CleanOnFire
Framework de prototipação ágil para aplicações Android

Heitor Gianastasio Pipet de Oliveira[1], Ramon dos Santos Lummertz[2]

[1]Acadêmico do Curso de Curso de Análise e Desenvolvimento de Sistemas – Universidade Luterana do Brasil (ULBRA) Campus Torres - RS

[2]Professor dos Cursos de Computação – Universidade Luterana do Brasil (ULBRA) Campus Torres – RS

heitor.gianastasio@ulbra.inf.br, ramon.lummertz@ulbra.br

Abstract. *This article will introduce the creation and development of a framework for mobile applications for the Android platform called CleanOnFire, whose goal is to make the creation of well-designed and structured mobile applications more agile and efficient. The framework consists of database abstraction, clean architecture structuring, asynchronous tasks execution, and data presentation APIs. The system was developed in the Java programming language, using Android Studio 3.0. The methodology chosen for project planning and execution was based on Scrum, a widely used agile methodology.*

Resumo. *Neste artigo será apresentada a criação e o desenvolvimento de um framework para aplicações móveis para a plataforma Android que recebeu o nome de CleanOnFire, cujo objetivo é tornar a criação de aplicações móveis bem arquitetadas e estruturadas mais ágeis e eficiente. O framework é composto por APIs de abstração de banco de dados, estruturação da arquitetura limpa, execução assíncrona de atividades, e apresentação de dados. O sistema foi desenvolvido na linguagem de programação Java, utilizando o Android Studio 3.0. A metodologia escolhida para planejamento e execução do projeto foi baseada no Scrum, metodologia ágil amplamente difundida.*

1. Introdução

Os dispositivos móveis inteligentes são hoje indiscutivelmente parte importante do cotidiano de um número gigantesco de pessoas no mundo todo. Por essa razão, o número de aplicativos móveis tem crescido exponencialmente com o passar dos anos. Com o crescente número de aplicações, o mercado de aplicativos para dispositivos móveis está cada vez mais exigente quanto a qualidade e disponibilidade dos produtos desenvolvidos.

Sendo assim, as empresas desenvolvedoras de aplicativos, sobretudo as de pequeno ou médio porte, trabalham com prazos cada vez mais curtos e clientes cada vez mais exigentes. E em muitos dos casos o setor que mais sofre com esta pressão é o de

qualidade de *software* (BASRI; OCONNOR, 2017), resultando em códigos ruins que são incorporados ao produto final e por consequência afetando a sua qualidade.

Mas para contornar o problema da produtividade em detrimento da qualidade, recorreu-se à capacidade do paradigma de programação Orientado a Objetos de reutilização de código e componentização (MACHADO, 2017). Várias bibliotecas e componentes, tanto de código aberto[1] como proprietário, que facilitavam o desenvolvimento dos *softwares* foram desenvolvidas.

Neste artigo será detalhado o processo de desenvolvimento de um *framework* que visa facilitar e automatizar certas atividades de um desenvolvedor de aplicações para a plataforma *Android*. O *framework* utilizará técnicas de geração de código fonte e recursos como tipagem genérica, ambos fornecidos pela plataforma de desenvolvimento da linguagem *Java*.

Este artigo foi construído conforme a seguinte estrutura: na seção de número dois aborda-se o referencial teórico que deu embasamento científico ao desenvolvimento do projeto; na seção de terceira três discute-se a metodologia escolhida para o desenvolvimento do projeto; na seção quatro relata-se o processo de pré-planejamento; já na quinta seção, o processo de elaboração no qual se mostra a evolução do desenvolvimento do aplicativo e, por fim, nas seções seis e sete, apresentam-se o pós- planejamento e as considerações finais.

2. Referencial Teórico

A medida que projetos de software crescem e se tornam mais complexos, os problemas estruturais relevados no início das definições de arquitetura e padrões também crescem e, por vezes, se tornam incontroláveis, levando muitos *softwares* à ruína.

Códigos mal escritos são responsáveis por um aumento significativo nos custos de manutenção, que em certas condições podem ser fatais para empresas que dependem da qualidade de seus *softwares*. Tal constatação pode ser apoiada pelo fato de que a maior parte do custo do desenvolvimento de um software vem da manutenção do mesmo, como aponta o Gráfico 1.

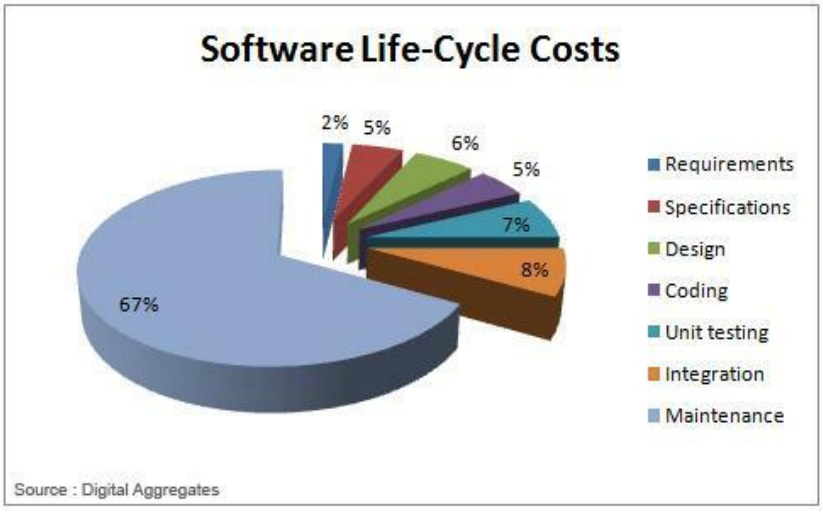


Gráfico 1. Percentual dos custos das fases do ciclo de vida de um *software*

[1] Código aberto é a modalidade de software na qual o código fonte é disponibilizado abertamente à comunidade de desenvolvimento, para utilização livre.

E a esse respeito Martin(2008,p.4) afirma que

> [...] as equipes que trabalharam rapidamente no início de um projeto podem perceber mais tarde que estão indo a passos de tartaruga. Cada alteração feita no código causa uma falha em outras duas ou três partes do mesmo código. Mudança alguma é trivial. Cada adição ou modificação ao sistema exige que restaurações, amarrações e remendos sejam "entendidas" de modo que outras possam ser incluídas. Com o tempo, a bagunça se toma tão grande e profunda que não dá para arrumá-la. Não há absolutamente solução alguma. Conforme a confusão aumenta, a produtividade da equipe diminui, assintoticamente aproximando-se de zero.

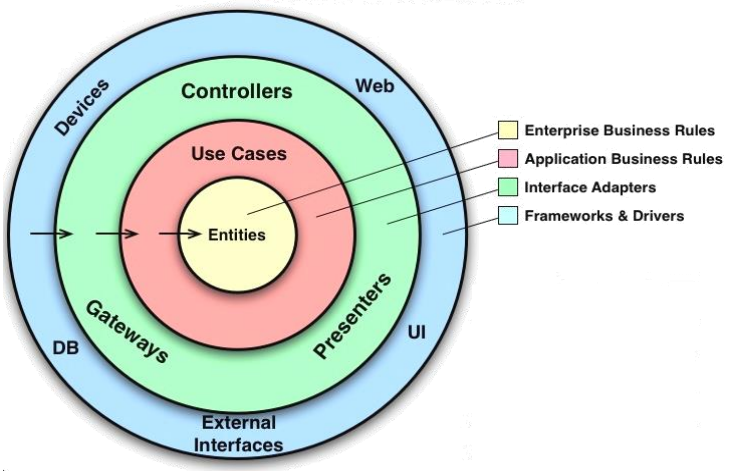
Por tanto é possível afirmar, baseando-se nos estudos de Martin, que a produtividade de uma equipe de desenvolvimento diminui exponencialmente em função da má qualidade do código, inconsistência de sua arquitetura e do tempo de vida do projeto. E é exatamente por essa decrescente produtividade que vários sistemas legados precisam ser completamente reescritos dentro de alguns anos, gerando custos e gastos que podem colocar em risco a viabilidade do projeto.

2.1. Arquitetura Limpa

Segundo Martin (2012), um software é, o que um software faz. O autor salienta que a dependência de SGBDs[2] e frameworks devem ser evitadas pois limitam as possibilidades de expansão do sistema e impõem barreiras ao desenvolvimento de novas funcionalidades.

A partir disso, ele demonstra uma variação da arquitetura hexagonal desenvolvida por Alistair Cockburn (2005), à qual batizou de Arquitetura Limpa (*Clean Architecture*, original). Onde a aplicação deve ser independente de interface gráfica, bancos de dados, dispositivos e todo o resto que não disser respeito estritamente às entidades e regras de negócios da aplicação. A implementação de tal arquitetura traz a maleabilidade e o desacoplamento tanto defendidos por Martin.

O autor ainda reforça o conceito de que uma camada interna da aplicação não pode depender de uma mais externa. Ele chama este conceito de Regra de Dependência. Um esquema básico de como deve ser estruturada uma aplicação com Arquitetura Limpa pode ser observada na figura 1.



[2] SGBD é a sigla para Sistema Gerenciador de Banco de Dados

Figura 1. Representação gráfica da Regra de Dependência

Um dos principais objetivos do *framework* é tornar a implementação da Arquitetura Lima mais simples e menos burocrática nas aplicações desenvolvidas para *Android*, tornando-a aplicável inclusive na fase de prototipação das aplicações. Montando assim uma base de código-fonte coerente e extensível para implementações incrementais.

2.2. Softwares similares

No processo de conceptualização do projeto foram encontrados softwares com funcionalidades semelhantes àquelas idealizadas primeiramente. No entanto nenhum deles contemplavam todas as funcionalidades desejadas.

Aqueles que mais se enquadravam na proposta do *framework* foram as bibliotecas *EasyMVP*[3], *Mosby*[4] e *ThirdInch*[5], no entanto, duas outras bibliotecas se mostraram muito similares com módulos específicos do framework.

Uma delas é o *Room*[6], desenvolvido pela própria *Google* no segundo semestre de 2017, o qual implementa uma abstração do acesso ao banco de dados por meio do processamento de anotações, tal qual o módulo CleanOnFireDB proposto para o *framework* aqui apresentado.

A outra biblioteca é o *AutoAdapter* [7]desenvolvida por Mohamed Hamdan, que se propõe a gerar o código fonte referente aos Adapters do componente *RecyclerView* por meio do processamento de anotações, porém sendo menos versátil do que a solução proposta neste artigo. A comparação por funcionalidade entre as essas bibliotecas e CleanOnFire está representada em forma de tabela nos anexos 1, 2 e 3.

2.2.1. EasyMVP

Desenvolvido pela *6thSolution*, o *EasyMVP* utiliza técnicas de processamento de anotações e *bytecode weaving* (ORACLE CORPORATION, 2017a) para facilitar a estruturação da arquitetura MVP dentro de aplicações *Android*, também utilizando conceitos da Arquitetura Limpa (MARTIN,2012).

Apesar de ser muito poderosa, a biblioteca é pouco versátil atrelando a utilização de alguns de seus componentes ao uso de outras bibliotecas.

2.2.2. Mosby

Desenvolvido por Hannes Dorfmann, o *Mosby* tem como propósito, assim como o *EasyMVP*, facilitar a estruturação de uma aplicação *Android* conforme a arquitetura MVP. Dentre as soluções encontradas, esta foi a mais completa e versátil quanto à sua implementação, mas ainda não incorpora todos os recursos desejados para o *framework* descrito aqui.

2.2.3. ThirdyInch

Desenvolvida pela empresa alemã Grandcentrix GmbH, esta biblioteca adiciona *presenters* para *activities* e *fragments*. Ela facilita o padrão do presenter, onde ele

[3] Link direto para a página da biblioteca: https://github.com/6thsolution/EasyMVP

[4] Link direto para a página da biblioteca: http://hannesdorfmann.com/mosby

[5] Link direto para a página da biblioteca: https://github.com/grandcentrix/ThirtyInch

[6] Link direto para a página da biblioteca: https://developer.android.com/topic/libraries/architecture/room.html

[7] Link direto para a página da biblioteca: https://github.com/mnayef95/AutoAdapter

sobrevive a mudanças de configuração do dispositivo. No entanto também não possui todas as APIs [8]desejadas para este projeto.

3. Metodologia

A metodologia deste projeto baseou-se no *Scrum*, uma das mais difundidas metodologias ágeis no mercado de software. Ela conta práticas, instruções e ferramentas que auxiliam a elaboração dos ciclos de atividades, chamados de *sprints*.

As *sprints* são ciclos iterativos que duram de 15 a 30 dias e apresentam ao final desse prazo um incremento ao sistema. Cada ciclo funciona como um pequeno projeto dentro do projeto principal, com o objetivo de gerar uma parte executável e estável, ainda que incompleta, para que haja assim uma resposta mais instantânea.

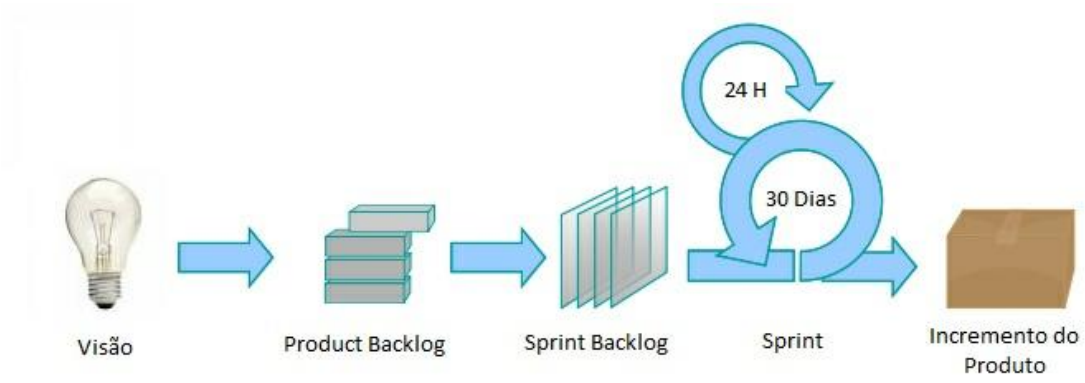


Figura 2. Ciclo de vida do *Scrum*

O *Scrum* visa reduzir os riscos na concretização do projeto já que conta com o cliente presente em maior parte do desenvolvimento do projeto e, além disso, busca a redução do desperdício, orientando o time de desenvolvimento a fazer somente o simples e necessário (SABBAGH, 2013). A metodologia permite que os membros envolvidos no projeto assumam as responsabilidades que melhor lhes couberem, dando-lhes capacidade de melhor controle sobre suas funções. Esses papéis são exibidos na Tabela 1.

Tabela 1. Papéis do *Scrum*

Papéis do *Scrum*	
Product Owner	**Responsável pelo retorno do investimento de projeto. Sua função é garantir que o produto entregue ao cliente atenda às suas necessidades.**
***Scrum* Master**	**Responsável por guiar o Team, estabelecer vínculo entre o *Product Owner* e a equipe e comprometer que todos sigam as diretrizes do *Scrum*.**
Team	**Equipe de desenvolvedores, designers, testadores, comprometidos a desenvolver o produto.**

Fonte: Disponível em "Modelo de Desenvolvimento ágil de Software Orientado à Qualidade" (GOMES,2011).

[8] *API* é a sigla em inglês para Interface de Programação de Aplicações

4. Pré-Planejamento

No momento posterior à definição da visão do projeto segundo as descrições do *Product Owner*, são decididas as ferramentas, tecnologias, padrões arquiteturais e conceitos aplicados ao desenvolvimento do projeto. Nesta fase também são definidas as funcionalidades gerais, descritas no *Product Backlog.*

4.1. Ferramentas e tecnologias utilizadas

Nesta seção são descritas as tecnologias, tecnologias e conceitos definidos para o desenvolvimento do *framework* CleanOnFire.

4.1.1. Android

O *Android* é o sistema operacional móvel desenvolvido e disponibilizado pela *Google*. É atualmente encontrado em mais de 80% dos smartphones ativos (GARTNER INC., 2017) que é também encontrado em uma variedade de dispositivos inteligentes, como *tablets*, caixas de transmissão de TV e outros aparelhos portáteis.

O *Android* tem sua base no sistema operacional de código aberto Linux, ao qual adiciona camadas de compatibilidade por meio de uma JVM[9] personalizada. Tomando vantagem do ambiente multiplataforma da plataforma de desenvolvimento *Java*, o sistema se tornou compatível com uma gama extensa de processadores e componentes, fazendo com que as fabricantes de dispositivos inteligentes o escolhessem para integrar muitos de seus produtos.

4.1.2. Java

O *Java* é uma plataforma de computação e linguagem de programação lançada pela Sun Microsystems em 1995 e desde 2009 tem sido mantida pela Oracle Corporation. Tendo seu maior diferencial na capacidade de operar com um modelo multiplataforma, fazendo com que um programa não tenha sua execução atrelada a uma arquitetura de processador, dependendo exclusivamente da plataforma *Java*.

O código escrito em *Java* é compilado para uma linguagem intermediária, chamada de *bytecode*, e, a partir do arquivo compilado, a aplicação é executada pela *Java Virtual Machine*, uma maquina virtual que é responsável por interpretar os comandos do *bytecode* e executa-los, como visto na figura 3.

[9] JVM (*Java Virtual Machine*) é a camada de execução de aplicações escritas na linguagem Java.

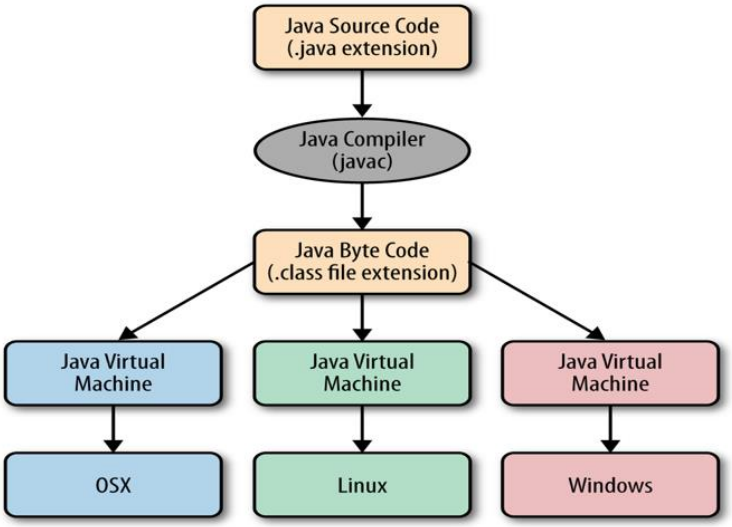


Figura 3: Esquema do processo de compilação e execução do *Java*

O *Java* foi escolhido para ser a plataforma base do *Android*, e por isso foi utilizada como linguagem de programação principal do software aqui apresentado.

4.1.3. Gradle

Gradle é uma ferramenta para gerenciamento de dependências, configuração de compilação, e automatização de processos relacionados ao *build*[10] de aplicações *Java*. Sendo a escolhida pela *Google* como ferramenta padrão para a configuração de compilação de aplicações *Android*.

Sendo altamente customizável, o *Gradle* permite o desenvolvimento de plug-ins instaláveis via código *Groovy* no script de compilação da aplicação. Através desse recurso é possível a geração de código fonte durante o processo de compilação, fator crucial para o desenvolvimento de algumas das funcionalidades desejadas para o *framework* CleanOnFire.

O *Gradle*, ao final de todos os processos de compilação comprime todos os arquivos executáveis e de recursos, como arquivos de mídia e de configuração, em um arquivo com a extensão ".apk", que é executável por qualquer dispositivo que tenha o *Android* como sistema operacional. O fluxo deste processo pode ser analisado na figura 4.

[10] *Build* é o processo no qual os arquivos código-fonte são compilados e em comprimidos com os arquivos de recursos em um arquivo executável.

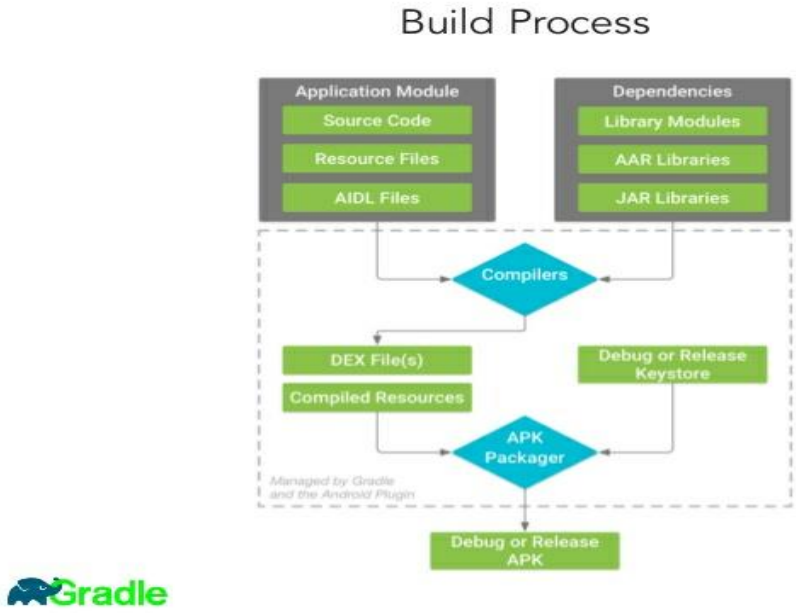


Figura 4: Processo de *build* de um aplicativo *Android* por meio do *Gradle Build Tool*

4.1.4. Android Studio

O *Android Studio* é o ambiente de desenvolvimento integrado oficial (IDE) para o desenvolvimento de aplicativos para Android, baseado no *IntelliJ IDEA*[11](GOOGLE INC, 2017a).

A ferramenta oferece um conjunto de utilidades desenvolvidas para o auxílio de tarefas cotidianas no desenvolvimento de aplicações, como um ambiente unificado para o desenvolvimento de aplicativos para qualquer tipo de dispositivo suportado pelo sistema operacional e ferramentas e estruturas de teste instrumentalizados e de unidade extensivas.

4.1.5. SQLite

O SQLite é o mecanismo de banco de dados SQL[12] incorporado ao sistema *Android*. Ao contrário da maioria dos bancos de dados SQL, o SQLite não possui um processo de servidor separado. O funcionamento dele se dá através da leitura e gravação direta em arquivos de disco comuns. O banco de dados completo com todas as tabelas, índices, gatilhos e visualizações está contido em um único arquivo de disco. Uma aplicação *Android* pode operar com quantas bases de dados SQLite lhe for conveniente, porém o acesso à base é restrito à aplicação que a criou.

A biblioteca do SQLite faz parte da *API* padrão do *Android*, o que torna sua utilização muito mais simples e integrada ao sistema. Outra característica que torna sua utilização mais simples é quantidade de material encontrada sobre ele, já que o código-fonte do projeto SQLite é de domínio público e, portanto, é gratuito para uso para qualquer finalidade, comercial ou privado.

4.1.6. Processamento de Anotações

[11] Ambiente integrado de desenvolvimento Java desenvolvido pela JetBrains Inc.

[12] SQL é a sigla para *Structured Query Language* que significa Linguagem de Consulta Estruturada, e define o padrão de interface dos bancos de dados relacionais.

O processamento de anotações é um conjunto de funcionalidades do compilador do *Java* para digitalização e processamento de anotações *Java*[13] em tempo de compilação. São múltiplas as suas possibilidades de uso, incluindo a geração de arquivos.

Um processador de anotações opera a partir do momento em que é registrado nas configurações de compilação de um projeto *Java*. O processamento de anotações foi inserido no JDK[14] a partir da versão 5 do *Java*, mas a *API* se tornou utilizável apenas na versão 6, lançada em dezembro de 2006 (DORFMANN, 2015).

A popularidade deste recurso tem crescido ao longo dos anos, a medida que vão sendo construídas ferramentas e utilitários para o desenvolvimento. Dois exemplos disso dentro da comunidade de desenvolvimento *Android* são o ButterKnife[15], biblioteca desenvolvida por Jake Wharton e o Dagger 2[16], mantido pela própria *Google*.

Um processador de anotação toma o código *Java*, compilado ou não, como entrada e, a partir das anotações, pode validar os arquivos processados assim como gerar novos arquivos como saída. No entanto não é possível manipular uma classe *Java* existente.

4.1.7. JavaPoet

JavaPoet[17] é uma biblioteca de código aberto desenvolvida pela Square Inc. que se propõe a facilitar a geração de código-fonte *Java*. Ela se baseia no padrão de projeto de criação *Builder* que como define Gamma (2000), tem a intensão de "separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações".

A sua utilização se mostra simples ao mesmo passo que a biblioteca mostra grande poder ao conter um inteligente sistema de inserção automática de *imports*[18] e a possibilidade de construção de qualquer tipo de estrutura suportada nativamente pelo *Java*.

A biblioteca é a base da geração de código-fonte de grande parte das bibliotecas que utilizam este recurso, por essa razão possui uma grande comunidade de apoiadores assim como uma extensa documentação. Por essa razão foi a ferramenta escolhida para incrementar o desenvolvimento deste projeto.

4.1.8. MVP

O padrão de arquitetura MVP é uma evolução do MVC[19]. Neste padrão, a exibição recebe os eventos de *UI*[20] e chama o *presenter* conforme necessário. O *presenter* também é responsável por atualizar a visualização com os novos dados gerados pelo modelo.

---

[13] Anotação Java é o mecanismo da linguagem para a inserção de meta-dados programaticamente acessíveis no código-fonte.

[14] JDK é a sigla em inglês para *Kit* de Desenvolvimento Java.

[15] Link direto para a página da biblioteca: http://jakewharton.github.io/butterknife/

[16] Link direto para a página da biblioteca: https://google.github.io/dagger/

[17] Link direto para a página da biblioteca: https://github.com/square/javapoet

[18] *Import* é o termo reservado dentro da linguagem Java para a importação de classes que não se encontram no mesmo arquivo.

[19] O MVC (Model-View-Controller) é um padrão arquitetural amplamente utilizado no mercado de desenvolvimento de *software* criado por Trygve Reenskaug em 1979.

[20] *UI* é a sigla em inglês para Interface de Usuário

A camada de modelo (*model*) pode ser pensado como a interface para os dados. Qualquer parte do programa que precisa de alguns dados para trabalhar deve passar pela interface ou funções definidas pelo desenvolvedor que está mantendo a parte do modelo. Tipicamente, o modelo abriga todas as rotinas de validação para os dados enviados pelo usuário final

A camada de visualização (*view*), como o nome indica, é a parte em que o usuário final interage. O desenvolvimento desta parte pode ser delegado a um designer especializado. Um programa pode ter qualquer número de visualizações.

A camada de apresentação (*presenter*) atua como intermediária para tornar possível a dissociação. Toda a lógica de negócios necessária para responder a um evento de usuário é escrita dentro desta camada. O *presenter* também é responsável por recuperar os dados solicitados do modelo e formata-lo para que a view possa exibi-lo sem lógica adicional.

O MVP, assim como seu predecessor MVC, é um dos padrões arquiteturais mais populares no mercado de software. O sistema projetado com padrão MVP também promove o teste de unidades, tornando o seu programa sólido. Seu fluxo de operação está representado na figura 5.

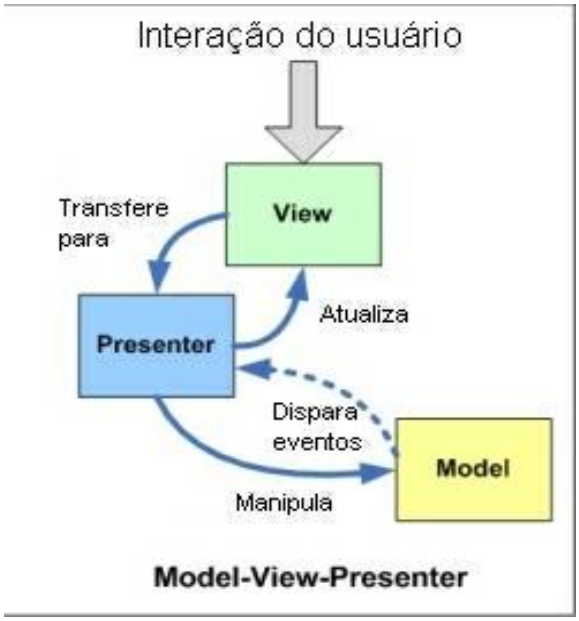


Figura 5. Fluxo de operação da arquitetura MVP

4.1.9. Astah Community

O Astah é uma ferramenta para a criação de diagramas de modelagem de software orientado a objetos nos padrões UML (*Unified Modeling Language*). O software foi desenvolvido no Japão e obteve o prêmio "Produto de Software do Ano 2006", pela Agência de Promoção de Informação Tecnológica no Japão. Em versões anteriores ele era conhecido como JUDE, mas ele rebatizado em 2009. Existem versões para as plataformas Windows, macOS e Linux.

4.2. Levantamento e Análise de Requisitos

Os requisitos funcionais e não funcionais foram levantados e analisados a partir das experiências pessoais e necessidades encontradas pelo autor deste artigo ao longo de dois anos de atuação no mercado de desenvolvimento para *Android*.

Foram enumeradas estruturas comuns à grande parte das aplicações desenvolvidas, que tendiam a conter código repetitivo. Desta forma abordou-se cada uma dessas estruturas, sugerindo estratégias para que fossem explorados potenciais ganhos de produtividade quanto a suas implementações.

A partir desta análise mais profunda foi gerado o documento de *Product Backlog*.

4.3. Product Backlog

O processo de definição de *APIs* e funcionalidades do *framework* se deu a partir da análise dos requisitos levantados. Essa definição é documentada no *Product Backlog*, com o conjunto de funcionalidades, cada uma respectivamente atrelada a um valor correspondente à estimativa de tempo total da implementação e outro correspondente à sua prioridade. Uma visão completa deste documento se encontra no anexo 4.

Pelo teor específico do domínio da aplicação para a plataforma *Android*, decidiu-se que termos técnicos como nomes de componentes e elementos da *API* padrão da plataforma seriam inseridos no documento para que fosse possível a contextualização dos seus elementos.

5. Desenvolvimento

Com base na divisão por módulos do *framework* o desenvolvimento do sistema foi dividido em três *sprints*. E após o término de cada fase, foi gerada uma versão utilizável do projeto.

As *sprint* de número 1 foi finalizada dentro do prazo, no entanto as de números 2 e 3 sofreram alterações em seu escopo e ultrapassaram o limite de suas respectivas estimativas de tempo. Esse fato resultou em uma reformulação total no tempo de entrega das últimas duas *sprints*.

Para que se mantivesse a integridade do artigo independentemente de futuras modificações quanto à implementação das ferramentas oferecidas pelo *framework*, exemplos com código-fonte foram omitidos das sessões que as descrevem. Sendo assim, maiores detalhes sobre a implementação de cada uma das funcionalidades podem ser encontrados na página oficial do projeto[21].

5.1. Primeira Srpint

Na primeira *sprint*, foram decididas as configurações iniciais do projeto, assim como os seus padrões de utilização, interfaces de desenvolvimento que seriam utilizadas pelos desenvolvedores.

Nesta fase também ocorreu o desenvolvimento do primeiro módulo do *framework*, referente à estruturação da arquitetura limpa em conjunto com o padrão MVP. Os elementos abordados nesta fase podem ser observados na tabela 2.

Tabela 2. *Sprint Backlog* da primeira *Sprint*

ID	Nome	Descrição	Estimativa	Prioridade
001	Criação de um *UseCase*	O desenvolvedor será capaz de criar uma classe que implementa um *UseCase*.	1	5

[21] Link direto: https://www.github.com/heitor-gia/CleanOnFire

002	Execução de um *UseCase* de forma assíncrona	O desenvolvedor será capaz de executar um *UseCase* de forma assíncrona a partir da classe *UseCaseExecution*.	1	5
003	Mapeamento de entidades na execução de um *UseCase*	O desenvolvedor será capaz de definir os tipos de entrada e saída de um *UseCase* a partir de mapeadores de entidade.	2	4
004	Definição de Thread postagem dos resultados	O desenvolvedor poderá definir em qual Thread o resultado do *UseCase* será postado	4	3
005	Mapeamento de entidades em Thread secundária	O mapeamento de entidades deverá ser feito fora da Thread de postagem de resultado	2	3
006	Criação de um *Presenter*	O desenvolvedor será capaz de criar um *Presenter* que executará os *UseCases* que lhe forem convenientes através da classe *UseCaseExecutor*	3	5
007	Vinculação de um *Presenter* a um contrato de visualização	O desenvolvedor será capaz de definir um contrato de visualização para o Presenter.	1	4
008	Parada de tarefas ao destruir o *Presenter*	Todos os *UseCases* em execução deverão ser parados e destruídos juntamente ao *Presenter* ao qual eles estão vinculados.	3	5

Ao finalizar a implementação de todas as funcionalidades referentes à primeira *sprint*, foi possível utilizar a estrutura de comunicação oferecida pelo *framework*. Esta estrutura consiste na execução de *UseCases*, que são as abstrações de tarefas únicas, com tipos de entrada e saída definidos, realizadas pelo sistema de forma assíncrona. Esta execução é efetuada a partir de uma entidade chamada *UseCaseExecutor*, que é responsável por controlar as execuções dos *UseCases* em *Threads*[22] de processamento concorrentes.

Execuções estas, que foram abstraídas pela entidade *UseCaseExecution*, na qual é possível mapear as entidades de entrada e saída do *UseCase*, assim como definir uma *Thread* na qual os resultados do processamento realizado serão postados.

Seguindo as diretrizes da arquitetura limpa, a execução de um UseCase era chamada de uma camada mais externa da aplicação, neste caso, o *presenter*. O presenter base do *framework* foi projetado para executar os *UseCases* e formatar os seus

[22] *Thread* é a entidade principal da API de concorrência do *Java*, é responsável por abstrair um núcleo de execução paralela.

resultados para então, enviá-los à camada de apresentação.

Ao final da realização dos testes, foi encontrada uma falha na estrutura que permitia vazamentos de memória. Este problema não permitia que os objetos não mais utilizados pelo sistema, que ainda se encontravam instanciados em memória, tivessem suas referências liberadas e fossem recolhidos pelo *garbage collector*[23]. No entanto, o problema foi solucionado antes do prazo de entrega da *sprint*, mantendo a regularidade do cronograma.

5.2. Segunda Srpint

Na *sprint* de número 2, o segundo módulo do framework foi desenvolvido. Este segundo módulo consistia na geração automática de entidades de *Adapter*[24] para visualizações de coleções de objetos, baseada em modelos de visualização. Este módulo foi batizado de *VisualizationModel API*. As funcionalidades implementadas nesta fase do projeto podem ser observadas na tabela 3.

Tabela 3. *Sprint Backlog* da segunda *sprint*.

ID	Nome	Descrição	Estimativa	Prioridade
009	Geração automática de classes *Adapter*	O desenvolvedor poderá utilizar classes de Adapters geradas em tempo de compilação conforme uma classe modelo anotada com a anotação @VisualizationModel.	15	5
010	Suporte a *RecyclerView* e *ListView* nos *Adapters* gerados	O desenvolvedor poderá definir se o modelo de visualização gerará um adapter para *ListView* ou *RecyclerView*.	7	4
011	Customização do procedimento de vinculação do atributo do modelo à visualização	O desenvolvedor poderá definir uma classe que implementa *ViewBinder* para customizar o processo de vinculação daquela propriedade à sua *View* atrelada.	15	4
012	Geração de métodos de para definição de gatilhos a partir de eventos de clique e clique longo	O desenvolvedor poderá definir se uma determinada visualização terá eventos de clique e clique longo, para que os métodos de definição sejam criados dentro do adapter.	10	3
013	Geração automática de classes *ViewHolder*	As classes Adapters deverão implementar o padrão *ViewHolder* e essas implementações devem ser geradas automaticamente.	10	4
014	Geração automática de classes	As classes responsáveis pela vinculação de um item ao *ViewHolder* deverão ser geradas	7	3

[23] *Garbage collector* é o mecanismo da API *Java* para limpeza de memória em tempo de execução.

[24] *Adapters* na *API* do *Android*, são entidades baseadas no padrão de projeto *Adapter*, responsáveis por converte objetos em implementações de visualizações.

	ViewHolderBinder	automaticamente.		

Este módulo opera baseado na geração de código-fonte. Neste caso, o código gerado é referente às entidades de *Adapter* da *API* padrão do *Android.* A geração do código é baseada em um *POJO*[25] anotado com uma anotação *Java* customizada chamada *VisualizationModel*, à qual são informados o recurso de *layout* e o tipo de *adapter* a ser gerado.

O parâmetro referente ao tipo de *adapter* informa ao processador de anotações se o *adapter* deve ser construído para uma *ListView*[26] ou para uma *RecyclerView*[27]. Já o parâmetro de recurso de *layout* informa o *layout* a ser renderizado para a visualização do modelo anotado.

O processo de vincular um atributo do modelo à visualização é feito por uma implementação da interface *ViewBinder*, que pode ser criada pelo desenvolvedor ou importada da *API* do *framework*. Dessa forma, o processo se torna extremamente transparente e versátil, dando ao desenvolvedor total controle sobre a forma como cada elemento vai ser vinculado à sua visualização.

Cada atributo vinculado à visualização deve ser anotado com a anotação customizada *Bind*, à qual são informados o recurso de visualização, a classe do componente visual vinculado, a classe que implementa *ViewBinder* e, opcionalmente, se este componente responderá aos eventos de clique e/ou clique longo.

Os *adapters* gerados implementam o padrão *ViewHolder*, que segundo a documentação do *Android* (GOOGLE INC., 2017b), armazena cada uma das visualizações dos componentes dentro do campo *tag* do Layout, para que se possa acessá-las imediatamente sem a necessidade de procurá-las repetidamente. Estes elementos também são gerados automaticamente. Esta prática faz com que o consumo de memória e processamento seja reduzido ao rolar uma lista.

É importante ressaltar que apesar de ser versátil quanto à renderização dos componentes, a *API* tem a limitação de não suportar listas com visualizações heterogêneas, isto é, não é possível definir mais de um tipo de visualização para uma mesma lista.

A entrega dessa *sprint* ultrapassou seu prazo limite em razão de dificuldades encontradas na implementação do processador de anotações, acarretando em um considerável atraso

5.3. Terceira Srpint

A terceira *sprint* foi a mais longa dentre todas, contemplando o desenvolvimento do módulo com a implementação mais complexa do *framework.* Nesta fase foi desenvolvido o módulo de abstração da base de dados, batizado de CleanOnFireDB. As funcionalidades implementadas durante a *sprint* podem ser observadas na tabela 4.

Tabela 4. Sprint Backlog da terceira sprint.

[25] *POJO* é a sigla para *Plain Old Java Object*, e se refere objetos já que possuem apenas atributos, e métodos de acesso e de atribuição a estes atributos.

[26] *ListView* é o componente de visualização de listas padrão do *Android*.

[27] *RecyclerView* é o novo componente de visualização de coleções do *Android*, desenvolvido para uma maior performance de renderização.

ID	Nome	Descrição	Estimativa	Prioridade
015	Geração automática de classes *DAO*	O desenvolvedor poderá utilizar classes de *DAO* geradas automaticamente a partir de classes anotadas com @Table.	25	5
016	Disponibilização de operações de inserção, deleção, edição e consulta nas classes *DAO*	Deverão ser disponibilizadas APIs de inserção, deleção e edição de dados em várias naturezas (listas, arrays ou itens únicos) assim como APIs de consulta ao banco de dados.	20	4
017	Geração do *script* *SQL* para a criação dos esquemas de banco de dados	O *script* de *DDL* (*Data Definition Language*) *SQL* deve ser gerado automaticamente de acordo com as definições das classes anotadas com @Table.	10	4
018	Geração dos métodos de consulta de chave estrangeira.	As classes *DAO* deverão disponibilizar métodos para a consulta de entidades relacionadas por meio de chaves estrangeiras.	5	2
019	Geração das classes de identificação de cada tabela	As classes DAO deverão ser atreladas a uma classe que implementa *Identification* e que abstrai o conjunto de chaves primárias da tabela atrelada ao *DAO.*	20	5
020	Disponibilização da *API* de consultas ao banco de dados independente de entidade	Deverá ser disponibilizada uma *API* de consulta ao banco de dados independente de entidade para consultas mais complexas ou que mesclam dois ou mais tipos de entidades.	10	2
021	Disponibilização de uma *API* de migrações.	O desenvolvedor poderá utilizar uma *API* de migrações para versionamento do esquema de banco de dados.	15	3

O desenvolvimento deste módulo também se baseou nas técnicas de geração de código-fonte e processamento de anotações para construir uma camada de abstração do banco de dados *SQLite*. Essa camada de abstração é construída conforme o padrão de projeto *DAO (*Objeto de Acesso a Dados, do inglês *Data Access Object)*, que se propõe a agrupar todas as operações de bancos de dados atreladas à uma entidade em um único objeto.

As definições de nome e versão da base de dados são definidas através da anotação *Database*, com a qual se anota uma interface que receberá também as implementações de migração entre versões.

O acesso aos *DAOs* gerados se dá através de uma classe com o padrão *singleton*[28], também gerada em tempo de compilação, chamada de CleanOnFireDB. Nesta classe também se encontram a definição de esquema do banco de dados e os dados definidos na interface anotada com *Database*.

A definição de tabelas, na qual o processador de anotações se baseia para gerar as classes *DAO*, ocorre através da anotação *Table*, na qual se pode, opcionalmente, informar o nome da tabela a ser criada. Por padrão o nome da tabela será o mesmo da classe anotada.

Todos os atributos de uma classe anotada com *Table* são considerados como definições de colunas, a menos que se declare que o elemento deva ser ignorado por meio da anotação *IgnoreField*. Configurações adicionais como de chave primária, simples e composta, chave estrangeira, nome, tamanho máximo entre outros são facilmente declarados por meio das anotações disponíveis na *API*.

A partir destas definições de tabela é possível gerar o objeto *DAO* específico de cada tabela. Para a facilitação deste processo, foi utilizado o recurso de tipagem genérica do *Java*, que segundo a documentação oficial (ORACLE CORPORATION, 2017b), permitem que tipos (classes e interfaces) sejam usados como parâmetros ao definir classes, interfaces e métodos. Sendo assim foi construída uma classe base com tipagem genérica que implementava métodos de inserção, deleção, edição e consulta baseados no tipo definido, que neste caso representa o tipo da entidade.

Porém é necessário salientar que o módulo se diferencia de um *ORM*[29], pois não mapeia, de fato, relações do esquema do banco de dados baseando-se na relação de composição entre objetos, mas sim define colunas que se relacionarão com colunas de outras tabelas por meio de índices.

Uma das principais limitações da utilização da ferramenta é a falta de suporte a múltiplas bases de dados. No entanto, este recurso deve ser adicionado em versões futuras do *framework*.

Em relação à entrega, a *sprint* teve sua finalização atrasada em razão de um equívoco inicial sobre o escopo do módulo, que compreendia mapear relações do esquema de dados por meio da relação de composição entre as classes definidas como entidades. Isso fez com que houvesse uma reformulação de seu funcionamento, acarretando em atrasos severos.

6. Pós Planejamento

Na fase final do projeto, foram realizados testes através de uma aplicação de exemplo que utiliza todos os recursos disponíveis no *framework*. Com o sucesso nos testes realizados pelo *team* do projeto, o processo de publicação da ferramenta foi iniciado.

O resultado final do projeto está publicado na plataforma do *GitHub*[30], que hospeda repositórios de código aberto com base no sistema de versionamento *Git*[31].

---

[28] *Singleton* é um padrão de projeto no qual o número de instâncias de uma classe é limitado a 1.

[29] *ORM* é a sigla em inglês para Mapeador Objeto-Relacional, que define uma modalidade de sistemas que se propõe a mapear os relacionamentos entre tabelas do banco de dados baseando-se na relação de composição das classes definidas.

[30] Link direto para página do framework: https://github.com/heitor-gia/CleanOnFire

[31] *Git* é um Sistema código aberto de versionamento de arquivos idealizado por Linus Torvalds, o criador do Linux.

Dessa forma a comunidade poderá contribuir ativamente para o crescimento da ferramenta, assim como a correção de eventuais falhas no código-fonte.

7. Considerações Finais

O objetivo desse projeto era integrar ferramentas facilitadoras para o desenvolvimento de aplicações *Android*, tornando a prototipação das aplicações mais ágil, sem que houvesse perda da qualidade do código. A plataforma pode uma considerável quantidade de *boilerplates*[32], por isso o desenvolvimento de boas bibliotecas de código aberto geralmente tem boa receptividade na comunidade.

Alguns dos maiores desafios encontrados foram os referentes a utilização das tecnologias escolhidas. Por se tratar de um recurso relativamente pouco popular, os materiais de estudo sobre processamento de anotações em tempo de compilação são escassos, assim como a documentação a seu respeito. Outras dificuldades foram encontradas também na implementação da *API* de programação concorrente oferecida pelo *Java*, que possui algumas peculiaridades que precisaram ser analisadas com mais cuidado. Outra causa de dificuldades dentro do projeto foi a falta de tempo disponível para a sua realização, que acarretou em alguns atrasos nas entregas das *spints.*

Entretanto, as dificuldades encontradas serviram para adquirir conhecimentos e habilidades que certamente podem agregar valor ao perfil profissional. Pois muitos dos conceitos abordados neste projeto têm pouca divulgação, porém grande potencial.

Sendo assim, ao final deste projeto pode-se afirmar que ele teve seus objetivos atingidos dentro das limitações de escopo e tempo disponível. A manutenção e evolução da ferramenta será administrada pelo autor do trabalho em conjunto à comunidade de desenvolvedores *Android*. Espera-se que estas ferramentas auxiliem a todos que se propuserem a utiliza-las.

Referências

BASRI, Shuib; O'CONNOR, Rory V.. Understanding the Perception of Very Small Software Companies Towards the Adoption of Process Standards. **Systems, Softwares And Services Process Improvements**, v. 99, p.153-164, jun. 2010.

COCKBURN, Alistair. **Hexagonal architecture**. 2005. Disponível em: <http://alistair.cockburn.us/Hexagonal+architecture>. Acesso em: 03 out. 2017.

DORFMANN, Hannes. **Annotation Processing 101**. 2015. Disponível em: <http://hannesdorfmann.com/annotation-processing/annotationprocessing101>. Acesso em: 14 set. 2017.

EMMATTY, John. **Differences between MVC and MVP for Beginners**. 2011. Disponível em: <https://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners>. Acesso em: 22 out. 2017.

[32] *Boilerplate* é o termo utilizado para estruturas dentro do código-fonte que se tornam repetitivas ao longo do desenvolvimento de um sistema.

GAMMA, Erich et al**. Padrões de Projeto**: Soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. 364 p.

GARTNER INC. (Org.). **Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2017**. 2017. Disponível em: <https://www.gartner.com/newsroom/id/3725117>. Acesso em: 18 set. 2017.

GUERRA, Eduaro. **Design Patterns com Java**: Projeto orientado a objetos guiado por padrões. São Paulo: Casa do Código, 2014. 274 p.

GOMES, Marcos Vinicius Rocha. **MODELO DE DESENVOLVIMENTO ÁGIL DE SOFTWARE ORIENTADO À QUALIDADE**. 2011. 18 f. Tese (Doutorado) - Curso de Graduação em Gestão e Governança de Ti, Universidade Luterana do Brasil (ULBRA), Canoas, 2011

GOOGLE INC. (Org.). **Android Studio**: O IDE oficial do Android. 2017. Disponível em: <https://developer.android.com/studio/index.html?hl=pt-br>. Acesso em: 19 set. 2017.

GOOGLE INC. (Org.). **Making ListView Scrolling Smooth**. Disponível em: <https://developer.android.com/training/improving-layouts/smooth-scrolling.html>. Acesso em: 29 out. 2017.

JENKOV, Jakob. **The DAO Design Pattern.** Disponível em: <http://tutorials.jenkov.com/java-persistence/dao-design-pattern.html>. Acesso em: 28 out. 2017.

MACHADO, Henrique. **Os 4 pilares da Programação Orientada a Objetos.** Disponível em: <https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>. Acesso em: 19 set. 2017.

MANCUSO, Sandro. **Bad Code:** The Invisible Threat. 2010. Disponível em: <http://craftedsw.blogspot.com.br/2010/09/bad-code-invisible-threat.html>. Acesso em: 2 nov. 2017.

MARTIN, Robert Cecil. **Código Limpo**: Habilidades Práticas do Agile Software. 3. ed. Rio de Janeiro: Alta Books, 2008. 440 p.

MARTIN, Robert Cecil. **The Clean Architecture.** 2012. Disponível em: <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 14 set. 2017.

ORACLE CORPORATION (Org.). **Best Practices:** Changing Compiled Java Classes with Byte Code Weaving. Disponível em: <https://docs.oracle.com/cd/E17904_01/doc.1111/e16596/bestpractice.htm#CHDIJHAA>. Acesso em: 14 set. 2017.

ORACLE CORPORATION. **Why Use Generics?** Disponível em: <https://docs.oracle.com/javase/tutorial/java/generics/why.html>. Acesso em: 15 nov. 2017.

SABBAGH, Rafael. **Scrum**: Gestão Ágil para Projetos de Sucesso. São Paulo: Casa do Código, 2013.

SILVEIRA, Paulo et al. **Introdução à Arquitetura de Design de Software**. Rio de Janeiro: Elsevier, 2011.

SQLITE ORG. (Org.). **About SQLite.** 2017. Disponível em: <https://sqlite.org/about.html>. Acesso em: 14 set. 2017.

QUICOLI, Paulo. **O Padrão MVP (Model-View-Presenter) Facebook Twitter.** Disponível em: <https://www.devmedia.com.br/o-padrao-mvp-model-view-presenter/3043>. Acesso em: 12 out. 2017.

Anexos

Anexo 1: Comparação com bibliotecas para estruturação de arquitetura

Funcionalidade	**CleanOnFire**	**EasyMVP**	**Mosby**	**ThirtyInch**
Implementação padrão do modelo MVP	X	X	X	X
Implementação de *presenters* genéricos	X	X	X	X
Integração *built-in* com a biblioteca RxJava		X	X	X
Estruturação com padrão de *UseCases*	X	X		
Execução de *UseCases* com padrão *pipeline*	X			
API de mapeamento de entidades	X	X		
Suporte a extensões de terceiros			X	X

Anexo 2: Comparação com bibliotecas para abstração de banco de dados

Funcionalidade	**CleanOnFireDB**	**Room**
Implementação no padrão DAO	X	X
Suporte a migrações	X	X
Suporte a *API* de *LiveData*		X
Suporta a múltiplos bancos de dados		X
Geração automática de funções de consultas	X	
Geração automática de funções de inserção, deleção e edição	X	
Suporte a consultas de modelos não anotados	X	

Anexo 3: Comparação com bibliotecas para geração de Adapters de

Funcionalidade	**VisualizationModel *API* - CleanOnFire**	**AutoAdapter**
Gera as classes de Adapters	X	X
Suporte a RecyclerView	X	X
Suporte a ListView e Spinner	X	
Geração automática de funções de clique e clique longo	X	
Personalização de vinculação a atributos da View	X	
Implementação de vinculação transparente ao desenvolvedor	X	

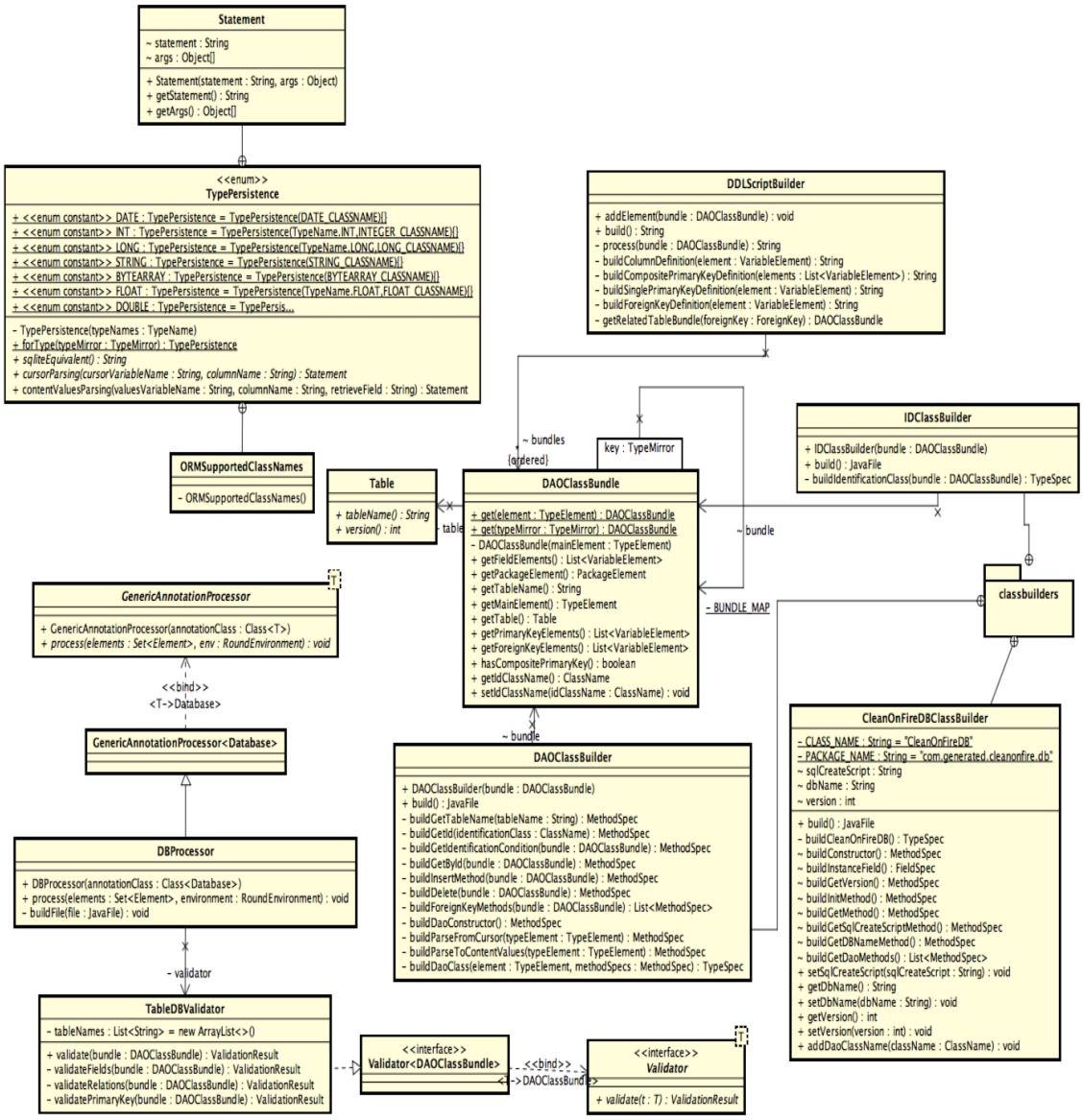
Anexo 4: Tabela de Product Backlog

ID	Nome	Descrição	Estimativa	Prioridade
001	Criação de um *UseCase*	O desenvolvedor será capaz de criar uma classe que implementa um *UseCase*.	1	5
002	Execução de um *UseCase* de forma assíncrona	O desenvolvedor será capaz de executar um *UseCase* de forma assíncrona a partir da classe *UseCaseExecution*.	1	5
003	Mapeamento de entidades na execução de um *UseCase*	O desenvolvedor será capaz de definir os tipos de entrada e saída de um *UseCase* a partir de mapeadores de entidade.	2	4
004	Definição de Thread postagem dos resultados	O desenvolvedor poderá definir em qual Thread o resultado do *UseCase* será postado	4	3
005	Mapeamento de entidades em Thread secundária	O mapeamento de entidades deverá ser feito fora da Thread de postagem de resultado	2	3
006	Criação de um Presenter	O desenvolvedor será capaz de criar um *Presenter* que executará os *UseCases* que lhe forem convenientes através da classe *UseCaseExecutor*	3	5
007	Vinculação de um *Presenter* a um contrato de visualização	O desenvolvedor será capaz de definir um contrato de visualização para o Presenter.	1	4
008	Parada de tarefas ao destruir o Presenter	Todos os *UseCases* em execução deverão ser parados e destruídos juntamente ao *Presenter* ao qual eles estão vinculados.	3	5

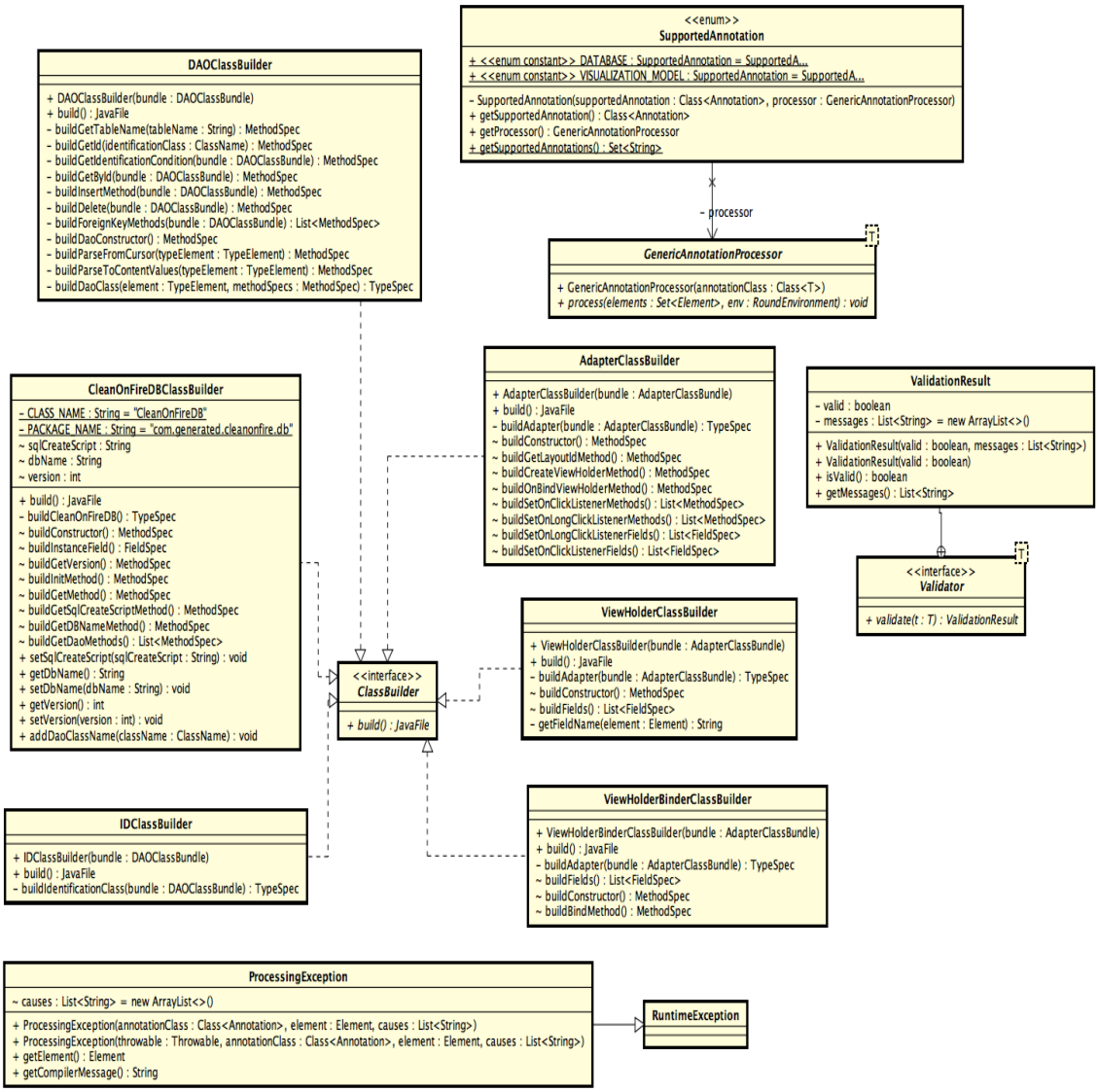
009	Geração automática de classes *Adapter*	O desenvolvedor poderá utilizar classes de *Adapters* geradas em tempo de compilação conforme uma classe modelo anotada com a anotação @VisualizationModel.	15	5
010	Geração de *Adapter* com suporte a *RecyclerView* e *ListView*	O desenvolvedor poderá definir se o modelo de visualização gerará um *adapter* para *ListView* ou *RecyclerView.*	7	4
011	Customização do procedimento de vinculação do atributo do modelo à visualização	O desenvolvedor poderá definir uma classe que implementa *ViewBinder* para customizar o processo de vinculação daquela propriedade à sua *View* atrelada.	15	4
012	Geração de métodos de para definição de gatilhos a partir de eventos de clique e clique longo	O desenvolvedor poderá definir se uma determinada visualização terá eventos de clique e clique longo, para que os métodos de definição sejam criados dentro do *adapter.*	10	3
013	Geração automática de classes *ViewHolder*	As classes Adapters deverão implementar o padrão ViewHolder e essas implementações devem ser geradas automaticamente	10	4
014	Geração automática de classes *ViewHolderBinder*	As classes responsáveis pela vinculação de um item ao ViewHolder deverão ser geradas automaticamente.	7	3
015	Geração automática de classes *DAO*	O desenvolvedor poderá utilizar classes de *DAO* geradas automaticamente a partir de classes anotadas com @Table.	25	5
016	Disponibilização de	Deverão ser	20	4

	operações de inserção, deleção, edição e consulta nas classes *DAO*	disponibilizadas APIs de inserção, deleção e edição de dados em várias naturezas (listas, *arrays* ou itens únicos) assim como *APIs* de consulta ao banco de dados.		
017	Geração do *script SQL* para a criação dos esquemas de banco de dados	O *script* de *DDL* (*Data Definition Language*) *SQL* deve ser gerado automaticamente de acordo com as definições das classes anotadas com @Table.	10	4
018	Geração dos métodos de consulta de chave estrangeira.	As classes *DAO* deverão disponibilizar métodos para a consulta de entidades relacionadas por meio de chaves estrangeiras.	5	2
019	Geração das classes de identificação de cada tabela	As classes DAO deverão ser atreladas a uma classe que implementa *Identification* e que abstrai o conjunto de chaves primárias da tabela atrelada ao *DAO.*	20	5
020	Disponibilização da *API* de consultas ao banco de dados independente de entidade	Deverá ser disponibilizada uma *API* de consulta ao banco de dados independente de entidade para consultas mais complexas ou que mesclam dois ou mais tipos de entidades,	10	1
021	Disponibilização de uma *API* de migrações.	O desenvolvedor poderá utilizar uma *API* de migrações para versionamento do esquema de banco de dados.	15	3

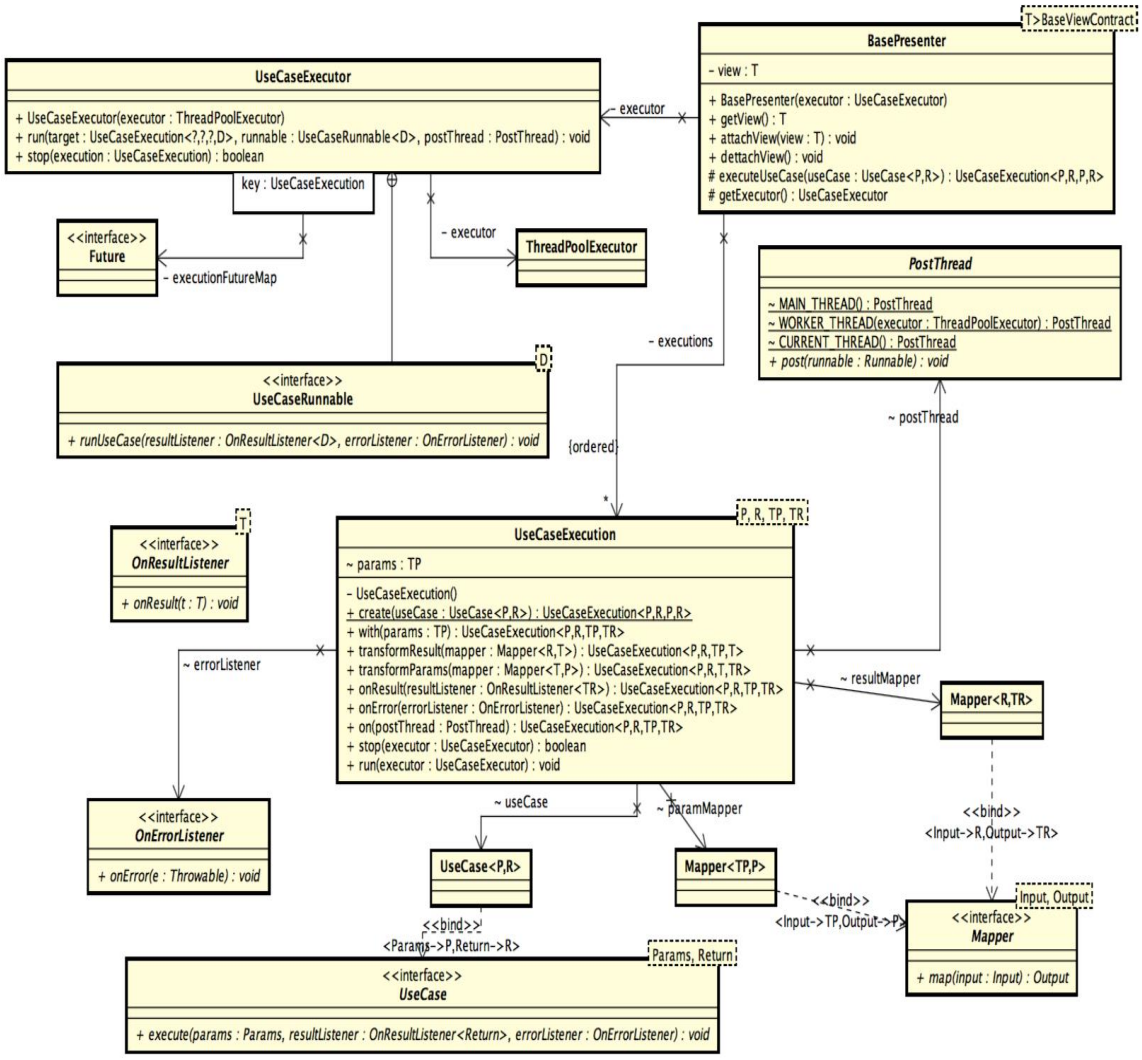
Anexo 5: Diagrama de Classe da API de geração de estruturas DAO



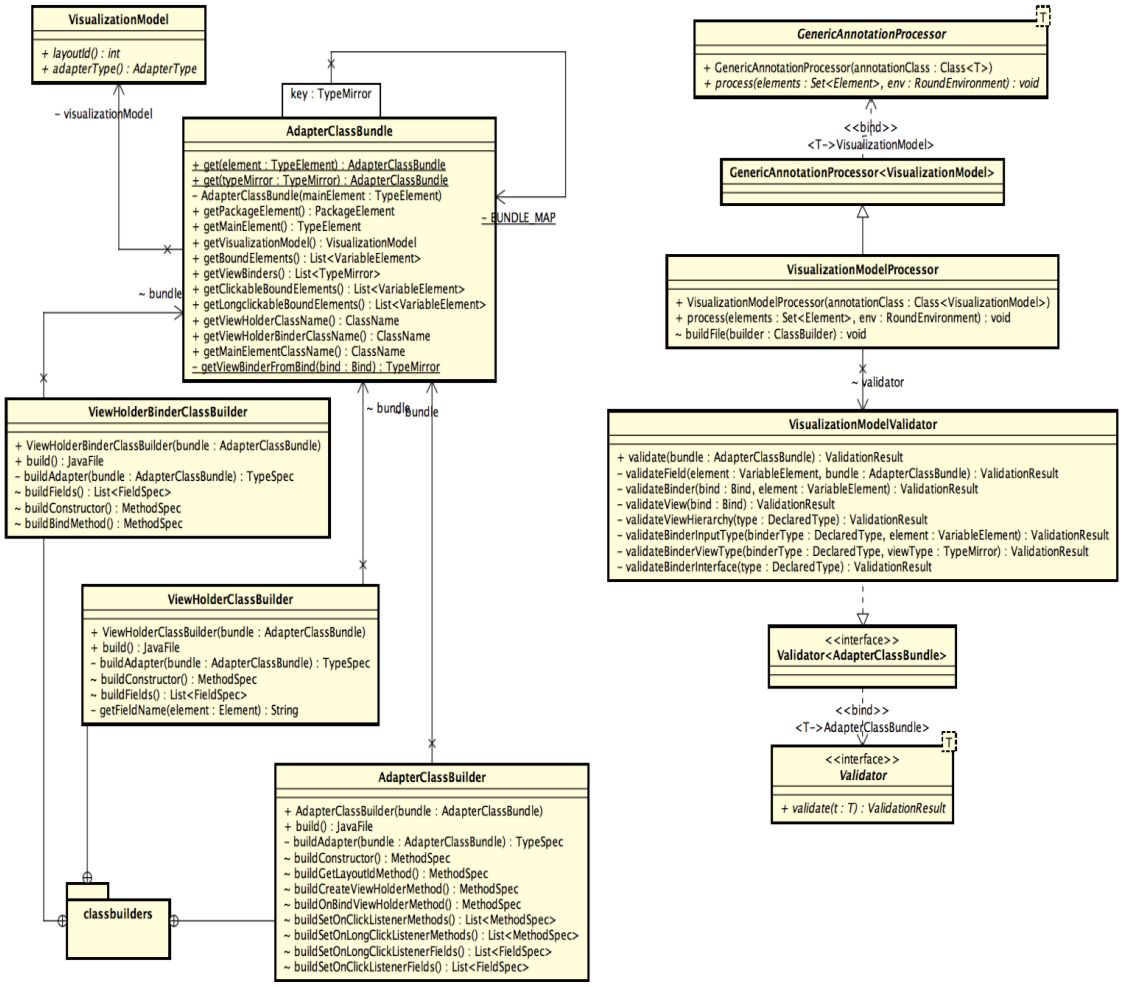
Anexo 6: Diagrama de Classe da API de geração de código-fonte.



Anexo 7: Diagrama de Classe da API de execução assíncrona de *UseCases*.



Anexo 8: Diagrama de Classe da API de geração de estruturas Adapter



Anexo 9 – *Printscreen* da página do projeto.