LUCAS HENRIQUE BENTO TOLEDO

Orientador: Rodrigo Geraldo Ribeiro

DESENVOLVIMENTO DE APLICAÇÕES ANDROID ADAPTÁVEIS UTILIZANDO A LINGUAGEM KOTLIN

Ouro Preto

Dezembro de 2019

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

DESENVOLVIMENTO DE APLICAÇÕES ANDROID ADAPTÁVEIS UTILIZANDO A LINGUAGEM KOTLIN

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

LUCAS HENRIQUE BENTO TOLEDO

Ouro Preto
Dezembro de 2019

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Lucas Henrique Bento Toledo

Desenvolvimento de aplicações Android adaptáveis utilizando a linguagem Kotlin

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 23 de Abril de 2021.

Membros da banca

Rodrigo Geraldo Ribeiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Reinaldo Silva Fortes (Examinador) - Mestre - Universidade Federal de Ouro Preto
Saul Emanuel Delabrida Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto

Rodrigo Geraldo Ribeiro, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 23/04/2021.

Documento assinado eletronicamente por **Reinaldo Silva Fortes**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 29/04/2021, às 11:35, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 29/04/2021, às 13:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0160359** e o código CRC **A8337C0E**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Resumo

Sistemas interativos vêm dominando o mercado de softwares nos últimos anos o que causou a necessidade de se padronizar o desenvolvimento desses sistemas. Com isso, surgiram padrões de projeto, padrões de arquitetura de software, e alguns frameworks de modo a automatizar certos a construção de software. Formulários estão presentes na maioria destes sistemas, e sua criação é mecânica, o que pode consumir um tempo precioso na construção de um projeto. Neste sentido, o presente trabalho apresenta um sistema para automatizar a criação de formulários a partir da descrição de um esquema relacional do banco de dados manipulado. A solução proposta foi implementada utilizando a linguagem Kotlin, amplamente utilizada para o desenvolvimento de aplicações para a plataforma Android. Experimentos conduzidos com a ferramenta produziram formulários automaticamente a partir da descrição de tabelas do banco de dados manipulado e das respectivas regras de validação para campos destas tabelas.

Abstract

Interactive systems have dominated the software market in the recent years, and as a result, there was a need to standardize its development. Design and architectural patterns, and frameworks have been created to automate software design and implementation. Forms are an essential component of many software systems and its construction is a trivial task that surely could be automated. In this work, we describe a solution to automatically create a form from a relational schema. The proposed solution was developed using Kotlin, a new programming language which was created to ease the task of developing Android applications. Experiments with the developed tool showed that it can be used to automatically generate forms from table descriptions and its field validation rules.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

A demanda de sistemas de informação para dispositivos móveis vem crescendo ao longo dos últimos anos (Moskala e Wojda, 2017). Testemunho deste fato, é a popularização de aplicativos de entregas como o *IFood*, transporte, por exemplo o *Uber*, e redes sociais como *Facebook*, *Twitter*, entre outras. Visando suprir o desenvolvimento de software com qualidade, diversas técnicas de engenharia de software têm sido aplicadas com sucesso no contexto de aplicações móveis. Dentre essas técnicas, destaca-se o uso de padrões de projeto (Gamma et al., 1995) e arquiteturais (Buschmann, 1996) além do uso de arcabouços (frameworks) para garantir o reuso no desenvolvimento de sistemas móveis.

Atualmente aplicações para dispositivos móveis são desenvolvidas usando diferentes tecnologias que dependem do sistema operacional (Android ou IOS). Alguns arcabouços suportam o desenvolvimento para ambas as plataformas, e dois dos mais famosos são o Flutter e o React Native. Para desenvolver uma aplicação para Android, pode-se usar qualquer sistema operacional, porém as aplicações desenvolvidas só funcionam em aparelhos com o sistema Android. Neste trabalho de monografia concentramos nossos esforços em aplicações Android, por ser de código aberto e ser o sistema operacional mobile mais utilizado no mundo.

Grande parte das aplicações atuais, demandam alguma forma de interação com um sistema gerenciador de banco de dados (SGDB). Estes softwares devem prover suporte às principais operações suportadas por um SGDB: *Create* (incluir), *Read* (consultar), *Update* (atualizar) e *Delete* (remover). Usualmente essas operações são referidas pela abreviação CRUD que indica cada uma das funcionalidades de um SGDB.

Com o uso de padrões de projeto, grande parte do desenvolvimento de tais sistemas é uma tarefa essencialmente mecânica, dependente apenas da estrutura do banco de dados manipulado. Nesse sentido, justifica-se o uso de ferramentas para a geração de código para manipulação de informações armazenadas em um SGDB ou mesmo a elaboração de aplicações capazes de adaptar-se automaticamente à estrutura de um banco de dados. Um dos objetivos deste trabalho é o desenvolvimento de aplicações mobile capazes de adaptar-se a diferentes esquemas relacionais sem a intervenção de um programador ou necessidade de recompilar o

seu código.

Aplicações *Android* são desenvolvidas utilizando diferentes linguagens, a saber: XML para estruturar o layout do aplicativo e Java ou Kotlin para o desenvolvimento de recursos dinâmicos executados pelo aplicativo. Dessa forma, a tarefa de construir um novo elemento de interface consiste de reestruturar um arquivo XML e das regras de negócio para validação da informação manipulada.

Nesse contexto, o presente trabalho apresenta uma biblioteca, desenvolvida usando a linguagem Kotlin, para automatizar a criação de formulários a partir de um esquema relacional. O principal objetivo desta biblioteca é permitir uma rápida prototipação de aplicações mobile, o que é essencial em etapas iniciais de um projeto de desenvolvimento. Esperamos com isso, reduzir o tempo de desenvolvimento de aplicações Android para automatizar a criação de formulários simples para manipulação de bancos de dados.

1.1 Justificativa

Sistemas de informação são um dos pilares do sucesso de grandes empresas como Amazon, Submarino e E-bay Laudon e Laudon (2001). Tais empresas dependem crucialmente do funcionamento correto de software para seu sucesso. Porém, o desenvolvimento ágil e correto de software é uma tarefa desafiadora.

Nesse sentido, ferramentas que automatizam total ou parcialmente tarefas de desenvolvimento são fundamentais. Dessa forma, o presente trabalho justifica-se por propor uma biblioteca para minimizar os efeitos e custos de tarefas de desenvolvimento e manutenção de softwares para plataforma Android, bem como reduzir o tempo de construção de aplicações com interface de usuário.

1.2 Objetivos

O objetivo deste trabalho é o desenvolvimento de uma biblioteca Android para automatizar a criação de formulários a partir de um esquema relacional fornecido por um desenvolvedor. Especificamente:

- Projetar e implementar uma biblioteca, usando a linguagem Kotlin, para automatizar a criação de formulários a partir de um esquema relacional.
- Utilizar a biblioteca desenvolvida para a prototipação de uma aplicação para a plataforma Android.

1.3 Organização do Trabalho

O restante deste trabalho é organizado da seguinte maneira: no capítulo 2 discutimos sobre o padrão arquitetural MVC, usado no desenvolvimento deste trabalho e sobre a plataforma Android (seções 2.1 e 2.2, respectivamente). Uma breve introdução à linguagem Kotlin e como essa pode ser utilizada para desenvolver aplicações Android é apresentada na seção 2.2.1. Trabalhos relacionados são discutidos na seção 2.4. Detalhes de implementação da biblioteca são discutidos no capítulo 3 em conjunto com um estudo de caso em que a biblioteca foi utilizada para a criação de aplicações. Finalmente, o capítulo 4 apresenta as considerações finais.

Todo o código desenvolvido como parte desse trabalho monográfico encontra-se disponível no seguinte endereço: `https://github.com/lucashtb/monografia1`

Capítulo 2

Fundamentação Teórica

Neste capítulo apresentaremos os conceitos necessários para o entendimento do trabalho desenvolvido. Primeiramente, na Seção 2.1, descrevemos o padrão arquitetural MVC que é amplamente utilizado em aplicações Android. Em seguida, a Seção 2.2 apresenta as principais características do sistema Android, bem como seu ambiente de desenvolvimento, o Android Studio. Em seguida, apresentamos uma breve introdução à linguagem de programação Kotlin na seção 2.2.1. Finalmente, a Seção 2.4 apresenta os trabalhos relacionados.

2.1 Padrão de projeto MVC

Todo artefato de software utiliza alguma forma de interação e esta é feita usando uma interface com o usuário. Logo, percebe-se que o problema de integrar a lógica de interação com o domínio da aplicação é um problema recorrente na engenharia de software. Padrões de projeto (Buschmann, 1996; Gamma et al., 1995) provêm esquemas de soluções para problemas recorrentes no desenvolvimento de software. Em especial, destaca-se o padrão MVC (**M**odel-**V**iew-**C**ontroller) que descreve uma forma sistemática de integrar uma interface com o usuário ao domínio da aplicação desenvolvida. O MVC foi concebido por Reenskaug durante uma visita ao laboratório Xerox-PARC na década de 70 e logo se tornou popular na comunidade de Small-Talk. Na década seguinte já era amplamente utilizado nas mais diferentes aplicações (Burbeck, 1987). A Figura 2.1 apresenta o relacionamento entre os diferentes componentes da arquitetura MVC.

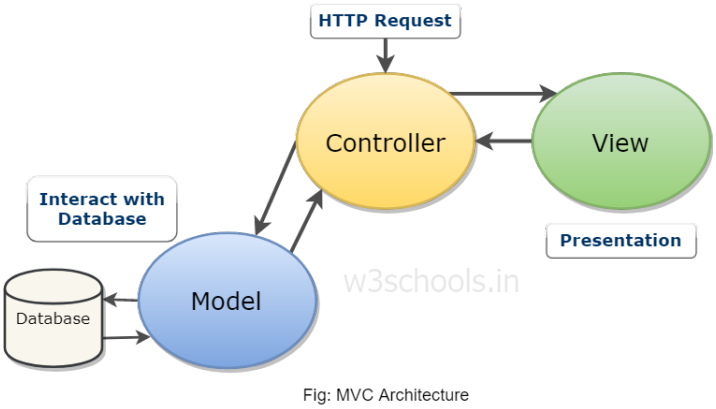

Figura 2.1: Arquitetura MVC - Fonte: https://www.w3schools.in/mvc-architecture

Na figura 2.1, o modelo é o componente responsável por definir, validar e armazenar os dados relevantes para a aplicação. O código presente na camada modelo é responsável por executar comandos originados na interface (visão), que são a ele repassados pelo controlador.

Por sua vez, a visão é responsável pela apresentação da informação armazenada pela aplicação. Visões são criadas por dados obtidos a partir do modelo e que podem ser exibidos utilizando diagramas, tabelas ou componentes de interface gráfica como caixas de texto e botões.

Finalmente, o controlador é a camada da aplicação responsável por manipular a interação com o usuário. Controladores interpretam eventos gerados a partir de dispositivos de entrada (mouse, teclados, etc) e solicitam ao modelo que essas ações sejam executadas. Também é responsabilidade do controlador notificar a visão sobre modificações dos dados contidos no modelo para que essa atualize a sua exibição.

Atualmente, o padrão MVC está presente em frameworks de desenvolvimento em praticamente todas as linguagens de programação existentes. O sucesso deste padrão deve-se (Buschmann, 1996):

- Facilidade para manutenção e evolução do código devido a separação de responsabilidades.
- Lógica de aplicação (modelo) pode ser testada separadamente de sua interação com o usuário (visão e controlador).
- Diferentes componentes (modelo, visão e controlador) podem ser desenvolvidos em paralelo.

2.2 Framework Android

Com o crescente uso de dispositivos móveis, surgiu a necessidade de garantir a qualidade e rapidez do desenvolvimento de softwares que operam sobre essa plataforma. Dentre todas as plataformas de desenvolvimento utilizadas, destacam-se as de desenvolvimento nativo, ou seja, desenvolvem utilizando todos os recursos exclusivos de algum sistema operacional, e no caso de aplicações mobile são eles *Android* e *iOS*. Neste trabalho será dado foco no desenvolvimento para aplicações no sistema *Android*, utilizando a linguagem de programação *Kotlin*.

Para justificar essa escolha, primeiro é preciso dizer ser certo que sistemas móveis vem cada vez mais dominando o mercado de tecnologia. Com isso posto, e sabendo que um sistema desenvolvido com tecnologia nativa obtém maior desempenho em relação a sistemas híbridos. É certo que os sistemas operacionais móveis mais difundidos no mercado são *Android* e *iOS*, porém nesse trabalho optou-se pelo primeiro, visto que é o com o maior número de usuários pelo mundo, além de ser de código aberto e seus aplicativos poderem ser desenvolvidos em qualquer sistema operacional.

Quanto a escolha da linguagem *Kotlin*, é preciso dizer primeiramente que apenas as linguagens *Java* e *Kotlin* são utilizadas para a construção de sistemas interativos padrões, excluindo jogos, que fogem do escopo deste trabalho. Isto posto, pode-se observar na subseção 2.2.1 que a linguagem *Kotlin* surgiu com a ideia de corrigir alguns problemas de *Java*, e por ser uma linguagem mais moderna e concisa, foi a escolhida para esse sistema.

Segundo Meier (2012), *Android* é uma pilha de software de código aberto que inclui o sistema operacional, middleware e principais aplicativos móveis, junto com uma API para escrever aplicativos que podem moldar a aparência, a sensação e a função dos dispositivos nos quais são executados. De forma simples, pode-se dizer que o *Android* engloba um sistema operacional, juntamente com uma série de recursos e bibliotecas para o desenvolvimento de aplicações que funcionem nesse sistema, além de um ambiente de desenvolvimento, o *Android Studio*.

Aplicações *Android* são normalmente desenvolvidas utilizando as linguagens Java ou Kotlin. Neste trabalho, optamos pela linguagem Kotlin por esta possuir recursos que facilitam o desenvolvimento de aplicações quando comparada com a linguagem Java.

Com o intuito de tratar-se da parte de layout do aplicativo, a ferramenta de desenvolvimento *Android Studio* cria sempre um arquivo da linguagem de marcação *XML* para cada tela criada dentro do aplicativo. Essa linguagem é derivada do HTML, porém mais sucinta, e por isso acredita-se que ela foi escolhida como padrão de desenvolvimento de telas no sistema *Android*, segundo Meier (2012).

Pode-se dizer que, na parte programática do desenvolvimento de aplicações *Android*, que se tem as classes comuns, utilizadas muitas vezes para manipular dados e regras de negócio, além das *Activities* e *Fragments*, utilizadas para programar o conteúdo das telas do aplicativo.

De acordo com Meier (2012), uma *Activity* é a classe base para os componentes visuais

e interativos de uma tela do seu aplicativo. De outra maneira, pode-se dizer que a *Activity* é uma classe que está relacionada com a tela e implementa seus respectivos comportamentos. Conforme Meier (2012), os *Fragments* permitem que o desenvolvedor possa dividir suas atividades em componentes reutilizáveis e totalmente encapsulados, com componentes independentes, porém operando sempre no contexto de uma *Activity*. Ou seja, *Fragments* são subtelas infladas em *Activity*, a fim de diminuir a coesão do código.

No presente trabalho, será detalhado como o ambiente de desenvolvimento *Android* foi explorado para a criação de um sistema de automatização de formulários. Para esse sistema específico, foi criada uma *Activity*, porém a sua ligação com seu arquivo *XML* foi removida, porque o layout dessa tela será adaptável de acordo com as tabelas de campos que a *Activity* recebe por parâmetro. E é importante pontuar que esta foi uma das maiores dificuldades deste projeto, afinal o layout teve que ser criado programaticamente, através de lógica, simulando a adição de elementos visuais em um arquivo *XML*. Ao final desse procedimento, foi gerada uma tela através dessa instância de layout criada artificialmente, com o que este sistema se propôs a fazer.

2.2.1 A Linguagem Kotlin

Kotlin é uma linguagem de programação executada sobre a máquina virtual Java. A linguagem é desenvolvida pela empresa JetBrains[1] conhecida pela criação do IDE InteliJIDEA, amplamente usado para desenvolvimento de aplicações Java (Moskala e Wojda, 2017).

A linguagem Kotlin foi projetada para corrigir alguns problemas conhecidos da linguagem Java para permitir o desenvolvimento de software mais seguro. A principal novidade de Kotlin é ser uma linguagem *null-safe*, isto é, o compilador da linguagem é capaz de detectar possíveis objetos nulos antes da execução do código, evitando assim, erros em tempo de execução. Outro importante aspecto do projeto da linguagem Kotlin é a sua expressividade quando comparada à linguagem Java. Como exemplo da expressividade de Kotlin, considere o seguinte trecho de código Java que define uma classe para armazenar dados de um cliente:

```
public class Client {
   private String name;
   private Int age;

   public Client (String name, Int age) {
      this.name = name ;
      this.age = age;
   }
   public String getName() {
       return name;
   }
```

[1] www.jetbrains.com

```
    public Int getAge() {
        return age;
    }
    public void setName (String name) {
        this.name = name ;
    }
    public void setAge(Int age) {
        this.age = age;
    }
}
```

IDEs Java possuem funcionalidades para automatizar a criação de código repetitivo para manipulação de dados em classes (funções get /set). A linguagem Kotlin possui uma construção específica para esse tipo comum de classe, como se segue:

```
data class Client (var name : String, var age : Int)
```

Apenas essa linha de código define a mesma funcionalidade que todo o código Java apresentado anteriormente: A linguagem Kotlin automaticamente gera funções para manipulação de informações armazenadas em uma classe além de método para conversão em String (método `toString()`), teste de igualdade (`equals()`) e `hashCode`.

Uma característica muito importante de Kotlin é a maneira que ela lida com objetos nulos. Em programas Java é comum encontrarmos uma grande quantidade de código destinado a lidar com a possibilidade de referências nulas para objetos. Em Kotlin esse tipo de situação é contornado por marcar explicitamente quando um objeto pode ter uma referência nula. Isso permite o compilador da linguagem detectar possíveis locais em que um erro de acesso a objeto nulo ocorreria em tempo de execução. Objetos que podem possuir referências nulas são marcados com o operador de chamada segura (representado por um ponto de interrogação, "?"). Os exemplos a seguir ilustram esses conceitos.

```
// erro de compilacao
var notNullClient : Client = null
// Ok
var client : Client? = null
```

A primeira definição de variável especifica um objeto de nome notNullClient de tipo `Client`, que representa objetos não nulos da classe `Client`. Porém, esse valor é inicializado com uma referência nula o que é inválido para o tipo `Client` e, portanto, é rejeitado pelo compilador de Kotlin. Por sua vez, a variável `client` é definida usando o tipo `Client?` que denota referências quaisquer (incluindo nulo) para objetos da classe `Client` Moskala e Wojda (2017).

Outro recurso de Kotlin para lidar com referências nulas é o chamado operador *Elis*, representado por "?:". A ideia do operador "?:"é que esse funciona como uma atribuição

possuindo um valor "padrão"que é utilizado caso o operando da esquerda seja nulo. Como exemplo desse operador, considere:

```
var teste : String? = null
val test2 : String = teste ?: "hello"
```

No trecho de código anterior, temos que o valor `test2` será inicializado com a string `hello` porque o operando esquerdo de `?:` é uma referência nula e, portanto, o resultado será o segundo operando.

Outro aspecto importante em *Kotlin* que facilita o desenvolvimento de aplicações *Android* é a conexão automática de valores presentes no código *Kotlin* com os elementos visuais definidos em arquivos *XML*. Para manipular e definir ações para elementos visuais, basta definir um identificador para o elemento no arquivo *XML*, e utilizar este identificador no código *Kotlin*. Observe que o operador `"?"` é também utilizado para acesso a membros ou métodos dos objetos potencialmente nulos. O trecho de código seguinte ilustra essa característica.

```
class Test {
    fun testFun(){
        // btnTeste representa um ID de um bot o no XML
        // setOnClickListener define a acao de clique do bot o
        btnTeste?.setOnClickListener{
            textViewTeste?.text = "hello world"
            // textViewTeste representa um ID de um texto no XML
        }
    }
}

// XML correspondente a classe Test
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/textViewTeste"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:id="@+id/btnTeste"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</FrameLayout>
```

Pode-se observar que Kotlin é uma linguagem moderna que possui recursos para facilitar o desenvolvimento de aplicações na plataforma Android.

2.3 Modelo Entidade-Relacionamento

Primeiramente, antes de definir o modelo entidade-relacionamento, precisa-se elucidar sobre o que é um sistema de banco de dados. Segundo Date (2004), se trata de um sistema de armazenamento de dados baseado em computador, ou seja, um sistema cujo objetivo global é registrar e manter informação.

Em seguida, é necessário definir o que é uma entidade e o que é um relacionamento em um sistema de banco de dados. Em HEUSER (1998), é definido que entidade representa conjunto de objetos da realidade modelada sobre os quais deseja-se manter informações no banco de dados. Enquanto ainda segundo HEUSER (1998), um relacionamento é uma relação entre entidades distintas dentro do sistema de banco de dados.

Com isso, pode-se estabelecer o modelo entidade-relacionamento para um sistema de banco de dados. De acordo com HEUSER (1998), essa modelagem basicamente permite criar um diagrama contendo entidades e relacionamentos, formando assim um esquema relacional de banco de dados. Através desse diagrama gera-se um modelo de entidade-relacionamento para estruturar o banco de dados a partir desse modelo.

Acerca dos relacionamentos nesse modelo, tem se que eles podem ser classificados em três categorias: 1:1(um-para-um), 1:n(um-para-muitos) ou n:m(muitos-para-muitos). Essas categorias dizem se a entidade compartilha apenas uma instância dela, ou várias, com a outra entidade do relacionamento, e o mesmo vale ao contrário. De forma intuitiva, a categoria 1:1 é utilizada quando duas entidades se relacionam através de apenas um item de cada uma delas, enquanto a categoria 1:n relaciona um item de uma entidade com vários de outra, e por fim m:n junta um número m de itens de uma entidade com outro número n de itens de outra entidade. A figura 2.2 exemplifica cada um desses casos, para que fique mais claro o entendimento.

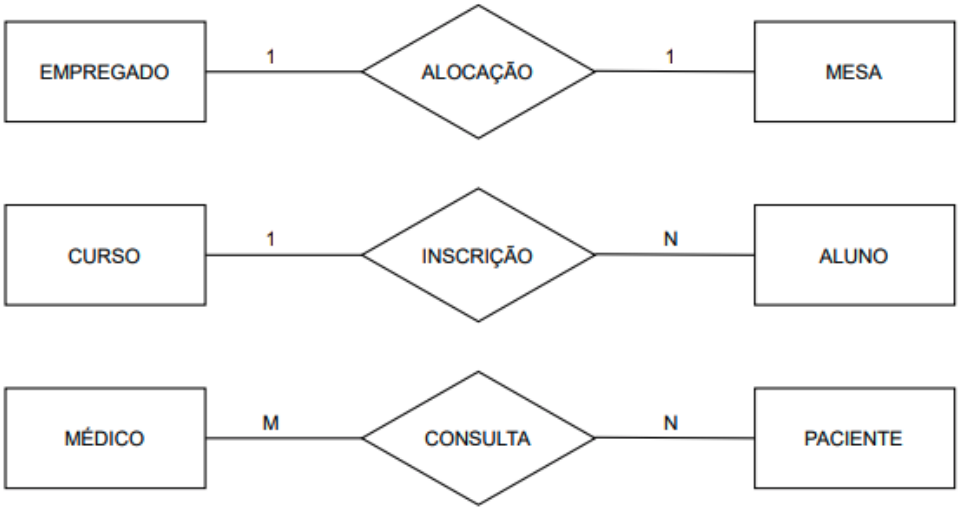

Figura 2.2: Exemplos de relacionamentos em um banco de dados

Para este presente trabalho, será englobado apenas as categorias 1:1 e 1:n do modelo entidade-relacionamento, pois são as categorias que englobam a maioria dos formulários necessários para sistemas interativos. Porém para trabalhos futuros, é pensado a adição da possibilidade de criação de formulários m:n, a fim de englobar todo esse modelo relacional.

2.4 Trabalhos relacionados

É notório que o presente trabalho trouxe à tona um tema que se torna cada vez mais relevante no âmbito da engenharia de software, uma vez que visa de alguma forma automatizar parte do desenvolvimento de sistemas. Contudo, na fase de concepção deste projeto, foram buscadas inspirações em outros projetos, com abordagens muitas vezes diferentes para este mesmo problema proposto. Nesta seção serão pontuados alguns trabalhos relacionados, a fim de embasar esse estudo.

Em Lima (2014), o autor criou um gerador de código padrão para aplicações *WEB* básicas, a fim de servir como uma versão inicial para um produto. No texto, nota-se que seu foco foi atingir o meio empresarial, uma vez que sua principal motivação para este trabalho foi maximizar os lucros das empresas sobre os produtos de desenvolvimento. Ao final do trabalho, o escritor constatou que seu gerador consegue diminuir o tempo de criação de um sistema *WEB* em até 39 dias, porém o software ainda não possui documentação, o que dificulta o uso para desenvolvedores menos experientes.

De forma que em alguns aspectos se assemelha com a abordagem deste presente trabalho, Monteiro (2016) se propôs a criar um gerador de código *WEB*, utilizando a linguagem de programação Java, no qual se baseava em um esquema de entidade-relacionamento. O software recebe esse esquema como entrada, e gera as classes correspondentes para cada entidade, além de montar um template de projeto com o padrão arquitetural MVC, a fim de otimizar ainda mais o trabalho do desenvolvedor.

Segundo Shimabukuro (2006), um dos problemas dos geradores de aplicação é o seu alto custo de desenvolvimento. Com isso posto, o autor completa que os geradores em sua maioria se propõem a servir a domínios específicos no âmbito do desenvolvimento de softwares. Desse modo, Shimabukuro (2006) apresenta uma ferramenta que é responsável por facilitar a construção de geradores específicos. Entretanto, essa ferramenta ainda se limitava a criar geradores apenas dos tipos: persistência de dados e gestão de recursos, limitando assim o escopo do sistema.

Bem como foi referenciada a metodologia em Monteiro (2016), o trabalho Vieira (2012) foi feito com uma abordagem que tem por fim a geração de código por meio de um esquema de banco de dados, porém com o diferencial de ter foco em aplicações mobile, utilizando o sistema *Android*. Nesse caso, o autor apresentou uma aplicação que gera as classes correspondentes ao esquema de entrada, utilizando os padrões de desenvolvimento do *Android*.

Tal qual se propõe os projetos (Monteiro, 2016) e (Vieira, 2012), o trabalho (Pores, 2011) apresenta um gerador de código a partir de uma entrada. Contudo, essa abordagem adiciona a liberdade do desenvolvedor escolher em qual linguagem o código vai ser gerado, bem como a preferência por algum padrão arquitetural de projeto. Além disso, o sistema promete gerar códigos mais sucintos, a fim de uma menor curva de aprendizado por parte do desenvolvedor.

Perante o exposto, pode-se dizer que existem diversas formas de se automatizar o desenvolvimento de uma aplicação. Para este presente trabalho, foi questionada a demora na construção de formulários dentro de aplicações, focando no sistema *Android*. Para tentar resolver essa questão, usou-se uma abordagem parecida com alguns dos trabalhos citados acima, no sentido de receber uma entrada, e processar esses dados na geração do código. No entanto, diferente dos trabalhos referenciados nesta seção, este projeto visa retornar um formulário completo, eliminando a necessidade do desenvolvedor fazer alterações no que foi gerado.

Capítulo 3

Desenvolvimento

Neste capítulo, descreveremos a estrutura do protótipo implementado, bem como os métodos a serem utilizados quando o sistema for disponibilizado como uma biblioteca. Foi necessário o desenvolvimento de uma infraestrutura que permita a representação de qualquer esquema relacional 1:1 ou 1:n. A representação genérica de tabelas utilizada neste trabalho é detalhada na seção 3.1. A respectiva construção dos formulários, utilizando a representação definida na seção 3.1, é detalhada na seção 3.2. Além disso, na seção 3.3 foi descrito como o protótipo foi implementado seguindo o Padrão Arquitetural MVC. Finalmente, foram feitos experimentos computacionais com o sistema, e os mesmos foram elucidados na seção 3.4.

3.1 Representação de tabelas

De maneira intuitiva, podemos entender uma tabela de um banco de dados como uma sequência de seus campos, em que cada campo é formado por um identificador (nome) e do tipo de dados por ele armazenado. Para ilustrar nossa representação de tabelas, vamos usar um exemplo. Considere o seguinte diagrama entidade-relacionamento [1] de uma tabela que armazena dados sobre alunos.

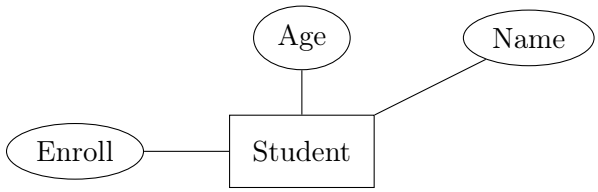

Figura 3.1: Tabela de exemplo

A tabela Student possui três atributos (campos): Name, que armazena o nome do aluno e, portanto representa um valores de tipo String; Age, que armazena a idade do aluno e, portanto

[1] Diagramas entidade-relacionamento são amplamente utilizados para a modelagem de bancos de dados. Mais sobre diagramas pode ser encontrado em Silberschatz et al. (2006).

representa valores de tipo inteiro e Enroll, um booleano que identifica se o aluno está ou não matriculado em um determinado momento.

Como tabelas são formadas por campos, primeiramente devemos definir como estes serão representados. Em nossa solução, utilizamos o tipo de dados `FieldSchema`, que armazena o nome e o tipo de um determinado campo, bem como uma variável que informa se o campo é ou não obrigatório para o formulário. Além disso, possui variáveis que armazenam os identificadores dos elementos de layout XML que representam respectivamente o nome e o valor daquele campo.

Para este trabalho, foi tratado apenas se o campo pode ou não ser vazio na tabela como tratamento de erros. Quanto ao tamanho máximo dos elementos de cada tipo, a própria linguagem Kotlin se encarrega de permitir apenas entradas válidas para cada tipo específico.

A seguir apresentamos a definição da classe `FieldSchema`.

```
data class FieldSchema (
    val fieldName: String?,
    val fieldType: String?,
    val isFieldRequired: Boolean = true,
    var fieldIdName: Int = 0,
    var fieldIdType: Int = 0
)
```

Algorithm 3.1: Representação de um campo da tabela

O campo `fieldName` armazena o nome do campo representado e `fieldType` o tipo deste campo. O valor `isFieldRequired` especifica se o campo é ou não obrigatório e `fieldIdName` e `fieldIdType` representam identificadores de elementos no layout XML do aplicativo.

A seguir, apresentamos um exemplo de representação de um campo obrigatório chamado "Name", de tipo string.

```
FieldSchema(fieldName = "Name", fieldType =  "String", isFieldRequired
    = true)
```

O leitor atento deve ter percebido que as variáveis `fieldIdName` e `fieldIdType` não foram inicializadas no exemplo anterior. Isso é proposital pois esses valores são preenchidos quando da inicialização da interface do formulário. A seguir, apresentamos as definições dos campos para armazenar a idade e o booleano para determinar se o aluno está ou não matriculado.

```
FieldSchema(fieldName = "Age", fieldType =  "Int", isFieldRequired =
   true)

FieldSchema(fieldName = "Enroll", fieldType =  "Boolean",
   isFieldRequired = true)
```

Usando a representação de campos podemos definir uma tabela como sendo formada por um nome e uma lista de campos.

```
data class FormSchema (
    val nameForm: String,
    val fieldsForm: List<FieldSchema>
)
```

Dessa forma, o esquema relacional descrito na Figura 3.1 pode ser presentado pelo seguinte valor do tipo `FormSchema` que utiliza os campos previamente definidos:

```
var form : FormSchema =
    FormSchema(nameForm = "Student", fieldsForm = listOf(
        FieldSchema(fieldName = "Name",
                    fieldType =  "String",
                    isFieldRequired = true
                    ),
        FieldSchema(fieldName = "Age",
                    fieldType =  "Int",
                    isFieldRequired = true
                    ),
        FieldSchema(fieldName = "Enroll",
                    fieldType =  "Boolean",
                    isFieldRequired = true
                    )
    ))
```

Como este trabalho se propôs a criar formulários a partir de tabelas relacionais 1:1 e 1:n, foi criado uma terceira estrutura, denominada `ListFormSchema`, que contém uma variável de nome, do tipo String, com nome de `nameListForm` e uma lista de tabelas, do tipo List<FormSchema>, chamada de `listFormChildren`. Pode se dizer que essa estrutura representa uma lista de tabelas, e possui um nome como identificador dessa lista. A definição do tipo `ListFormSchema` é apresentada abaixo.

```
data class ListFormSchema(
    val nameListForm: String,
    val listFormChildren: List<FormSchema>
)
```

Para fins de exemplificação, iremos criar um esquema semelhante ao da Figura 3.1, porém para endereço, denominado "Address", deste estudante. Neste esquema relacional, pode-se ter um campo "Street", do tipo String que representa a rua, e um campo "Number", do tipo inteiro, que retorna o número da casa, sendo ambos os campos obrigatórios para este formulário. Agora juntando os formulários "Student"e "Address", criamos a lista de formulários "Enrollment", que representa a matrícula de um estudante juntamente com seu endereço em uma instituição de ensino. Pode-se verificar esse esquema relacional abaixo:

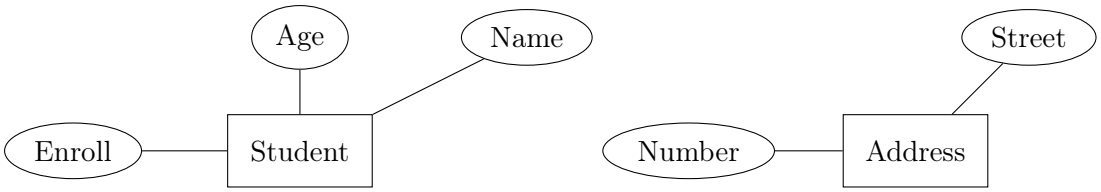

Figura 3.2: Esquema de exemplo da estrutura Enrollment

Em termos de código, foi feito essa estrutura que comtempla todos esses dados de forma que eles se relacionam, embora cada um dos formulários tenha seus campos definidos de forma independente. Segue abaixo a estrutura "Enrollment"feita no código:

```
var enrollment : ListFormSchema =
    ListFormSchema(
        nameListForm = "Enrollment",
        listFormChildren = listOf(student, address)
    )
```

3.2 Construção dos formulários

Utilizando a estrutura de tabelas apresentada na seção anterior, podemos automatizar a construção de formulários através de uma função criada no código, denominada `setFormLayout (listFormSchema: ListFormSchema)`, que recebe uma lista de formulários de entrada, e retorna um elemento completo e estruturado de layout para ser inflado na tela do aplicativo. É preciso salientar que esse método é público na classe, logo será um dos métodos disponíveis na biblioteca deste sistema. A figura 3.3 resume o funcionamento desta função.

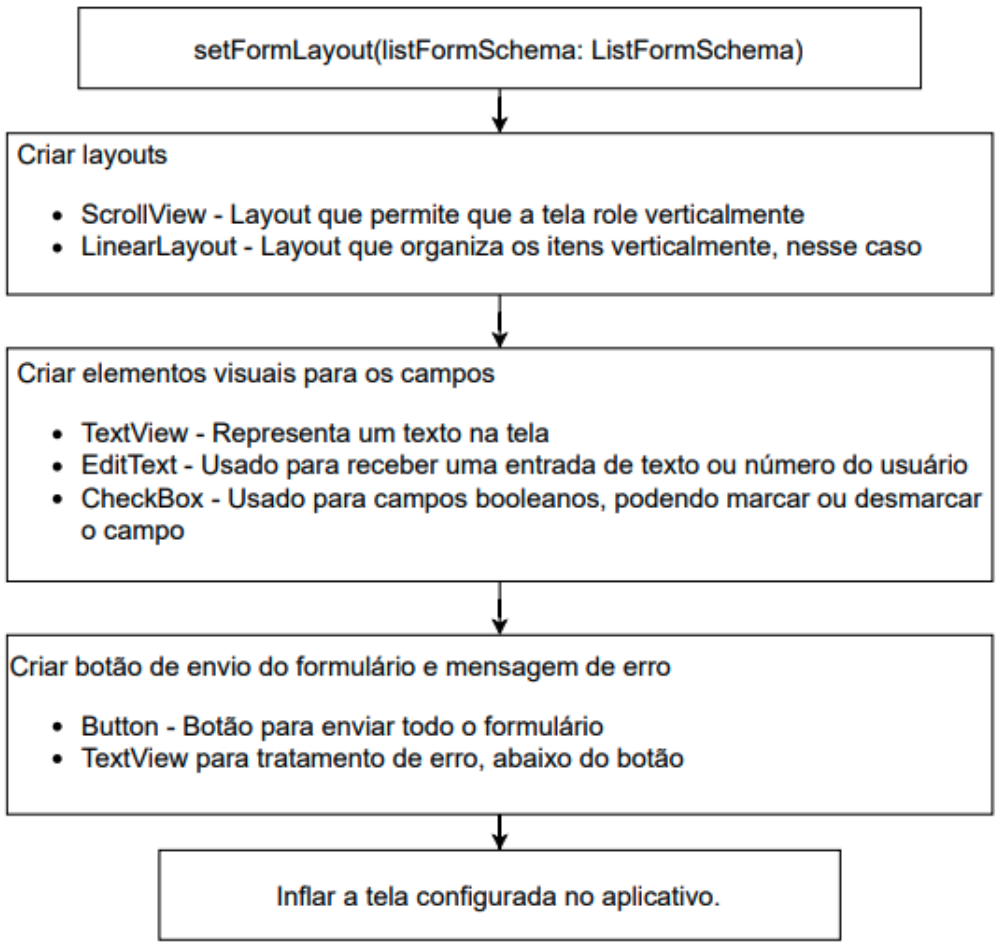

Figura 3.3: Passo a passo da função setFormLayout

Primeiramente, a função `setFormLayout` instancia um LinearLayout com orientação vertical, informando que os formulários serão postos na tela alinhados de cima para baixo. Primeiramente, pra criar um layout é preciso definir seu tamanho, e no caso foi definida a largura como `MATCH_PARENT`(Preenche toda a largura da tela), e a altura como `WRAP_CONTENT`(Preenche apenas o espaço necessário da tela). Para se ter uma ideia de como é feita uma instância de um layout programaticamente, temos abaixo a implementação da instância desse LinearLayout:

```
val layoutList = LinearLayout(this)
layoutList.layoutParams = LinearLayout.LayoutParams(
    ViewGroup.LayoutParams.MATCH_PARENT,
    ViewGroup.LayoutParams.WRAP_CONTENT
)
layoutList.orientation = LinearLayout.VERTICAL
```

Algorithm 3.2: Exemplo de instância de layout

Na sequência, o método inicia um laço de repetição que percorre todas as tabelas da lista que vem como entrada. Para cada tabela, primeiro instancia-se um LinearLayout para esta tabela, também com orientação vertical, semelhante ao layout 3.2. Após isso, cria-se um TextView para mostrar o título daquela tabela, e adiciona esse elemento de visualização ao layout daquela tabela, conforme mostrado na figura 3.3.

Posteriormente, cria-se outro laço de repetição responsável por percorrer todos os campos da presente tabela, e em cada campo adiciona-se um TextView referente ao nome daquele campo, e um outro elemento correspondente ao valor daquele campo, sendo um EditText para os casos do campo esperar uma String ou Int, ou um CheckBox para o caso do campo esperar um booleano. Quando a função percorrer todos os campos daquela tabela, encerra aquele laço de repetição, adicionado aquele layout criado para essa tabela ao layout de lista de tabelas, conforme 3.2 e passa para a próxima tabela.

Terminando o último laço de repetição, primeiramente foi adicionado um Button para enviar aquele formulário para validação, com um TextView reservado a mensagens de erro dos campos, caso algum campo obrigatório não tenha sido preenchido. Por fim, foi instanciado um layout rolável(NestedScrollView), para os casos que os formulários ocupem um espaço maior que o da tela do celular, e foi adicionado o layout de listas de tabelas contendo todos elementos visuais de todas as tabelas à esse NestedScrollView. Com isso, a função `setFormLayout` se encerra retornando esse layout NestedScrollView, para que ele seja inflado na tela.

Com a tela de formulário instanciada e inflada na tela do celular, foi criada uma função `setClicksBtnDone()` para criar os eventos de clique dos botões de enviar cada tabela para validação, com as regras de negócio de tratamento de erros dos campos obrigatórios da respectiva tabela. Para que a biblioteca envie o formulário preenchido quando o mesmo for enviado, foi criado o método público lstinline|setCallBackForm(callBack: (ListFormSchema) -> Unit)|, que passa uma função de ordem superior como parâmetro, para que o desenvolvedor possa tratar a ação de retorno do formulário. Esta função é chamada no clique do botão do formulário, em caso de todos os campos seguirem as regras estipuladas. Posto isso, está concluído este protótipo para automatização de formulários com uma ou mais tabelas, em aplicações Android.

3.3 Padrão MVC no sistema

Para fins de organização de código, e para adequar o projeto à algum padrão arquitetural, nesta seção será detalhado como este trabalho seria separado por um Model, uma View e um Controller. Para isso, primeiramente definimos as estruturas `FieldSchema`, `FormSchema` e `ListFormSchema` para a representação das tabelas no Model, e para que essas tabelas preenchidas cheguem no Model a fim de serem processadas pelo automatizador de formulários.

Primeiramente, para definirmos o Model, deve-se criar uma classe de repositório, e definir

de onde deve vir a lista de tabelas preenchida, sendo através de uma requisição de API REST ou já salva em banco local. No caso da aplicação optar por comunicação com o servidor através do protocolo REST, é recomendado utilizar a biblioteca para Android chamada retrofit. Esta biblioteca facilita todo o processo de envio de requisição, recebimento de retorno desta requisição nos casos de erro e sucesso, além dos tratamentos de erro. Caso o sistema seja feito para receber esta tabela através de um banco local, recomenda-se utilizar bibliotecas como a "Room"ou a "Realm", ambas feitas para facilitar o acesso e gerenciamento do banco de dados do aplicativo.

Na sequência, precisa-se definir o Controller, que no caso existiria apenas um método para chamar a função do repositório que é responsável por retornar a lista de tabelas. Esse método do Controller deve salvar essa tabela e enviar para a camada da View, através de outro método nesta mesma classe.

Por fim, a View se responsabiliza por chamar o método do Controller no qual retorna a lista de tabelas carregada no Model, e processa essa lista, através do método `setFormLayout`. Após a lista ser processada, a View é inflada na tela já com todas as especificações do formulário, e se responsabiliza pelo tratamento de erros e validação de todas as tabelas desse formulário.

3.4 Experimentos Computacionais

Nesta seção apresentaremos alguns experimentos com o protótipo desenvolvido. A forma adotada para testar este sistema foi mostrando, em diferentes casos, como o gerador de formulários se comportou. Como um primeiro exemplo, vamos considerar uma tabela formada apenas por campos de tipo String.

Tabela 3.1: Tabela com campos dos três tipos

Nome	Tipo
Nome	String
Idade	Int
PCD	Boolean

Considerando a tabela acima, e lembrando dos detalhes de desenvolvimento expostos na seção 3.2, pode-se imaginar que a partir desse esquema, o software vai gerar um formulário com todos os tipos permitidos nesse trabalho(String, Int e Boolean), e um botão de envio do formulário. Segue abaixo a demonstração do que foi dito.

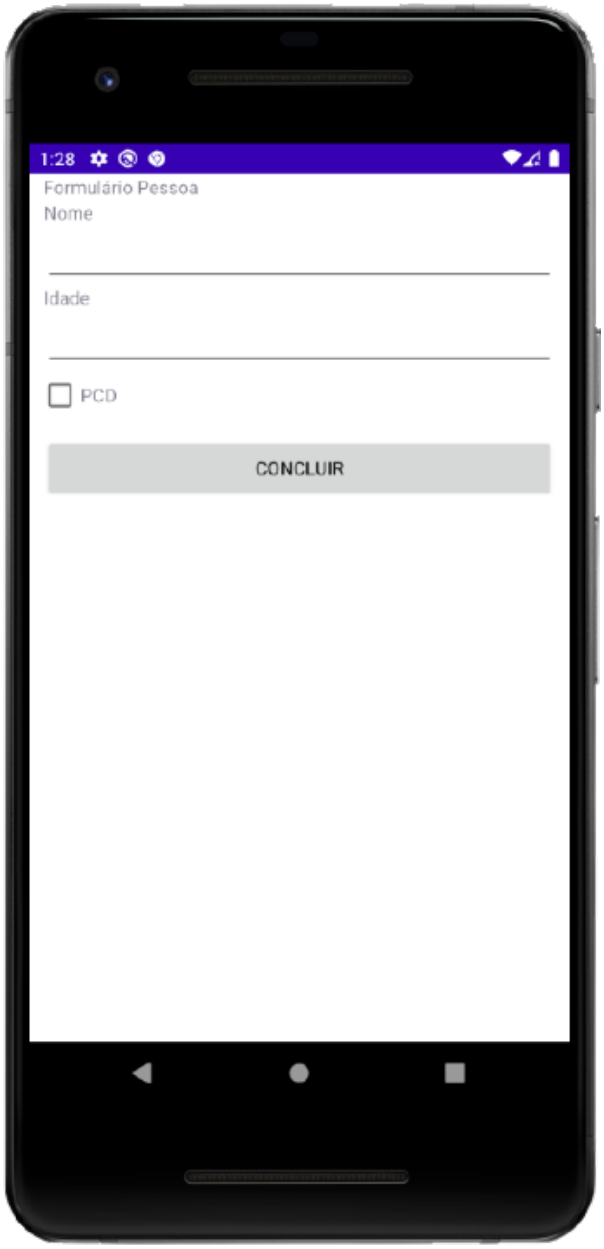

Figura 3.4: Tabela com campos dos três tipos

Concebendo agora um esquema no qual é formado por mais tabelas, tem-se que além do formulário *Pessoa*, exemplificado em 3.4, pode-se adicionar outra tabela, denominada cachorro, no qual esse cachorro tem relação com a pessoa. Lembrando que, o número de cachorros pode ser infinito, de acordo com a entrada fornecida pelo desenvolvedor.

Isto posto, tem-se que uma entidade Cachorro possui os campos Nome e Raça, ambos do tipo String. Agora considera-se que a tabela será carregada para um cachorro para a pessoa. Portanto, segue o exemplo de como seria gerado esse formulário de mais de uma tabela.

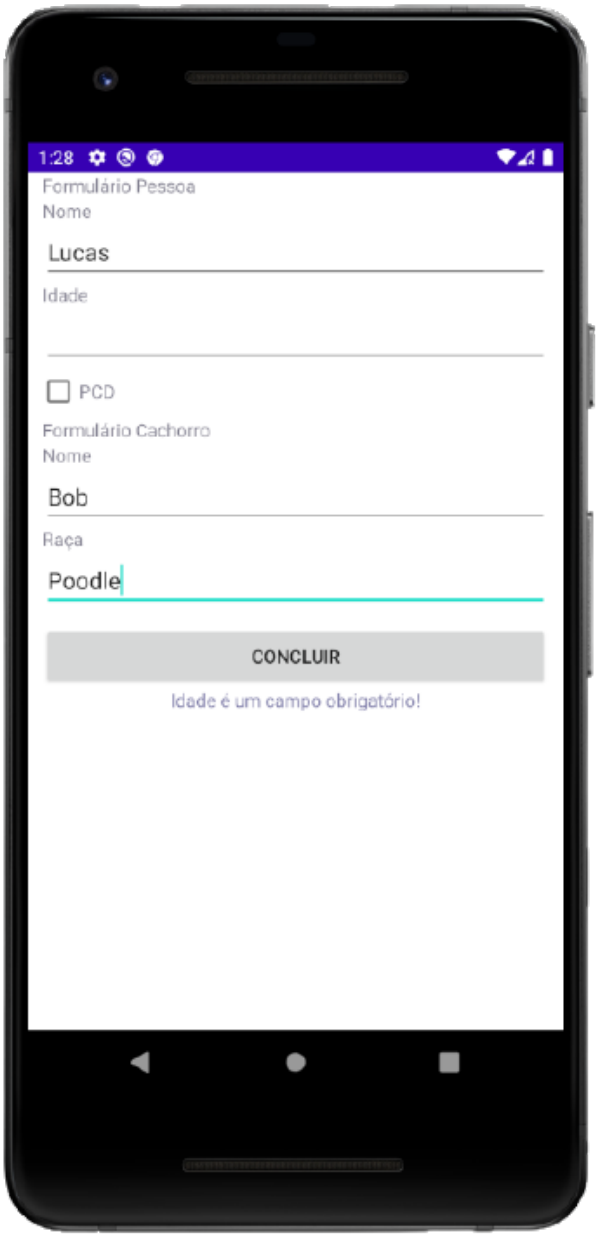

Figura 3.5: Formulário com duas tabelas

Nota-se na imagem 3.5 que ela possui todos os campos previamente informados, bem como exemplifica um tratamento de erro já embutido neste protótipo. Para esse formulário foram colocados os campos Nome e Idade da entidade Pessoa como campos obrigatórios para o formulário, lembrando que essa informação vem na estrutura de entrada, explicada na seção 3.1.

Com isso, é possível dizer que o sistema se demonstrou satisfatório para formulários cujo tipo dos campos esteja entre os permitidos. Além disso, conseguiu-se mostrar como o tratamento de erros funciona para o caso de campos obrigatórios estarem vazios. Logo foi demons-

trado que, dada uma entrada do tipo `ListFormSchema`, o sistema consegue gerar um formulário com todas as especificações citadas, com uma ou mais tabelas.

3.5 Conclusão

Neste capítulo apresentamos os detalhes de implementação do protótipo de formulários adaptáveis. Para esta versão, consegue-se criar formulários para uma ou mais tabelas, com validação individual e tratamento de erros para cada campo das tabelas. Os limites desse protótipo estão nos tipos suportados, já que este sistema automatiza apenas formulários com campos dos tipos String, Int e Bool.

Capítulo 4

Conclusão

4.1 Considerações Finais

A demanda do desenvolvimento de sistemas mobile vem crescendo ano a ano, e a maioria desses sistemas possuem formulários para manipulação de informações. Além disso, sabe-se que o custo de tempo para a construção, na maioria das vezes de forma mecânica, desses formulários é alto. Nesse sentido, torna-se imprescindível automatizar a construção deste tipo de software. Neste trabalho foi construído uma biblioteca Android interativo que permite a construção de sistemas de software que se adaptam à uma ou mais tabelas de entrada, criando um formulário de acordo com a estrutura destas tabelas. Para a implementação deste trabalho, optou-se pela linguagem Kotlin.

Acerca das limitações deste trabalho, temos que o trabalho não possui um servidor para backend integrado a ele, uma vez que ele foi pensado para se tornar uma biblioteca Android, dando assim a liberdade do desenvolvedor escolher se as tabelas proverão de um servidor ou do seu próprio banco de dados local. Outra restrição é existir essa funcionalidade apenas para o sistema Android, e não para iOS e Web, por exemplo.

Atualmente, o protótipo considera esquemas formados por uma ou mais tabelas e com um número limitado de tipos de campos. Apesar dessas limitações, consideramos que o protótipo cumpriu seus objetivos ao mostrar que a proposta é viável e automatiza completamente o desenvolvimento de tais formulários.

4.2 Trabalhos futuros

Como trabalho futuro, almeja-se construir um sistema que consiga se adaptar a todo de um sistema de um banco de dados relacional. Além disso, também pretende-se criar um sistema back-end para se comunicar através de uma API REST com esse sistema Android criado neste trabalho, deixando o projeto mais robusto e modularizado. Por fim, pretende-se incrementar esse trabalho atual, tendo um sistema autoadaptável a um banco de dados

relacional, que possui comunicação com o back-end de através de um arquivo JSON, que conterá todo o esquema do banco de dados para ser processado pelo aplicativo e transformá-lo em formulários interativos.

Outra pretensão futura é a concepção desse sistema também para a plataforma iOS, a fim de abranger um número maior de dispositivos. Por fim, pretende-se incluir todos os tipos que faltaram no tratamento dos formulários, uma vez que atualmente são suportados apenas os tipos String, Int e Bool.

Referências Bibliográficas

Burbeck, S. (1987). Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc).

Buschmann, F. (1996). *PatternOriented-Software-Architecture-A-System-of-Patterns-Volume-1*, volume 1. Ashish Raut.

Date, C. J. (2004). *Introdução a sistemas de bancos de dados*. Elsevier Brasil.

Gamma, E.; Helm, R.; Johnson, R. e Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HEUSER, C. A. (1998). Projeto de banco de dados, 6ª edição. *Instituto de Informática. Editora Sagra*.

Laudon, K. e Laudon, J. (2001). *Gerenciamento de sistemas de informação*. LTC.

Lima, A. (2014). *XGen - Um Gerador de aplicações para a otimização do tempo de desenvolvimento de MVPs para startups*. UEPB.

Meier, R. (2012). *Professional Android 4 application development*. John Wiley & Sons.

Monteiro, C. (2016). *Gerador de códigos para o desenvolvimento de aplicações web a partir da modelagem Entidade-Relacionamento*. Exacta.

Moskala, M. e Wojda, I. (2017). *Android Development with Kotlin*. Packt Publishing Ltd.

Pores, S. (2011). Gerador de código multilinguagem e multiplataforma.

Shimabukuro, E. (2006). *Um gerador de aplicações configurável*. USP.

Silberschatz, A.; Korth, H. e Sudarshan, S. (2006). *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edição.

Vieira, C. (2012). *Gerador de código para aplicativo Android*. FATEC.