipo de ligação, à cardinalidade em cada um dos extremos e ao tipo de relação entre as entidades (RelationshipType). Esta última informação, apesar de redundante, por resultar diretamente da informação da cardinalidade dos extremos, é guardada por facilitar a leitura do modelo no momento da geração do suporte para a persistência de dados e futuras transformações.
A leitura do modelo de Casos de Uso dá origem a uma representação em memória com a estrutura do diagrama da figura 10.

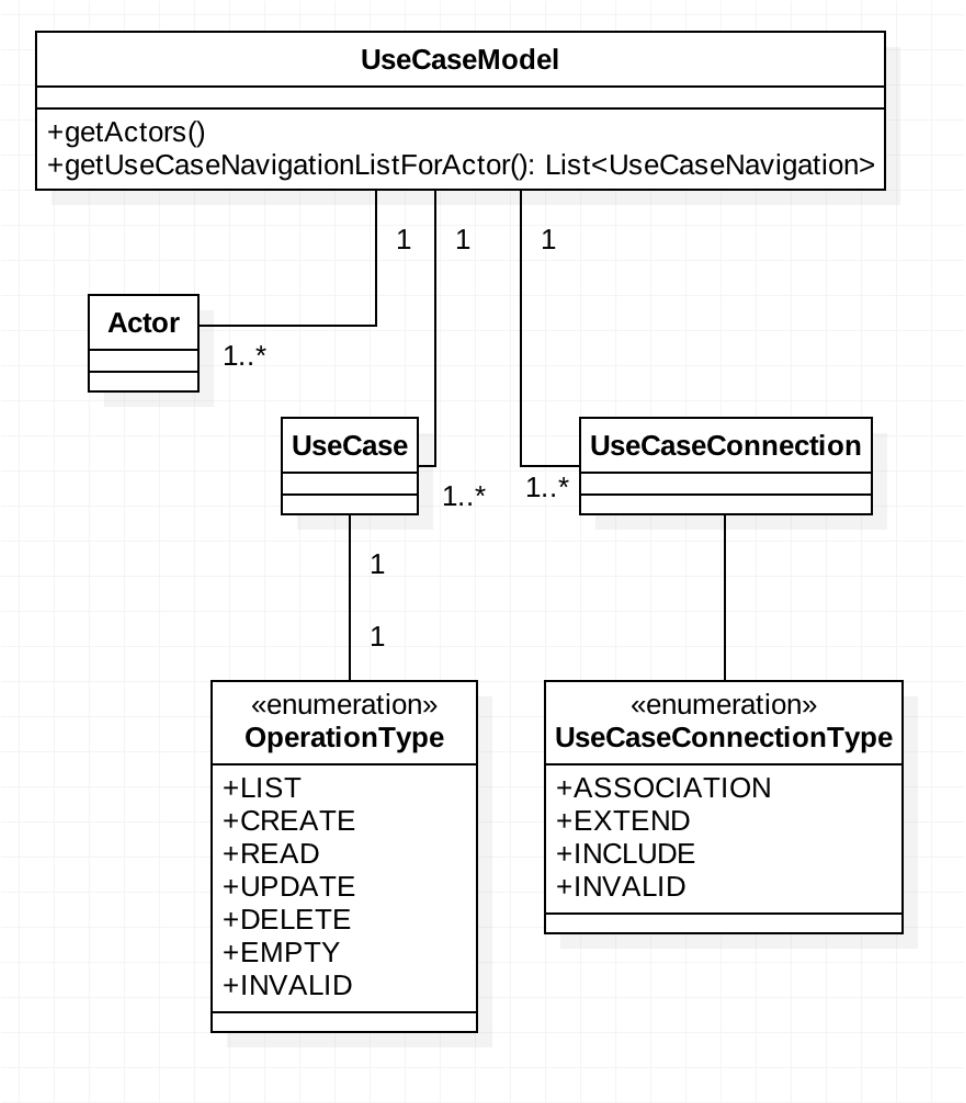

Figura 10: Representação da estrutura da representação, em memória do modelo, de casos de uso depois da sua importação a partir do ficheiro XML produzido pelo editor de modelos AMALIA.

Desta feita não se trata da simples tradução do ficheiro XML AMALIA para um conjunto de objetos em Java. Aqui, os tag <element> do modelo de casos de uso são decompostos em duas listas. Uma de instâncias de Actor, para cada um dos <element> de tipo actor, e outra de instâncias de UseCase, para cada um dos <element> de tipo use case. Em cada instância de UseCase é guardada informação relativa ao nome do caso de uso, à operação a realizar (OperationType) e à respetiva entidade do modelo de domínio. Se o caso de uso corresponder a uma operação realizada sobre uma entidade relacionada com outra (master entity) essa informação é igualmente guardada na respetiva instância de UseCase. As relações entre os Atores e casos de uso, bem como as relações entre os casos de uso dão origem a instâncias de UseCaseConnection. As entidades relacionadas, bem como o tipo de associação, são aí guardadas. A geração desta representação é feita de forma equivalente à da representação do DM, através da associação de um leitor do XML do AMALIA a uma instância da classe UseCaseModel.

Para efeitos de futura transformação a informação contida no modelo de casos de uso é disponibilizada, através de uma lista de árvores, que representam os casos de uso associados a cada um dos atores. Ao percorrer cada uma dessas árvores será possível gerar um PIM (UI Model) de uma aplicação correspondente a cada perfil de utilizador, modelado por um Ator. Cada uma das árvores tem a estrutura representada da figura 11.

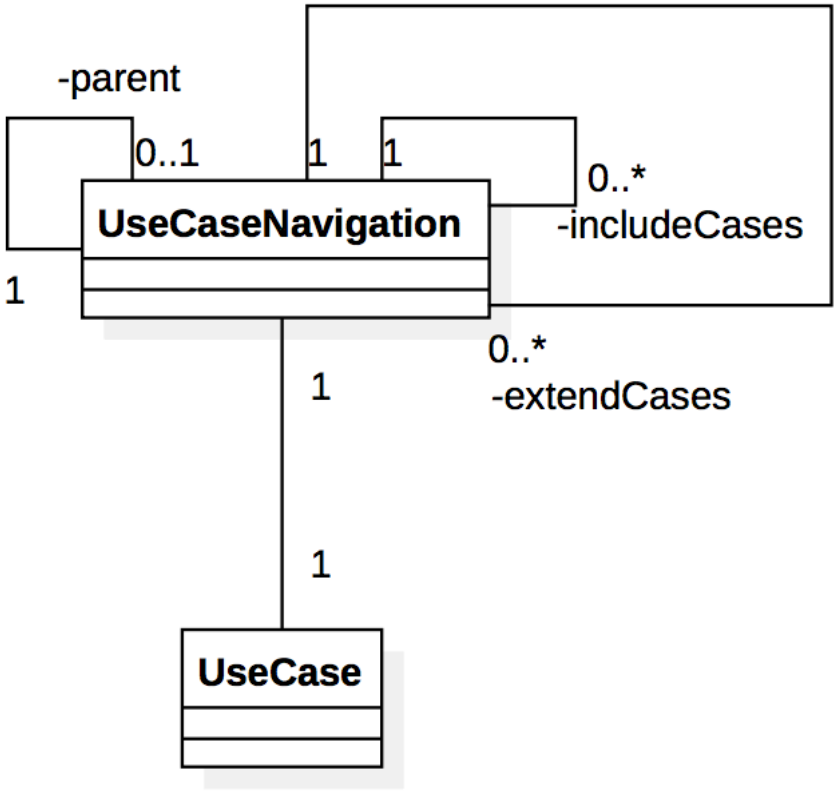

Figura 11: Representação da estrutura de através da qual o modelo de casos de uso é disponibilizado para as seguintes operações de transformação. Através dela o encadeamento dos casos de uso é representado sob a forma de uma árvore que facilita a sua leitura na produção dos novos artefactos.

Nesta estrutura, a classe UseCaseNavigation é um involucro para uma instância de UseCase e que permite navegar através dos casos de uso que lhe estão associados. A geração destas árvores é feita no momento de produção do UI Model evocando o método apresentado na listagem 6.

Listagem 6: Método evocado para a obtenção da lista de árvores que representam os casos de uso associados a um Ator. Através dele é possível obter os casos de uso que representam as ações disponíveis para um Ator bem como as relações entre elas, com vista à sua transformação num UI Model.

```
public List<UseCaseNavigation> getUseCaseNavigationListForActor(Actor actor ){
    List <UseCaseNavigation> useCaseNavList = new ArrayList<>();

    addUseCasesAssotiatedToActor(useCaseNavList , actor);
```

```
        return useCaseNavList;

    }

private void addUseCasesAssotiatedToActor(List<UseCaseNavigation> useCaseNavList, Actor actor) {

        String actorId = actor.getId();

        List<UseCaseConnection> associationConnectionsToActor = connections.stream()
                .filter((con) -> con.getType() == UseCaseConnectionType.ASSOCIATION
                                        && con.getDestination().equals(actorId) )
                .collect( toList() );

        for (UseCaseConnection con: associationConnectionsToActor ){
            UseCase useCase = useCases.stream()
                .filter( (uc) -> uc.getId().equals( con.getOrigin() ) )
                .findFirst().get();
            UseCaseNavigation nav = new UseCaseNavigation (useCase);
            useCaseNavList.add( nav );
            addLinkedUseCases (useCase , nav, UseCaseConnectionType.EXTEND);
            addLinkedUseCases (useCase , nav, UseCaseConnectionType.INCLUDE);
        }
    }
```

4.4 Transformação dos Modelos de Domínio e de Casos de Uso, em Memória, num Modelo da Interface com o Utilizador Independente da Plataforma (PIM)

Usando as representações em memória dos Modelos de Domínio e de Casos de Uso é produzido um modelo da interface com o utilizador Independente da plataforma (PIM – UI Model). Este processo é feito em memória e dele resulta a estrutura apresentada na Figura 12. Através dela os espaços de interação da interface com o utilizador, da aplicação modelada, representados pela classe View são encadeados numa estrutura em árvore através de instâncias da classe Navigation. Esta permite representar as possibilidades de navegação entre os diferentes espaços de interação da aplicação.

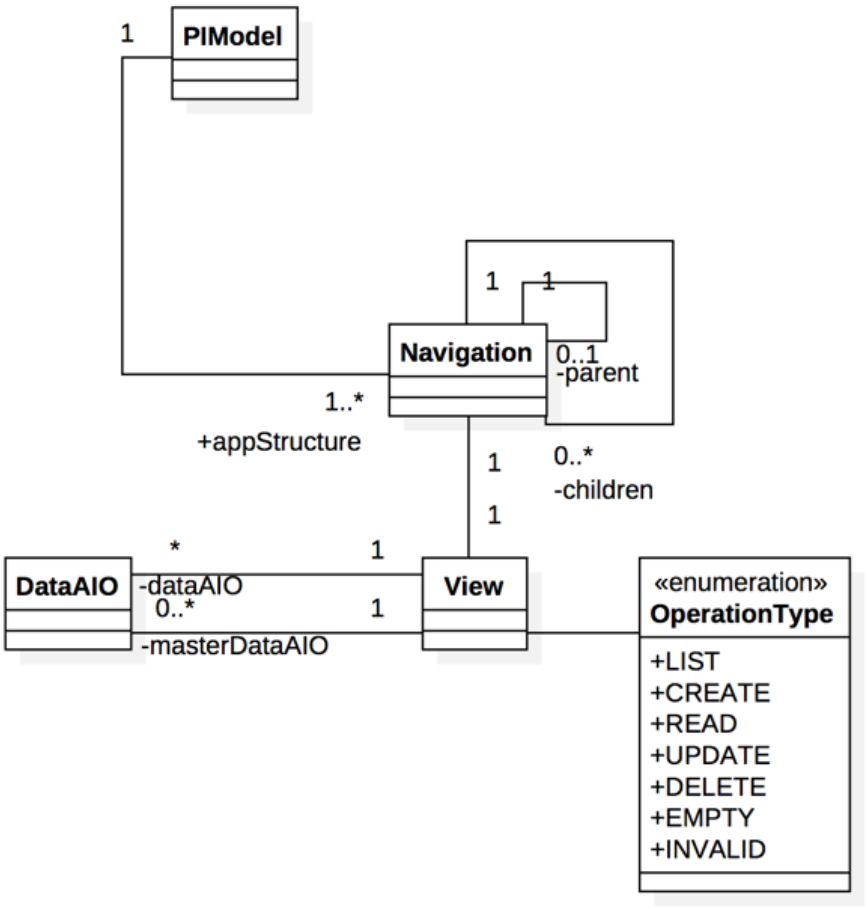

Figura 12: Estrutura da representação, em memória do PIM, gerado a partir do modelo de casos de uso e de domínio. A classe Navigation permite representar as possibilidades de navegação entre os diferentes espaços de interação da aplicação que são representados por View.

A produção desta estrutura faz-se percorrendo as árvores de casos de uso associadas a cada Actor. Ao percorre-las, cada instância de UseCase origina uma instância de View, que neste trabalho simplifica e agrega num só os diferentes espaços de interação definidos em (Cruz, 2010; Cruz & Faria, 2010; Cruz, 2015). A cada um destes espaços de interação está associado o nome e a operação definidos para o caso de uso. As propriedades (atributos) das entidades do modelo de domínio associadas ao caso de uso originam DataAIO, numa correspondência direta de um para um, cada um deles representando um espaço de visualização ou manipulação da respetiva propriedade. O espaço de interação View permite guardar dois conjuntos de DataAIO, tantos quantas as entidades que podem ser associadas a um caso de uso, podendo desta forma representar espaços de interação correspondentes aos definidos como ViewRelatedEntity e ViewRelatedList em (Cruz, 2010; Cruz & Faria, 2010; Cruz, 2015; Silva, Paiva, & Cruz, 2016).

A cada View é associada uma OperationType, correspondente a uma operação sobre a entidade do domínio associada ao caso de uso e que têm correspondência com as operações CRUD ou de listagem (*read all*) e que dessa forma são equivalentes à definição de CRUDop definidas em (Cruz, 2010; Cruz & Faria, 2010; Cruz, 2015; Silva, Paiva, & Cruz, 2016).

Concluído o processo obtém-se a Lista appStructure, a qual reúne os diferentes pontos de entrada na aplicação. Cada um desses pontos de entrada corresponde a um perfil de utilizador definido por um Actor no modelo de casos de uso. A implementação deste processo é apresentada na listagem 7.

Listagem 7: geração do PIM – UI Model como uma lista de árvores de objetos Navigation a partir do modelo de domínio e de casos de uso. Cada árvore de objetos Navigation corresponde ao conjunto de ações possíveis para um dado utilizador, de acordo com o modelado no modelo de casos de uso.

```
public static PIModel buildModel (DomainModel dm , UseCaseModel uc){
      PIModel pim = new PIModel();
      pim.buildAppStructure(dm,uc);
      return pim;
   }

private void buildAppStructure(DomainModel dm, UseCaseModel uc) {
      this.appStructure = new ArrayList<>();
       addRootNavigationsNodes (dm, uc);
   }

   private void addRootNavigationsNodes(DomainModel dm, UseCaseModel uc) {
      List<Actor> actors = uc.getActors();
      actors.stream().forEach( actor -> {
         Navigation nav = new Navigation (new View (actor.getName() ) );
         appStructure.add( nav  );
         addViewsFromUseCaseNavigations(nav, dm ,uc.getUseCaseNavigationListForActor(actor));
      });
   }
```

A partir da estrutura do PIM da interface com o utilizador, são produzidas duas representações da aplicação, para cada Ator (modelos específicos da platraforma - PSM). Uma dessas representações corresponde à definição dos layouts dos ecrãs de uma aplicação Android e dos respetivos menus. A outra corresponde a um conjunto de ficheiros em java que permitem detetar as ações do utilizador e realizar operações sobre um modelo de dados em SQLite. Este último é produzido por uma transformação a partir do modelo de domínio.

4.5 Transformação do Modelo Independente da Plataforma (PIM) da Interface com o Utilizador num conjunto de Layouts para a plataforma Android

Na transformação do PIM da interface com o utilizador num conjunto de layouts para a plataforma Android, cada View origina um documento XML representado por uma instância de org.w3c.dom.Document, especificando um layout. As instâncias de View com OperationType.CREATE, OperationType.READ, OperationType.UPDATE ou OperationType.DELETE originam um documento XML cujo elemento de topo é um

<ScrollView>, no interior do qual é colocado um elemento correspondente a um <LinearLayout>. Dentre deste, cada DataAIO origina um elemento específico em função do tipo de dados e da OperationType associada à View. Assim, por exemplo, para o caso de uso criar atleta (ver figura 7), associado à entidade Athlete e relacionada com a entidade Sport (ver figura 6) os componentes do layout gerado são um ScrollView, que permitirá aceder a campos do formulário que possam ficar fora dos limites do ecrã do dispositivo. No interior deste elemento surge um LinearLayout com orientação vertical onde são colocados todos elementos que permitem a manipulação da informação relativa à entidade Athlete (figura 13)

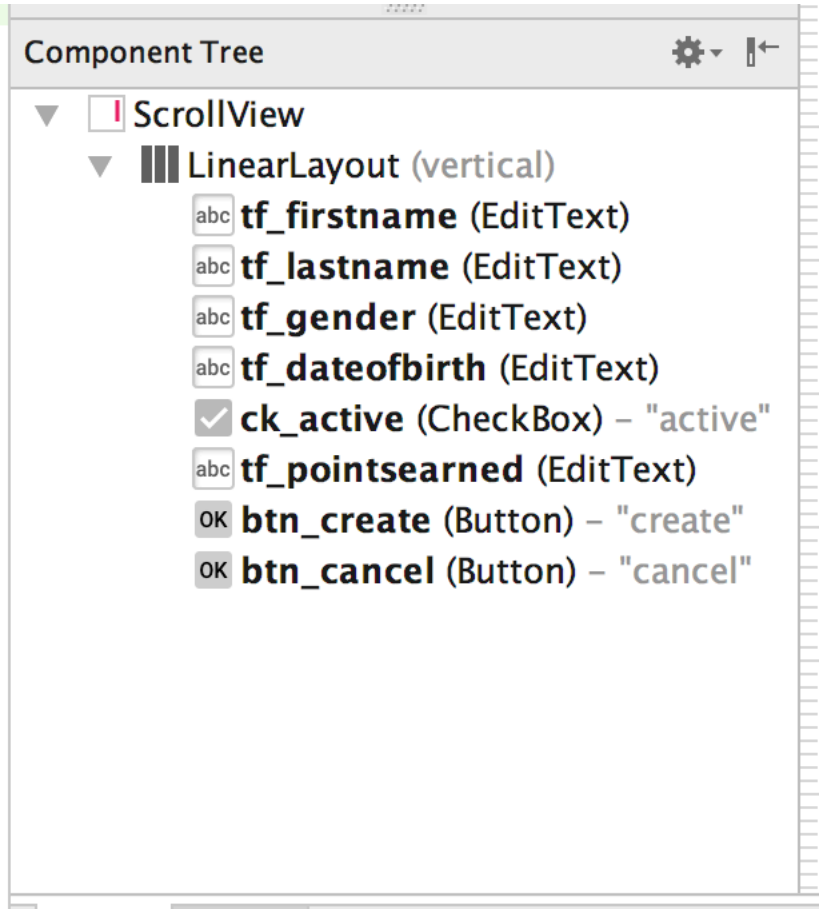

Figura 13 : Imagem do Android Studio evidenciando a estrutura do layout gerado para a View correspondente ao caso de uso Create Athelete. Nele um ScrollView permite aceder a elementos que possam ficam ocultos fora do espaço disponível no ecrã do dispositivo. O ScrollView contem um LinearLayout que apresenta verticalmente todos os elementos do formulário. Estes são restringidos em função da operação e tipo de dados.

Nos casos de OperationType.CREATE ou OperationType.UPDATE, os DataAIO correspondentes a tipos de dados string, int, float, double ou Date originam elementos <EditText> que são limitados pelo atributo **android:inputType** nos casos dos tipos int, float, double ou Date. Por sua vez, DataAIO correspondentes ao tipo boolean originam elementos <CheckBox>.

Nos casos de OperationType.READ ou OperationType.DELETE, os DataAIO correspondentes a tipos de dados string, int, float, double ou Date originam elementos

<TextView>. Para o tipo boolean geram-se elementos <CheckBox> limitados pelo atributo **android:clickable="false"** impedindo-se assim a possibilidade de edição.
As operações de criação, leitura, atualização, apagar e cancelar originam elementos <Button>. A operação cancelar é adicionada quando as View contêm operações criar, atualizar ou apagar.
No caso de OperationType.List, o elemento de topo é <RelativeLayout> dentro do qual são colocados dois elementos. Um <ListView>, que representa a lista a ser apresentada ao utilizador, e um <TextView>, que representa a informação a apresentar ao utilizador quando não existem dados para povoar a lista (figura 14).

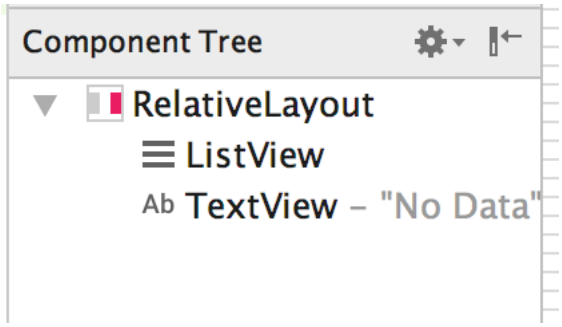

Figura 14 : Imagem do Android Studio evidenciando a estrutura do layout gerado para a View correspondente ao caso de Uso List Sports, contendo um RelativeLayout povoado com uma ListView e um TextView.

São também produzidos layouts para cada elemento da lista contendo um elemento <TextView> para cada DataAIO da View, no entanto, estes layouts não são ainda utilizados na aplicação gerada. Estes permitiriam mostrar nas listas mais informação do que um simples elemento de texto, usado atualmente, para identificar cada um dos elementos da lista.
A navegação na aplicação é representada pelas associações entre objetos Navigation do PIM da interface com o utilizador. Assim, a partir dos elementos ligados como filhos de uma instância de Navigation geram-se layouts de menus. Os layouts produzidos desta forma correspondem aos espaços de interação através dos quais o utilizador poderá manipular o modelo, que originou a base de dados definida em SQLite.

4.6 Transformação do Modelo de Domínio num conjunto de Ficheiros Java responsáveis pela Base de Dados SQLite na plataforma Android

A geração da base de dados SQLite resulta da transformação do modelo de domínio pela seguinte estrutura (figura 15).

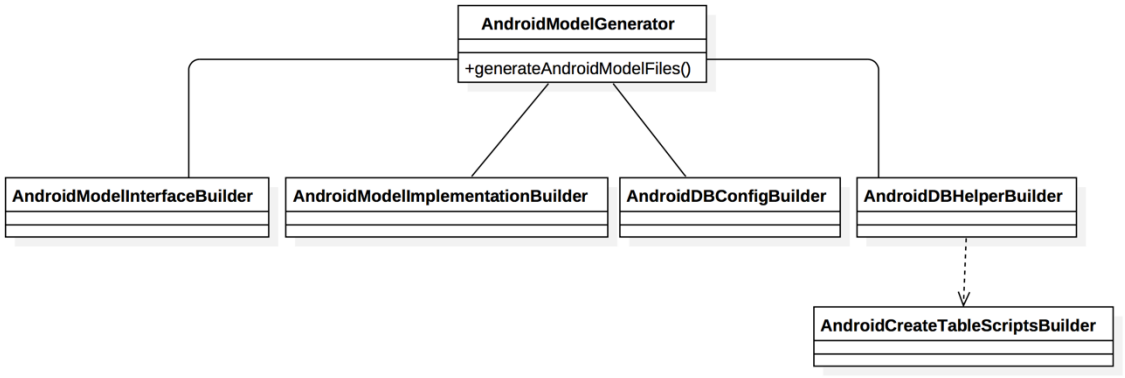

Figura 15: Estrutura do motor de transformação do modelo de domínio numa base de dados SQLite.

AndroidModelGenerator recebe a representação em memória do modelo de domínio e irá produzir uma base de dados SQLite. O produto final da transformação tem a estrutura representada no seguinte modelo (figura 16).

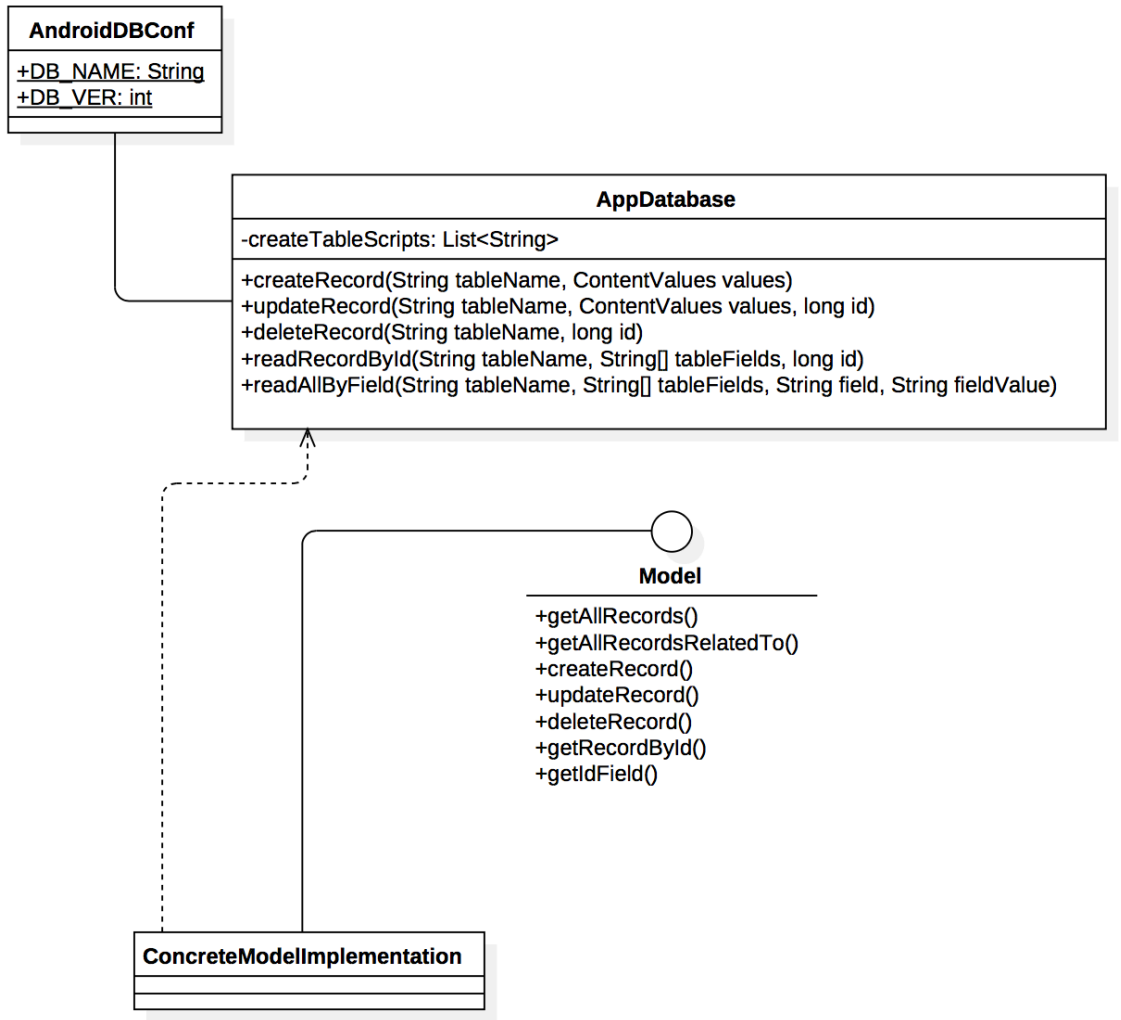

Figura 16: Estrutura dos artefactos produzidos pela transformação do modelo de domínio numa base de dados SQLite. A classe AndroidDBConf representa dados de configuração da base de dados. A classe AppDatabase herda de SQLiteOpenHelper e proporciona a interface para as operações com a base de dados. A classe

ConcreteModelImplementation uma das várias classes geradas a partir do modelo de domínio, representado cada uma das suas entidades. Cada uma dessas classes implementa a mesma interface que uniformiza a comunicação com as diferentes Activities geradas.

A classe AndroidDBConf é gerada de forma independente de qualquer um dos modelos anteriormente definidos, e é apenas um repositório de informação para a criação da base de dados SQLite, como o nome e a versão. No estado atual do trabalho, o nome será sempre appDatabase.db e a versão será sempre 1 (listagem 8).

Listagem 8: A classe AndroidDBConf gerada para conter dados de conficuração da base de dados. Desta forma estes elementos podem ser facilmente modificados depois da geração do código.

```
public class AndroidDBConf {
  public static final String DB_NAME ="AppDatabase.db";
  public static final int DB_VER = 1;
  }
```

A classe AppDatabase é uma classe que herda de SQLiteOpenHelper e disponibiliza os métodos necessários para as operações CRUD e de listagem. Estes são gerados de forma abstrata e por isso são gerados de forma independente dos modelos criados no AMALIA. Todavia, é usada informação do modelo de domínio para gerar os scripts de criação das tabelas da base de dados. Estes são gerados por uma instância de AndroidCreateTableScriptsBuilder com base no modelo. Esses scripts são passados a uma instância de AndroidDBHelperBuilder que é responsável por gerar a classe AppDatabase (listagem 9).

Listagem 9 : Apresentação parcial do código gerado para a classe AppDatabase evidenciando os scripts de criação da base de dados e os métodos de acesso a dados utilizados pelos diferentes modelos. A integridade referencial das chaves estrangeiras não é imposta devido à impossibilidade de prever a ordem pela qual cada um dos scripts de criação das tabelas é produzido.

```
public class AppDatabase extends SQLiteOpenHelper {

  private static final List<String> createTableScripts = Arrays.asList(
  "CREATE TABLE event (_id INTEGER PRIMARY KEY, name TEXT, date TEXT, location TEXT, maximumage
INTEGER, minimumage INTEGER, id_sport INTEGER);",
  "CREATE TABLE call (_id INTEGER PRIMARY KEY, calldate TEXT, participated INTEGER, id_athlete INTEGER,
id_event INTEGER);",
  "CREATE TABLE sport (_id INTEGER PRIMARY KEY, name TEXT);",
  "CREATE TABLE athlete (_id INTEGER PRIMARY KEY, firstname TEXT, lastname TEXT, gender TEXT,
dateofbirth TEXT, active INTEGER, pointsearned INTEGER, id_sport INTEGER);"
  );
//código omitido

  public long createRecord (String tableName , ContentValues values){
    SQLiteDatabase db = getWritableDatabase();
    return db.insert(tableName, null, values);
    }
```

```
   public int updateRecord (String tableName , ContentValues values , long id){
      SQLiteDatabase db = getWritableDatabase();
      return db.update(tableName , values , "_id = ?" , new String[]{Long.toString(id)} );
      }

   public int deleteRecord (String tableName , long id){
      SQLiteDatabase db = getWritableDatabase();
      return db.delete( tableName , "_id = ?" , new String[]{Long.toString(id)} );
      }

   public Cursor readRecordById (String tableName, String[] tableFields, long id){
      SQLiteDatabase db = getReadableDatabase();
      return db.query(tableName , tableFields , "_id = ?" , new String[]{Long.toString(id)} , null , null , null ,
null );
      }

   public Cursor readAllByField ( String tableName , String[] tableFields , String field , String fieldValue ){
      SQLiteDatabase db = getReadableDatabase();
      if (field != null && fieldValue != null){
         return db.query( tableName , tableFields , field+"=?" , new String[]{fieldValue} , null , null , null , null
);
         }else{
            return db.query(tableName,tableFields,null,null, null , null , null , null );
            }
         }

      private void createTables(SQLiteDatabase db){
         for(String createTableScript : createTableScripts){
            db.execSQL(createTableScript);
            }
         }

      }
```

É igualmente gerada uma interface que define todos os métodos que os modelos gerados irão implementar. Como as operações que é possível fazer sobre os dados estão pré-definidas na ferramenta AMALIA todos os modelos irão implementar versões das mesmas que diferem apenas em detalhes específicos que podem ser passados como argumento de métodos com a mesma assinatura. Assim, uma instância de AndroidModelInterfaceBuilder gera a interface Model sem necessidade de qualquer informação do modelo de domínio (listagem 10).

Listagem 10 : Interface Model gerada de acordo com as operações possíveis na ferramenta AMALIA. Os detalhes específicos de cada modelo são tidos em conta na geração e tem em conta a informação de cada uma das entidades do modelo de domínio.

```
public interface Model{

   public Cursor getAllRecords();
   public Cursor getAllRecordsRelatedTo(String field , long id);
   public void createRecord( ContentValues values );
   public void updateRecord( ContentValues values, long id );
   public void deleteRecord( long id );
   public ContentValues getRecordById( long id );
   public String getIdField();

   }
```

As implementações concretas da interface, representadas por ConcreteModelImplementation, são geradas por diferentes instâncias de AndroidModelImplementationBuilder, e são feitas sob orientação da informação de cada uma das entidades do modelo de domínio, que permitem definir elementos como o nome da tabela e os respetivos campos (listagem 11).

Listagem 11: Exemplo do código gerado para o modelo Athlete, destacando os aspetos que o diferenciam dos restantes modelos e que são derivados da respetiva entidade do modelo de domínio.

```
public class Athlete implements Model {

   private final String tableName = "athlete";

   private final List<String> columnNames =
Arrays.asList("_id","firstname","lastname","gender","dateofbirth","active","pointsearned","id_sport");
   private final String idField = "firstname";
   private final AppDatabase db;

   //código omitido

   @Override
   public Cursor getAllRecords(){
      return db.readAllByField(tableName,columnNames.toArray(new
String[columnNames.size()]),null,null);
      }
//código omitido
   @Override
   public ContentValues getRecordById( long id ){
      Cursor result = db.readRecordById ( tableName ,columnNames.toArray(new
String[columnNames.size()]), id);
      result.moveToFirst();
      //Build the ContentValues
      ContentValues cv = new ContentValues( columnNames.size() );
```

```
    for (String column : columnNames){
      cv.put(column,result.getString(result.getColumnIndex( column ) ) );
      }
    result.close();
    //And return it

    return cv;
    }
  //código omitido
  }
```

A geração da interface Model, a que todas as entidades da base de dados SQLite obedecem, facilita a geração das classes Activity ou ListActivity que as utilizarão para implementar as operações CRUD da aplicação, uma vez que as linhas de código a gerar para fazer a evocação dos métodos são, em muitos aspetos, iguais em todas elas.

4.7. Transformação do Modelo Independente da Plataforma (PIM) num conjunto de Ficheiros Java responsáveis pela Funcionalidade na plataforma Android

A implementação da funcionalidade de uma aplicação Android depende de instâncias da classe Activity ou de classes que herdam dela. Estas podem implementar uma série de métodos que governam o ciclo de vida da Activity ou a implementação de funcionalidades específicas como, por exemplo, a criação de menus.
Para a aplicação são gerados dois tipos de ficheiros. Ficheiros java que herdam de Activity, para a implementação das interações relacionadas com as operações criar, ler, atualizar e apagar (CRUD), e ficheiros que herdam de ListActivity, para as interações relacionadas com a operação listar. Na geração dos ficheiros que herdam de Activity é utilizado o motor de transformação representado pela estrutura da figura 17.

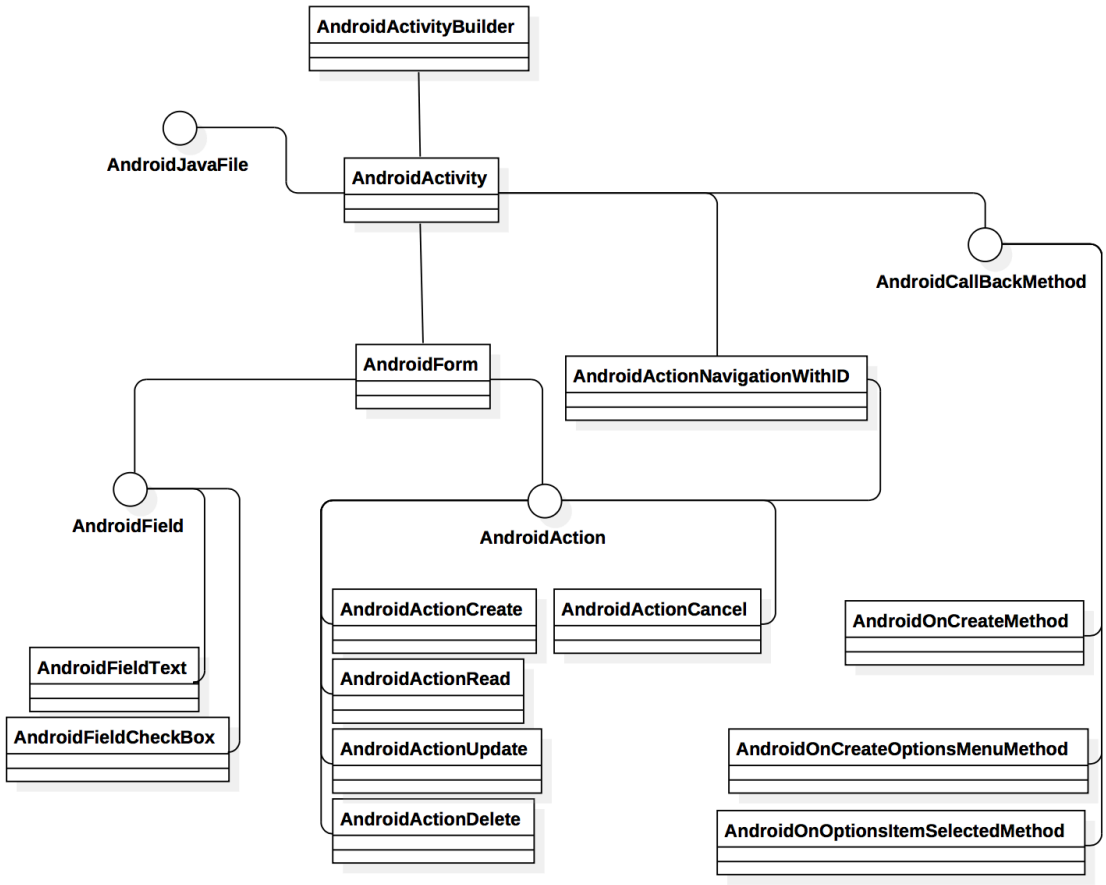

Figura 17: Estrutura da parte do motor de transformação que permite produzir o código java das Activities da aplicação Android. A classe AndroidActivity representa um ficheiro de uma Activity na plataforma Android e recolhe informações do PIM para organizar um formulário constituído por diversos campos representando os atributos de uma entidade do domínio, sobre a qual é possível uma operação (AndroidAction). A classe AndroidActivity recolhe igualmente no PIM informação que lhe permite definir as possibilidades de navegação para outras Activities e a definição dos métodos de callback que governam o seu ciclo de vida e a implementação de menus.

Durante o processo, uma instância de AndroidActivityBuilder recebe uma instância de Navigation do PIM da interface com o utilizador. Tendo esse objeto uma View com uma das operações CRUD, é criada uma instância de AndroidActivity que é responsável pela geração das linhas de código que serão escritas para um ficheiro java. Por esse motivo a classe AndroidActivity implementa uma interface AndroidJavaFile, cuja única razão de ser é forçar a existência de um método que facilita a produção de um ficheiro java (listagem12).

Listagem 12: estrutura básica da classe AndroidActivity evidenciando a sequencia de operações necessárias à produção do código de uma Activity. São apresentados também os métodos que a implementação da interface AndroidJavaFile impõem. Estes destinam-se à obtenção das linhas de código geradas e à obtenção de referências aos nomes da Activity e respetivo layout.

```
public class AndroidActivity implements AndroidJavaFile {

   private final String name;
   private final String layoutName;
```

```
    private final AndroidForm form;

    private final List<AndroidCallBackMethod> callBacks;
    private final List<AndroidAction> actions;

    private List<String> imports;

    public AndroidActivity (Navigation nav){
        this.name = Code.activityName(nav.getName());
        this.layoutName = Code.layoutName(nav.getName());
        this.form = new AndroidForm ( nav.getView() );
        // isto tem que ser reprensado
        this.callBacks = generateCallBacks(nav);
        this.actions = buildActions (nav);
        this.imports = new ArrayList<>();
        this.imports.add("import android.app.Activity;");
        this.imports.add("import android.os.Bundle;");
        this.imports.addAll( this.form.getImports() );
    }

    @Override
    public String getName(){
        return this.name;
}

    @Override
    public String getLayoutName(){
        return this.layoutName;
    }

    @Override
    public List<String> getCode(){
        List<String> code;
        code = new ArrayList<>();
        code.addAll(getImports());
        code.addAll( getSeparator() );
        //All other imports here
        code.addAll( activityCode() );
        return code;
    }
//código omitido
}
```

Ao gerar uma Activity estamos a criar um controlador para um formulário a cujos elementos é necessário aceder (listagem13). A representação desse formulário fica a cargo de uma instância de AndroidForm. Para a sua construção é necessária informação contida na instância de View associada ao objeto Navigation. Assim Um AndroidForm irá agregar instâncias que implementam a interface AndroidField, que representa um campo

do formulário. Por esse motivo cada instância de DataAIO (lista dataAIO) origina uma implementação específica dessa interface. As instâncias com tipos de dados string, int, float, double e Date originam instâncias de AndroidFieldText, ao passo que as instâncias com tipo de dados boolean originam instâncias de AndroidFieldCheckBox. Um AndroidForm tem também associada uma operação CRUD. Estas operações são representadas por instâncias de classes que implementam a interface AndroidAction e são implementadas respetivamente nas classes AndroidActionCreate, AndroidActionRead, AndroidActionUpdate e AndroidActionDelete. Tal como referido anteriormente, quando foi abordada a geração dos layouts, uma operação Cancel, não definida nos modelos, é associada aos espaços de interação associados às operações criar, atualizar ou apagar. Essa operação representa a possibilidade de interromper o processo e regressar ao espaço de interação anterior, e é representada pela classe AndroidActionCancel. Estas instâncias de AndroidForm, AndroidField e AndroidAction, descritas, são responsáveis pela geração do código necessário para instanciar as variáveis que permitem acesso aos elementos do layout, como por exemplo os TextView ou EditText, a modificação do seu conteúdo e a implementação das funcionalidades correspondentes aos botões da interface. Todavia, a legibilidade do ficheiro produzido leva a que o código seja segregado em métodos que serão chamados no momento determinado pelo ciclo de vida da Activity (ver secção 2.3.4). Este ciclo é governado por chamadas a métodos específicos. A geração desses métodos é da responsabilidade de um conjunto de classes que implementam a interface AndroidCallBackMethod. Um desses métodos é o onCreate() que é gerado por uma instância de AndroidOnCreateMethod, gerada no seio de AndroidActivity, com informação contida da instância de AndroidForm associada à mesma. Desta forma, consegue-se um método onCreate() mais legível e com as operações segregadas em métodos que podem ser usados noutros momentos do ciclo de vida da Activity.

O objeto Navigation, do PIM da interface com o utilizador, passado à instância de AndroidActivityBuilder, contém Navigation filhos que representam possibilidades de navegação a partir de um dado ponto na aplicação. Esta funcionalidade corresponde a uma ação não relacionada com o formulário e por isso é implementada como um AndroidAction independente do AndroidForm. Neste caso cada filho originará uma instância de AndroidActionNavigationWithID. Este é responsável pela geração do código necessário pela implementação de uma chamada de uma nova Activity, com passagem de uma referência do ponto de origem. De novo a implementação desta funcionalidade num ficheiro java depende da implementação de métodos de callback que implementam menus e controlam as ações da seleção das diferentes opções. Assim a geração desses métodos é determinada pela mesma informação que é usada para gerar as instâncias de AndroidActionNavigationWithID .

Listagem 13 : Exemplo de uma Activity gerada a partir do objeto Navigation contendo a View resultante do caso de uso Update Athlete. Destaca-se a relação entre os campos do formulário editáveis e a operação Update, a sua inicialização no método onCreate(). A associação da Activity ao respetivo modelo – Athlete -, bem como a definição botões que controlam as ações possíveis. Todo o código relativo a estas operações foi segregado em métodos privados facilitando a sua alteração após a operação de geração.

```
public class Update_athleteActivity extends Activity {
   private EditText tf_firstname;
   private EditText tf_lastname;
   private EditText tf_gender;
   private EditText tf_dateofbirth;
   private CheckBox ck_active;
   private EditText tf_pointsearned;
   private Model model;
   private long id;


   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.update_athlete_layout);

      // form field assignments
      setFormFieldsAssignments();
      //Model assignments
      model = new Athlete( getApplicationContext() );

      //form actions set up function calls
      readRecord();


      //form actions function calls
      makeUpdateButton();
      makeCancelButton();


      }

//Form Action Methods

   private void makeUpdateButton(){
      ((Button)findViewById(R.id.btn_update)).setOnClickListener(new OnClickListener(){
         @Override
         public void onClick(View v) {

            //implementation code
            ContentValues cv = new ContentValues();
            cv.put( "firstname" , tf_firstname.getText().toString());
            cv.put( "lastname" , tf_lastname.getText().toString());
            cv.put( "gender" , tf_gender.getText().toString());
```

```
        cv.put( "dateofbirth" , tf_dateofbirth.getText().toString());
        cv.put( "active" , ck_active.isChecked()? 1 : 0 );
        cv.put( "pointsearned" , tf_pointsearned.getText().toString());
         model.updateRecord( cv , id );
        finish();
        }
    });
    }

private void makeCancelButton(){
    ((Button)findViewById(R.id.btn_cancel)).setOnClickListener(new OnClickListener(){
        @Override
        public void onClick(View v) {
            finish();

            }

        });
        }
    //Form field declaration functions
    private void setFormFieldsAssignments(){

        tf_firstname = (EditText)findViewById(R.id.tf_firstname);
        tf_lastname = (EditText)findViewById(R.id.tf_lastname);
        tf_gender = (EditText)findViewById(R.id.tf_gender);
        tf_dateofbirth = (EditText)findViewById(R.id.tf_dateofbirth);
        ck_active = (CheckBox)findViewById(R.id.ck_active);
        tf_pointsearned = (EditText)findViewById(R.id.tf_pointsearned);
        }
    //Form actions funtions
    private void readRecord(){
        ContentValues cv;
        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            id = extras.getLong("id");
            cv = model.getRecordById(id);
            }else{
                cv = new ContentValues();
                }
            if (cv.size() > 0){
                tf_firstname.setText(cv.getAsString ("firstname") );
                tf_lastname.setText(cv.getAsString ("lastname") );
                tf_gender.setText(cv.getAsString ("gender") );
                tf_dateofbirth.setText(cv.getAsString ("dateofbirth") );
                ck_active.setChecked(cv.getAsInteger("active") == 1 );
                tf_pointsearned.setText(cv.getAsString ("pointsearned") );
                }
            }
           }
```

Quando a operação associada à instância de View, incluída no objeto Navigation passado ao gerador AndoidActivityBuilder, corresponde a listar, será produzido um ficheiro de uma classe java que herda de ListActivity. A sua produção é da responsabilidade de uma instância de AndroidListActivity que implementa a interface AndoidJavaFile. Neste caso o gerador pode ser representado pelo diagrama da figura 18.

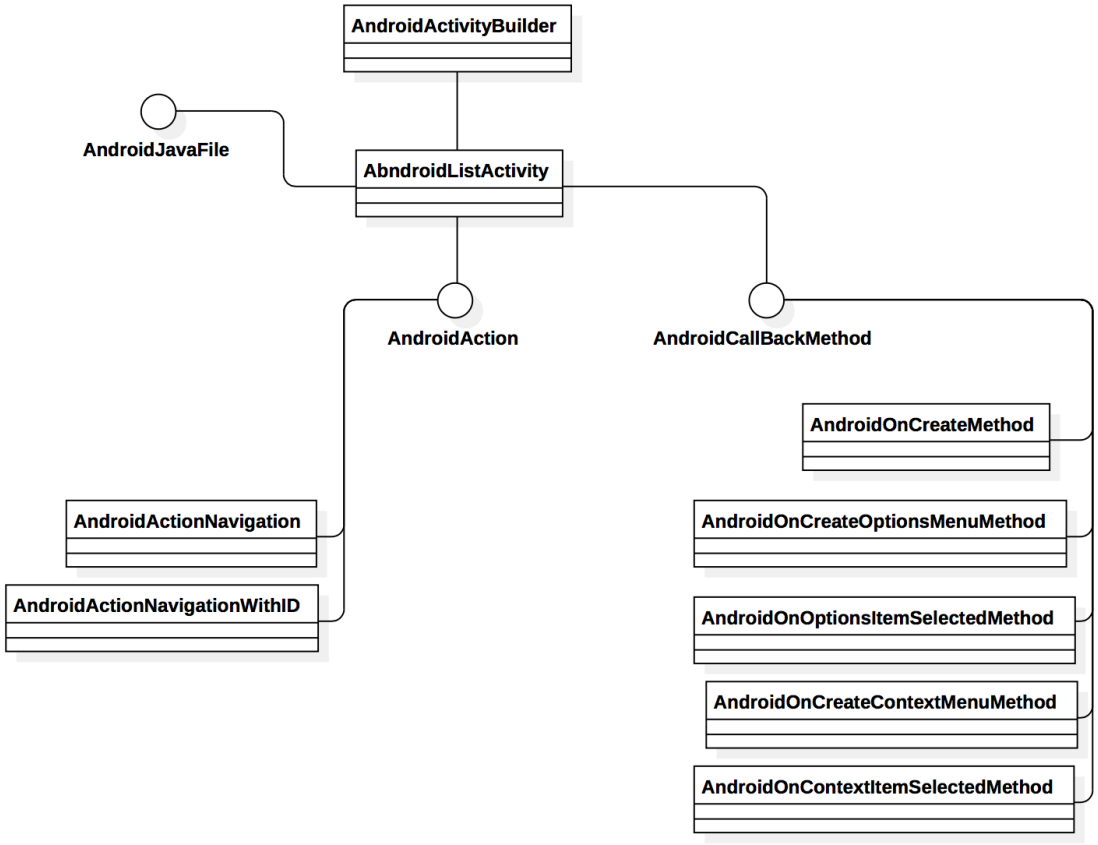

Figura 18: Representação do motor de estrutura da parte do motor de transformação que permite produzir o código java das ListActivities da aplicação Android. A classe AndroidListActivity representa o ficheiro java de uma ListActivity da plataforma android. Esta utiliza informação do PIM para definir ações de navegação entre possíveis espaços de interação e para gerar os métodos de callback responsáveis pela gestão do seu ciclo de vida e da produção e gestão de menus.

A responsabilidade de geração do código necessário para a inicialização da lista é de uma instância de AndroidOnCreateMethod e de AndroidEntity. O código necessário à implementação do método onResume() é da responsabilidade de AndroidOnResumeMethod e de AndroidEntity

Uma instância de Navigation pode estar ligada a instâncias filhas, que representam opções de navegação. Estas podem ser possibilidades de navegação relacionadas com um elemento específico da lista ou independentes de qualquer elemento da mesma. Assim, existem dois tipos de ações a navegação, com passagem de uma referência relativa à origem ou a navegação sem qualquer referência. Estas duas possibilidades são implementadas através de implementações de AndroidAction distintos. A navegação associada a um item da lista implica a implementação de menus de contexto, cujos layouts

foram gerados anteriormente, e cujo controlo depende da geração de código para os métodos onCreateContextMenu() e onContextItemSelected() específicos do Android. A geração dessas porções de código é da responsabilidade de instâncias AndroidOnCreateContextMenuMethod e AndroidOnContextItemSelected() que usam informação do objeto Navigation e dependem de código produzido pela AndroidActionNavigationWithID. Por sua vez a navegação para operações não associadas a um elemento específico da lista (por exemplo a criação de um novo elemento a inserir na lista) está dependente de menus de opções para os quais foram gerados os respetivos layouts. A utilização desses layouts depende da geração de código para o método onCreateOptionsMenu() e a implementação da sua funcionalidade necessita da geração do código para o método onOptionsItemSelected(). A geração do código destes métodos é da responsabilidade de instâncias de AndroidOnCreateOptionsMenuMethod e AndroidOnOptionsItemSelectedMethod gerados com informação passada no objeto Navigations. Esse código está dependente de um método gerado pelo AndroidActionNavigaton (listagem 14).

Listagem 14 : Exemplo de uma ListActivity gerada a partir do objeto Navigation contendo a View resultante do caso de uso List Sports. Destaca-se a associação da ListActivity ao respetivo modelo, a inicialização da lista, de um OnItemClickListener e o registo para um menu de contexto no método onCreate(). O código para a inicialização da lista foi segregado num método próprio. No que toca às possibilidades de navegação a partir desta ListActivity destaça-se a associação da navegação para as Activities responsáveis pelo update, delete e read a partir de um menu de contexto, ao passo que a navegação para a Activity responsável pela criação de um novo desporto está associado a um menu de opções. Finalmente destaca-se que a navegação para novas Activities é feita através chamadas diferentes do método launchActivityForClass() que se distinguem pelo número de parâmetros e cuja evocação depende da necessidade de passar informação para a Activity que vai ser lançada.

```
public class List_sportsActivity extends ListActivity {

   private Model model;
   private long id;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.list_sports_layout);

      model = new Sport( getApplicationContext() );
      setupList();
      getListView().setOnItemClickListener( new AdapterView.OnItemClickListener(){
         @Override
         public void onItemClick(AdapterView<?> fonte, View arg1,
         int posicao, long arg3) {
            Cursor cursor = (Cursor) fonte.getItemAtPosition(posicao);
            long id = cursor.getLong(cursor.getColumnIndex("_id"));
            launchActivityForClass( Read_sportActivity.class, id );

            }
```

```
        });

        registerForContextMenu( getListView() );

        }

    @Override
    public void onResume(){
        super.onResume();
        setupList();
        }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo
menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);

        AdapterView.AdapterContextMenuInfo info = ( AdapterView.AdapterContextMenuInfo )menuInfo;
        menu.setHeaderTitle( "Options " );

        MenuInflater inflater = this.getMenuInflater();
        inflater.inflate(R.menu.context_list_sports, menu);
        }

    @Override
    public boolean onContextItemSelected(MenuItem item) {

        AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo)
item.getMenuInfo();

        switch (item.getItemId()) {
            case R.id.read_sport:
            launchActivityForClass(Read_sportActivity.class , info.id );
            return true;
            case R.id.update_sport:
            launchActivityForClass(Update_sportActivity.class , info.id );
            return true;
            case R.id.delete_sport:
            launchActivityForClass(Delete_sportActivity.class , info.id );
            return true;
            default:
            return super.onContextItemSelected(item);
            }
        }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.list_sports_menu, menu);
```

```
        return true;
        }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.create_sport:
            launchActivityForClass(Create_sportActivity.class);
            return true;
            default:
            return super.onOptionsItemSelected(item);
            }
        }
    private void launchActivityForClass( Class activityClass , long id){

        Intent intent = new Intent(getApplicationContext(), activityClass);
        intent.putExtra("id",id);
        startActivity(intent);

        }
    private void launchActivityForClass( Class activityClass ){

        Intent intent = new Intent(getApplicationContext(), activityClass);
        startActivity(intent);

        }
    private void setupList(){
        Cursor cursor = model.getAllRecords();
        String[] from = new String[]{ model.getIdField() };
        int[] to = new int[]{android.R.id.text1};
        getListView().setAdapter(new
SimpleCursorAdapter(this,android.R.layout.simple_list_item_1,cursor,from,to,SimpleCursorAdapter.FLAG
_REGISTER_CONTENT_OBSERVER));
        }
    }
```

4.8 CONCLUSÕES

A produção de uma aplicação para plataformas Android a partir de modelos de caso de uso e de domínio revela-se possível. A partir destes é possível derivar padrões que correspondem a estruturas específicas da interface de utilizador da plataforma e a operações CRUD. Os modelos utilizados permitem recolher elementos suficientes para definir as possibilidades de navegação entre Activities e possibilitar a construção dos métodos que governam os respetivos ciclos de vida. Todavia as características dos modelos que atualmente é possível construir com o editor AMALIA permitem um nível

elevado de ambiguidade pelo que as aplicações geradas devem apenas ser encaradas como protótipos, cuja principal utilidade pode ser o levantamento de requisitos ou a recolha de elementos que permitem identificar fragilidades do editor e dos processos de transformação aqui descritos.

5. Discussão dos Resultados do Trabalho

5.1 Introdução

No presente estádio de desenvolvimento o processo aqui apresentado tem já alguns aspetos positivos, mas tem ainda numerosas limitações. Algumas destas derivam da ferramenta usada para produzir os modelos que serão objeto de transformação, outras resultam da própria implementação atual do mecanismo de transformação. Neste capítulo procura-se inventariar aspetos positivos e negativos associados á globalidade do processo, desde a geração dos modelos até à geração da aplicação final e suas características.

5.2 Vantagens do Editor AMALIA para a elaboração de modelos

O editor AMALIA foi desenvolvido no sentido de permitir a realização integrada de modelos de domínio e de casos de uso, usando para o efeito diagramas de classes e de casos de uso. A utilização deste tipo de diagramas tem a vantagem mobilizar o poder do UML. Esta é uma linguagem amplamente conhecida e definida de forma rigorosa pelo que essas propriedades podem ser benéficas para os modelos produzidos e para a facilidade com que faz a sua construção e comunicação. A integração, ainda que incompleta, entre os dois modelos permite garantir que os casos de uso são sempre detidos por entidades do modelo de domínio.

A definição dos casos de uso exige sempre ou suporte exterior ao do diagrama de casos de uso. O editor AMALIA disponibiliza uma ferramenta para que essa definição seja feita para um conjunto restrito de operações pré-definidas. Essas são operações comuns em aplicações baseadas em dados e formulários e correspondem às operações CRUD e listagem. Esta pode ser entendida como um caso especial da operação de leitura (Read) em que todos os registos são apresentados. Estas operações podem ser limitadas por relacionamentos entre entidades, no entanto, é responsabilidade do modelador garantir que o relacionamento existe no modelo de domínio, pois o editor não faz essa verificação. A dependência de operações simples e pré-definidas retira complexidade aos modelos, pelo que não é necessário um grande conhecimento técnico para a sua produção, estando esta ao alcance mesmo de um modelador com um modesto conhecimento do UML.

5.3 Limitações da versão do Editor AMALIA utilizada para a elaboração de modelos.

Um dos problemas conhecidos da versão utilizada do editor AMALIA é que a garantia de consistência entre modelos não é ainda perfeita. Assim é possível associar a um dado caso de uso duas entidades que não têm qualquer relacionamento no modelo de domínio. O editor assume que garantir essa consistência é responsabilidade do modelador. No

entanto, essa assunção permite a criação de modelos inconsistentes cuja transformação irá conduzir a aplicações não funcionais.
A definição dos atributos é feita segundo o UML e permite a definição da sua visibilidade, nome e tipo de dados. No entanto a consistência e respeito pelas regras do UML não é verificado. Assim ao definir um atributo é possível a introdução de texto arbitrário, o que conduzirá a um processo de transformação com erros, que impossibilitarão a produção da aplicação desejada.
A ferramenta também não possibilita a definição de restrições sobre os atributos, como por exemplo a definição de restrições simples como a definição de valores máximos e mínimos para atributos numéricos ou a limitação de um conjunto de valores possíveis. Deste facto resulta que na aplicação gerada a implementação dessas limitações e verificações terá de ser feita manualmente, no código, por um *developer* com conhecimentos de programação para a plataforma android.
Tal como não permite definir restrições sobre os atributos também não permite definir atributos calculados a partir de outros valores. Por exemplo a ferramenta não permite definir que o valor total de uma encomenda corresponde à soma do valor de todos os itens que a compõem. Esta limitação conduz à geração de aplicações que são incapazes que realizar cálculos esperados, e que por isso, exigem a introdução manual, nos formulários de todos os valores para todos os atributos. Desta forma perde-se uma vantagem da utilização de um dispositivo computacional, porque a aplicação gerada nunca realizará algumas das tarefas que comummente se assume como garantidas nas aplicações associadas a este tipo de dispositivos, pois é atualmente inaceitável para todas as partes interessadas de uma aplicação comercial que seja pedido ao utilizador para calcular e introduzir o valor total de uma encomenda num formulário.
Por outro lado, a restrição das operações que é possível associar aos casos de uso introduzem um preço significativo a pagar pela simplicidade da sua definição. Um grande conjunto de operações comuns em aplicações baseadas em dados não são possíveis por transformação direta dos modelos. Por exemplo, numa aplicação que faça a gestão de um clube desportivo, não seria possível associar atletas a uma convocatória pela simples seleção a partir de uma lista.
Com um conjunto de limitações como o apresentado as aplicações geradas a partir dos modelos do editor AMALIA têm de ser consideradas incompletas e apenas podem ser consideradas protótipos.

5.4 Vantagens do processo proposto para a geração de aplicações

Assumindo que as aplicações geradas não podem merecer outra classificação que a de protótipo, existem, nesta fase de desenvolvimento do projeto, algumas vantagens no processo apresentado.
O processo de geração dos layouts e código a partir dos modelos é rápido, decorre consistentemente em tempos de poucos minutos. Isto permite a rápida produção de protótipos funcionais cuja funcionalidade pode ser testada, discutida e modificada, no

decorrer de uma simples reunião com as partes interessadas. As limitações destas produções mostrarão claramente que não se está perante um produto final pelo que a discussão terá obrigatoriamente que se centrar no esclarecimento das funcionalidades pretendidas e nas restrições que é necessário impor.
A geração rápida de bases de dados, mesmo que rudimentares, permitirá também discutir e testar diferentes modelos de dados para futura aplicação, o que pode contribuir para identificar a solução que da melhor forma satisfaz as partes interessadas.
A discussão e teste dos requisitos da futura aplicação pode exigir a rápida modificação do protótipo, para algumas alterações este processo tem que ser feita diretamente no código gerado. Esta necessidade fez com que houvesse o cuidado de produzir código e layouts facilmente legíveis e que pudessem ser inseridos num projeto no android studio e dessa forma fossem facilmente alteráveis por um *developer*. Estes projetos modificados podem ser preservados documentando funcionalidades pretendidas ou alterações desejadas.
A *user interface* é derivada com base nos modelos de domínio e de casos de uso, sem recurso à modelação explicita da mesma através de uma ferramenta dedicada. Isto facilita a modelação, porque reduz objetivamente o tempo necessário para a produção de um modelo transformável e porque não exige a aprendizagem de uma nova linguagem de modelação que seria necessária, porque o UML não permite a modelação direta da *user interface*. Deste facto resulta que as *user interfaces* criadas a partir de uma dada entidade do domínio associada a um dado caso de uso, podem apresentar todos os seus atributos e este comportamento pode não ser o desejável. Assim numa fase relativamente inicial do processo de desenvolvimento podem ser identificadas e discutidas as limitações que é necessário impor à visibilidade dos atributos para os diferentes perfis de utilizador.
O processo de produção dos modelos e da sua transformação em aplicações é feita com uso de tecnologia multiplataforma, html e javascript para o editor AMALIA e java para o processo de transformação dos modelos em aplicações, pelo que pode facilmente ser levado a cabo em qualquer dispositivo em que seja possível fazer desenvolvimento para android, ou seja, nos principais sistemas operativos para computadores pessoais (Windows, MacOS e Linux).

5.5 Limitações da Ferramenta apresentada para a geração de aplicações

A ferramenta proposta para a geração de aplicações a partir de modelos de domínio e de casos de uso expressos em ficheiros XML, depende exclusivamente do editor AMALIA. Esta dependência propaga para o processo de transformação modelos que ainda são incompletos e nos quais ainda existe ambiguidade. Essa limitação poderia ser corrigida por alguma possibilidade de manipulação e restrição do modelo na aplicação que vai fazer a transformação, mas tal ainda não é possível pelo que, tal como já foi referido, o produto final é incompleto. Por outro lado, o processo de transformação não faz uso de toda a semântica do UML, nomeadamente das associações *include* entre casos de uso. A sua modelação é possível no editor AMALIA.

No desenvolvimento deste projeto as ligações de tipo *include* entre os casos de uso são tratadas de igual forma às ligações de tipo *extend*, não respeitando por isso a semântica do UML. Esta opção resulta do facto de no presente estado de desenvolvimento do projeto de as aplicações geradas apenas implementarem operações CRUD e de listagem atómicas. Desta forma não se implementam comportamentos complexos que exigissem uma composição obrigatória de casos de uso. Todavia o editor AMALIA permite a modelação da ligação de tipo *include* pelo que se optou pela sua indiferenciação relativamente ao extend, comportamento que pode ser modificado em futuras evoluções do projeto.

O processo apresentado é unidirecional, dos modelos para a aplicação final. Esta unidirecionalidade implica que qualquer alteração realizada ao nível da aplicação, no android studio não pode ser convertida nos modelos. Assim estes começam a perder rapidamente sincronia com o seu produto sempre que um *developer* altera o código gerado. A solução pode ser a reconstrução dos modelos no editor AMALIA para refletir as alterações efetuadas, o que pode não ser um processo moroso ou complexo, mas que consome algum tempo e em muitas situações pode não ser de todo possível devido às características já enunciadas da ferramenta.

Atualmente a ferramenta exige a transferência manual dos ficheiros gerados para um projeto existente no android studio, pois depende deste para gerar o apk da aplicação. Este processo apesar de não ser complexo exige algum conhecimento da organização dos projetos do android studio e sendo um processo manual introduz alguma morosidade e possibilidade de erro.

A persistência de dados resultante do processo de transformação é feita numa base de dados SQLite, no dispositivo. Estas podem servir para efeitos de demonstração de um protótipo, mas são desadequadas para uma aplicação que pode servir diferentes perfis de utilizador em diferentes dispositivos, pois cada um teria acesso apenas aos dados por si gerados. Por outro lado, na iteração atual, é produzida uma aplicação diferente para cada Actor do modelo de casos de uso e os respetivos casos de uso, o que implica que se instaladas no mesmo dispositivo duas aplicações correspondentes a Actores diferentes, não haverá partilha de dados entre as duas pois o acesso à base de dados SQLite é vedado a todas as aplicações exceto à que a produziu. A alteração destes aspetos no código no android studio implica uma modificação significativa da aplicação, o que por sua vez elimina as vantagens de um desenvolvimento orientado por modelos com geração automática de código. Finalmente é conveniente frisar que neste aspeto, o código gerado para a implementação da base de dados é simples e desadequado a qualquer implementação fora do âmbito de um protótipo funcional.

O conjunto de todas as limitações apresentadas mostra claramente que a ferramenta apresentada não está de todo preparada para a produção de aplicações finais e que estas têm ainda que ser implementadas fazendo uso do android studio ou através de abordagens de transformação de modelos alternativas. Neste aspeto é de notar que o tempo necessário para produzir, no android studio, aplicações equivalentes às geradas

pela abordagem apresentada não deve ser muito grande, principalmente quando o developer já tem um conjunto de projetos anteriores equivalentes que possam ser modificados. A diferença de tempo para a produção de um protótipo, entre as duas possibilidades não foi avaliada, mas admite-se que abordagem aqui proposta tenha alguma vantagem, mesmo incluindo o tempo de produção dos modelos, geração do ficheiro XML, importação do mesmo na ferramenta de transformação, criação do projeto no android studio e migração manual dos ficheiros produzidos.

5.6 Conclusões

O processo aqui apresentado encontra-se num estádio de desenvolvimento que ainda não permite a sua utilização para a produção de uma aplicação a disponibilizar a um utilizador final. A origem das limitações encontradas é diversa, se algumas se propagam da ferramenta escolhida para a produção dos modelos usados, existem várias que resultaram da forma como o processo de transformação foi implementado e das opções feitas quanto à persistência de dados.

6. Conclusões e Trabalho Futuro

A correção das fragilidades identificadas é o caminho para o futuro desenvolvimento do processo descritos neste relatório. A integração das duas ferramentas numa mesma plataforma permitiria eliminar alguns desses problemas, em primeiro lugar porque eliminaria a necessidade de manter duas aplicações distintas, que partilham dados através de um ficheiro XML. A capacidade de importação de modelos poderia ser preservada desde que a sua especificação estivesse rigorosamente definida.

A possibilidade de definição de restrições sobre os elementos de ambos os modelos é fundamental para a produção de uma aplicação final. Assim será desejável que seja possível restringir os atributos das entidades do modelo de domínio, e que na descrição dos casos de uso, seja possível definir quais os atributos das entidades associadas que podem ser visualizados, uma vez que pode não ser adequado que todas os perfis de utilizador tenham um acesso ilimitado.

A alteração da forma de persistência de dados, abrindo a possibilidade de opção por diferentes soluções é fundamental uma vez que a opção por uma base de dados local em SQLite não é a melhor em muitas das aplicações móveis.

A unificação dos diferentes perfis de utilizador numa só aplicação é também uma necessidade, que terá de ser acompanhada pela geração de um mecanismo de autenticação eficiente.

Depois de resolvidos estes problemas e fragilidades a possibilidade de geração de código para diferentes plataformas como por exemplo o iOS é uma possibilidade a explorar.

Referências

Android API Guide, disponível em https://developer.android.com/guide/index.html, consultado em fevereiro 2017

Applause. (n.d.). Retrieved from: https://github.com/applause/applause

Balagtas-Fernandez, F., Tafelmayer, M., & Hussmann, H. (2010). Mobia Modeler: Easing the Creation Process of Mobile Applications for Non-Technical Users. In Proceedings of the 15th international conference on Intelligent user interfaces (IUI'10), (pp. 269-272). Hong Kong, China. doi:10.1145/1719970.1720008

Brambilla , M., Mauri , A., & Umuhoza , E. (2014, Aug). Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End. *Mobile Web Information Systems* .

Cruz, A.M.(2010). Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models. PhD Dissertation, Faculty of engineering, University of Porto, Sep. 2010.

Cruz, A. M. & Faria, J. P. (2010). A Metamodel-based Approach For Automatic User Interface Generation. In Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part 1, LNCS 6394, pp.256-270, Oslo, Norway, October 2010. Springer-Verlag Berlin Heidelberg.

Cruz, A. M. (2014). A Pattern Language for Use Case Modeling. In Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (Modelsward 2014), Lisboa, Portugal, January 2014, INSTICC - Institute for Systems and Technologies of Information, Control and Communication, INSTICC Press.

Cruz, A.M. (2015). Use Case and User Interface Patterns for Data Oriented Applications. In Hammoudi, S., Pires, L.F., Filipe, J., das Neves, R.C. (Eds.) Model-Driven Engineering and Software Development, Second International Conference, MODELSWARD 2014, Lisbon, Portugal, January 7-9, 2014, Revised Selected Papers. Series: Communications in Computer and Information Science, vol. 506, pp. 117-133, Springer International Publishing, Dec. 2015.

Díaz, V. G., Lovelle, J. M., & García-Bustelo, B. C. (2015). Handbook of Research on Innovations in Systems and Software Engineering (2 Volumes) (pp. 1-745). Hershey, PA: IGI Global. doi:10.4018/978-1-4666-6359-6.

Heitkötter, H., Majchrzak, T. A., & Kuchen, H. (2013, March). Cross-platform model-driven development of mobile applications with md 2. *In Proceedings of the 28th Annual ACM Symposium on Applied Computing* , 526-533.

Jézéquel, J.-M. (2005). Model Transformation Techniques. Retrieved January 13, 2015,

from http:// people.irisa.fr/Jean-Marc.Jezequel/enseignement/ModelTransfo.pdf

Kramer, D., Clark, T., & Oussena, S. (2010). MobDSL: A Domain Specific Language for multiple mo- bile platform deployment. In International Conference on Networked Embedded Systems for Enterprise Applications (NESEA). IEEE. doi:10.1109/NESEA.2010.5678062

Limbourg , Q., & Vanderdonckt , J. (2004). USIXML: a User Interface Description Language Supporting Multiple Levels of Independence. *In M. Matera & S. Comai (eds.), ICWE Workshops* , 325-338.

ModAgile. (n.d.). Retrieved from: http://www.modagile-mobile.de

OMG (2015). OMG Unified Modeling Language TM (OMG UML). Version 2.5. Available at: http://www.omg.org/spec/UML/2.5/

Ortiz , G., & Prado , A. G. (2010). Improving device-aware Web services and their mobile clients through an aspect-oriented, model-driven approach. *Information and Software Technology* (52), 1080–1093.

Parada , A. G., & Brisolara , L. B. (2012). A model driven approach for Android applications development. *Brazilian Symposium on Computing System Engineering* , 192-197.

Paternó, F., Santoro, C., & Spano, L. D. (2009). *ACM Transactions on Computer-Human Interaction , 16* (4), 19:1-19:30.

Petrov, I., & Buchmann, A. (2008). Architecture of OMG MOF-based Repository Systems.

In G. Kotsis, D. Taniar, E. Pardede, & I. Khalil (Eds.), Proceedings of the 10th International

Conference on Informa- tion Integration and Web-based Applications & Services (iiWAS

‘08) (pp. 193–200). New York, NY: ACM. http://doi.acm.org/10.1145/1497308.1497346

Porta, D. (2010). Towards Model-driven Development of Mobile Multimodal User Interfaces for Services. In K.-P. Fähnrich & B. Franczyk (eds.), Informatik 2010: Service Science -- Neue Perspektiven für die Informatik, Beiträge *der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Band 1* , 497-502.

Ribeiro , A., & Silva , A. R. (2014). XIS-Mobile: A DSL for Mobile Applications. *In Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)* , 1316-1323.

Silva, J., Paiva, S., & Cruz, A.M. (2016). Model-driven Development of Data-Centered Mobile Applications: A case study for Android. Modern Software Engineering Methodologies for Mobile and Cloud Environments, Cruz, A.M. and Paiva, S. (Eds.), IGI-Global, Jan. 2016.

Sommerville, I. (2011) . Software Engineering 9th Edition. Addison-Wesley

Truyean, F. (2006). The Fast Guide to Model Driven Architecture – The Basics of Model Driven Architecture. Whitepaper, Cephas Consulting Corp. Available at: http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

Vaupel, S., Taentzer, G., Harries, J., Stroh, R., Gerlach, R., & Guckert, M. (2014). Model-Driven Develop- ment of Mobile Applications Allowing Role-Driven Variants. In Model-Driven Engineering Languages and Systems, Proceedings of 17th International Conference, MODELS 2014 (LNCS), (vol. 8767, pp. 1-17). Springer International Publishing. doi:10.1007/978-3-319-11653-2_1

Warmer, J., Bast, W., Pinkley, D., Herrera, M., & Kleppe, A. (2003). MDA Explained - The

Model Driven Architecture: Practice and Promise. Addison-Wesley Professional.