Instituto Federal de Educação, Ciência e Tecnologia de São Paulo
Câmpus São João da Boa Vista

Wendreo Luciano Fernandes

Análise de Usabilidade da Linguagem de Programação *Kotlin* na Plataforma *Android*

São João da Boa Vista - SP
2019

Wendreo Luciano Fernandes

Análise de Usabilidade da Linguagem de Programação *Kotlin* na Plataforma *Android*

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, como parte dos requisitos para a obtenção do grau de Especialista em Desenvolvimento de Aplicações para Dispositivos Móveis.

Área de Concentração: Desenvolvimento de Sistemas

Orientador: Prof. Dr. Breno Lisi Romano

São João da Boa Vista - SP

2019

Folha destinada à inclusão da Catalogação na Fonte - Ficha Catalográfica (a ser solicitada à Biblioteca IFSP – Câmpus São João da Boa Vista e posteriormente impressa no verso da Folha de Rosto (folha anterior).

Catalogação na Fonte preparada pela Biblioteca Comunitária "Wolgran Junqueira Ferreira"

do IFSP – Câmpus São João da Boa Vista

Dados da ficha

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO
CÂMPUS SÃO JOÃO DA BOA VISTA
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

Formulário de Avaliação Final de TCC

Estudante: _____ Prontuário: _____

Título: _____
_____
_____

Tipo do Trabalho: ( ) Monografia ( ) Artigo ( ) Relatório Técnico

Professor Orientador: _____

Nome	Aprovado	Reprovado	Assinatura
	( )	( )	
	( )	( )	
	( )	( )	
Resultado:	( )	( )	

O orientador somente registrará o resultado e assinará este formulário após a entrega da versão final corrigida.

São João da Boa Vista, ___/___/_____

_____ _____
Orientador Coordenador

Dedico este trabalho a minha mãe Sandra por toda paciência, compreensão, carinho, amor, sacrifícios e esforços realizados para que esse momento se tornasse possível

AGRADECIMENTOS

Primeiramente a Deus, por estar comigo em todos os momentos. A minha irmã Eloá e minha companheira Bianca pelo companheirismo e paciência. Aos meus colegas, professores e coordenadores de graduação e pós-graduação. Cada um de vocês teve um papel importante em meu desenvolvimento pessoal e profissional.

"*... se você passar muito tempo pensando, você nunca irá fazer nada.*"

Bruce Lee

RESUMO

A evolução dos dispositivos móveis afeta não somente os seus usuários comuns, mas também todos os profissionais do ramo de tecnologia, por traz de toda a arquitetura de uma aplicação, seja ela simples ou *rocket science*. A escolha da linguagem de programação deve levar em consideração inúmeros fatores, desde conhecimento e experiência do(s) envolvido(s), até as peculiaridades referentes ao ecossistema no qual almeja-se atingir. O presente trabalho de conclusão de curso, tem o objetivo de apresentar um estudo quali-quantitativo entre as linguagens de desenvolvimento de *software Java* e *Kotlin*, com ênfase nos algoritmos utilizados na criação de aplicações móveis para os usuários da plataforma móvel *Android*, com o objetivo de efetuar a constatação de qual tecnologia necessita de menores quantidades de caracteres, linhas de codificação e tamanho em *bytes*, por meio de experimentos específicos como programas de Olá Mundo!, classes, operações matemáticas e por fim a elaboração de uma aplicação que efetua a conversão de medidas termométricas, implementado nas linguagens analisadas, apoiando-se no padrão de projeto Adapter. Dados os experimentos utilizados, obteve-se o resultado de que por meio da linguagem Kotlin, torna-se possível chegar aos mesmos resultados com Java, porém com a menor necessidade códigos clichês e desnecessários.

Palavras-chave: Kotlin. Java. Desenvolvimento. Tecnologia.

ABSTRACT

The evolution of mobile devices affects not only its ordinary users, but also all professionals in the field of technology, behind all the architecture of an application, be it simple or rocket science. The choice of the programming language should take into account innumerable factors, from the knowledge and experience of the involved ones, to the peculiarities regarding the ecosystem in which one aims to attain. The aim of this work is to present a qualitative and quantitative study of Java and Kotlin software development languages, with emphasis on the algorithms used in the creation of mobile applications for users of the Android mobile platform, with the objective to make the verification of which technology needs smaller amounts of characters, coding lines and size in bytes, through specific experiments like Hello World !, classes, mathematical operations and finally the elaboration of an application that makes the conversion of thermometric measurements, implemented in the languages analyzed, based on the Adapter design pattern. Given the experiments used, we obtained the result that through the Kotlin language, it is possible to reach the same results with Java, but with the least need unnecessary codes.

Keywords: Kotlin. Java. Development. Technology.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TDD	Test Driven Development (Desenvolvimento Orientado por Testes)
I/O	Input/Output (Entrada/Saída)
iOS	iPhone OS (Sistema operacional móvel da Apple)
HTML	Hypertext Markup Language (Linguagem de Marcação de Hipertexto)
CSS	Cascading Style Sheets (Folhas de Estilo em Cascata)
WWW	World Wide Web (Rede Mundial de Computadores)
GPS	Global Localization System (Sistema de Posicionamento Global)
XML	Extensible Markup Language (Linguagem de Marcação Extensível)
UML	Unified Modeling Language (Linguagem de Modelagem Unificada)
CRUD	Acrônimo para create, read, update e delete
macOS	Machintosh OS (Sistema operacional desktop da Apple)

SUMÁRIO

1 INTRODUÇÃO

Em relação ao desenvolvimento de *software* voltado para dispositivos móveis, existem inúmeras opções de tecnologias. Para criar aplicativos nativos voltados para usuários da plataforma *Android*, pode-se utilizar a linguagem *Java* por meio do Ambiente de Desenvolvimento Integrado (*Integrated Development Environment - IDE*) *Android Studio*. Já para o desenvolvimento nativo voltado para usuários de *iOS*, pode-se optar pela linguagem *Swift* ou *Objective-C* por meio da *IDE Xcode*. Para criar aplicações híbridas (*Android* e *iOS*) existem mais opções, como o *C#* e o *.NET Framework* por meio do plugin *Xamarin* disponível na *IDE Visual Studio*, ou o *Ionic Framework* utilizando *HTML*, *CSS*, *Node.js*, *TypeScript* e *Apache Cordova* utilizando as *IDE's Android Studio* e *Xcode*. Cada tecnologia citada anteriormente possui suas especificidades como ecossistema próprio, sintaxe, nomes, tipagem, semântica, um ou mais paradigmas de programação e curva de aprendizado. Com a variedade de opções, pode ser difícil escolher a ideal para cada situação/contexto.

Tucker e Noonan (2009) disseram que,

> Talvez a maior motivação para o desenvolvimento de linguagens de programação nas últimas décadas tenha sido a rápida evolução da demanda de poder computacional e as novas aplicações por parte de uma grande e diversa comunidade de usuários.

Segundo Fowler (2009), "Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender". Independente da tecnologia, o propósito sempre é solucionar um problema e/ou atender uma necessidade específica. Porém, apenas escrever algoritmos não é o suficiente, códigos não endentados, variáveis e operações com nomenclaturas pouco intuitivas, excesso ou falta de comentários e operações muito extensas e com várias responsabilidades podem ocasionar na dificuldade de manutenção e falta de flexibilidade.

Martin (2009) cita que,

> "[...] A lógica deve ser direta para dificultar o encobrimento de *bugs*[1], as dependências mínimas para facilitar a manutenção, o tratamento de erro completo de acordo com uma estratégia clara e o desempenho próximo do mais eficiente de modo a não incitar as pessoas a tornarem o código confuso com otimizações sorrateiras. O código limpo faz bem apenas uma coisa.

Linguagens de programação mais verbosas como por exemplo o *Java*, precisam de mais palavras e/ou palavras mais longas e mais símbolos que o necessário para expressar

[1] Defeito, falha ou erro no código de um programa que provoca seu mau funcionamento.

adequadamente a intenção do código, o que não é necessariamente um problema, mas sim a característica da linguagem. Uma tentativa de diminuir a verbosidade tão comum da linguagem *Java* removendo códigos desnecessários para obter algoritmos mais concisos e sucintos foi a criação da biblioteca *Project Lombok*. Juneau (2009-2019) diz que, "A biblioteca substitui o **código clichê** por anotações fáceis de usar, [...] tornando o código mais fácil de ler e menos sujeito a erros e tornando os desenvolvedores mais produtivos"[2], porém como não houve nenhuma modificação na linguagem, código clichê pode acabar resultando em **mau cheiro de código** (*code smell*). Segundo Turini (2014), "Mau cheiro é o termo utilizado quando um determinado código possui algum indício de que está precisando ser **refatorado**".

De acordo com Fowler (2009), "[...] o propósito da refatoração é tornar o *software* mais fácil de entender e modificar". Em 1997 foi criada uma metodologia ágil denominada de Programação Extrema (*Extreme Programming* - XP), com intuito de facilitar o desenvolvimento de *software* e aprimorar a qualidade dos projetos, sendo a refatoração de código, umas das principais práticas utilizadas. Em 1999 foi criado o Desenvolvimento Orientado a Testes (*Test Driven Development* - TDD), onde na primeira etapa se escreve o teste para uma funcionalidade ainda inexistente, na segunda etapa se escreve a funcionalidade a ser testada e na terceira e última o código é refatorado, diminuindo assim futuros *bugs*. Segundo Guerra (2018), o TDD "é uma técnica de desenvolvimento e projeto de *software* na qual os testes são criados antes do código de produção. É uma técnica que vem se tornando cada vez mais popular no mercado e mais atenção da academia".

Conforme Martin (2009), "Não basta escrever um código bom. Ele precisa ser mantido sempre limpo". Devido à grande variedade de linguagens, tecnologias, metodologias de desenvolvimento disponíveis no mercado, até mesmo falta de tempo, escolher a linguagem de programação ideal para resolução de um determinado problema e escrever um código limpo independente do paradigma pode ser uma tarefa árdua para desenvolvedores de todos os níveis.

1.1 Justificativa

Durante o *Google* I/O 17, uma conferência anual voltada para o desenvolvimento de aplicações para os seus próprios sistemas operacionais, mais especificamente o *Android*, a linguagem ***Kotlin*** entrou para o seleto grupo de linguagens oficiais de desenvolvimento de aplicativos para a plataforma da *Google*. Resende (2018) diz que, "isso já era esperado, pois *Kotlin* vinha recebendo muitos *feedbacks* positivos da comunidade de desenvolvedores".

[2] De acordo com Juneau (2009-2019), *Project Lombok is a mature library that reduces boilerplate code. The library replaces boilerplate code with easy-to-use annotations[...] making code easier to read and less error-prone and making developers more productive.*

A linguagem *Kotlin* possui uma sintaxe simples, pouco verbosa e concisa. Segundo Lecheta (2018), "uma das grandes vantagens do *Kotlin* é a sintaxe moderna e expressiva, pois, se comparado ao *Java*, pode-se escrever o mesmo código com muito menos linhas".

O propósito deste trabalho é elucidar de forma comparativa e concisa as vantagens de se optar pela linguagem de programação *Kotlin*, em meio às múltiplas alternativas. Aproveitando os recursos atuais e expressivos da linguagem para desenvolver um código menos clichê, otimizando assim o processo de manutenção e de teste.

1.2 Objetivos

Nesta secção serão apresentados os objetivos gerais e específicos desta pesquisa.

1.2.1 Objetivo Geral

O objetivo dessa pesquisa é apresentar uma análise quali-quantitativa sobre a utilização da linguagem de programação *Kotlin*, referente às questões de usabilidade em desenvolvimento de *software* com foco em aplicações móveis.

1.2.2 Objetivos Específicos

Os passos abaixo descrevem as etapas necessárias para a conclusão do objetivo geral da pesquisa:

- Estudar a linguagem de programação *Kotlin* e questões relacionadas com a usabilidade na programação;
- Definir métricas de usabilidade na programação;
- Comparar o uso do *Kotlin* em relação a linguagem de programação *Java*;
- Analisar quanti-qualitativamente o uso do *Kotlin*, evidenciando as métricas de usabilidade estabelecidas.

1.3 Organização deste Trabalho

O presente trabalho está estruturado em cinco capítulos. No primeiro capítulo é apresentada uma breve introdução, objetivos buscados e justificativa da pesquisa. No segundo capítulo são expostos os textos referenciais. A metodologia do projeto é demonstrada no terceiro capítulo. No quarto capítulo são exibidos os resultados finais. As conclusões são desenvolvidas no quinto capítulo.

2 REVISÃO DA LITERATURA

Neste capítulo, serão apontados os levantamentos bibliográficos realizados para esta pesquisa. Nele, serão apresentados os seguintes temas: dificuldades existentes na programação, linguagens de programação nativas e híbridas voltadas para desenvolvimento de dispositivos móveis, qualidade de código e trabalhos relacionados a este trabalho.

2.1 Dificuldades Existentes na Programação

Devido a rápida popularização dos aplicativos e sua fácil e prática utilização, os iniciantes podem acabar optando por entrar no ramo de desenvolvimento de *software* com a ideia de conquistar uma vida financeira próspera por meio da disponibilização de uma aplicação simples em qualquer uma das famosas lojas disponíveis, e isso sem maiores esforços. No entanto, o aprendizado em tecnologia da informação, assim como em qualquer outra área/ramo, demanda empenho e prática. Um estudante de ciências contábeis, administração leva um tempo para se adaptar à normas e cálculos atuais, da mesma forma como um estudante da área da saúde não realiza manobras de salvamento sem conhecimento prévio, o mesmo conceito se aplica na programação. Pular os fundamentos da área pode ser uma das piores práticas, pois o conhecimento básico aplica-se a todos os tipos de desenvolvimento, seja ele voltado para *web*, *desktop* ou *mobile* (BRITO, 2018).

O ato de desenvolver uma simples solução ou um *software* extremamente complexo demanda muito mais do que apenas escrever um algoritmo em uma determinada linguagem e tecnologia. O mais comum na área de tecnologia, é trabalhar e dar manutenção em um sistema que já está sendo utilizado, e que foi escrito por outros profissionais, para realizar qualquer modificação, faz-se necessário **compreender** toda a estrutura, conseguir visualizar/imaginar as consequências de cada futura alteração. Na maioria das vezes um código mais enxuto é sempre superior a outro com inúmeros trechos inúteis de códigos desnecessários. Nesse momento fazem-se úteis as melhorias internas como **reescrita** usando técnicas como *TDD*, programação em par, revisão de código e versionamento de projeto (FERREIRA, 2018).

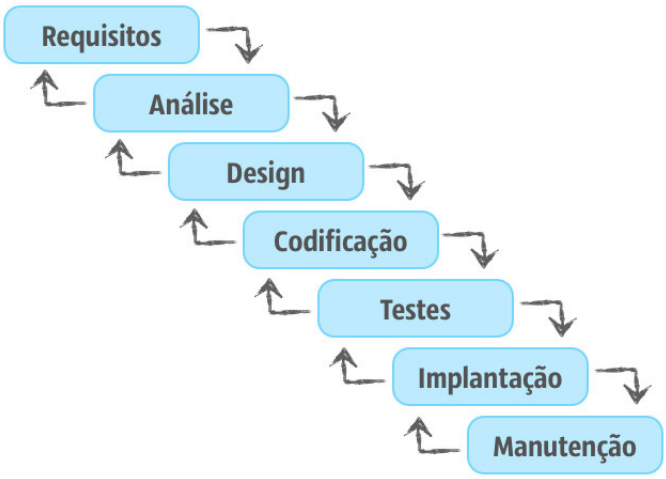
O mercado de tecnologia da informação é um dos poucos onde a demanda por profissionais qualificados de todos os níveis cresce independente de qualquer fator interno ou externo. Porém, a escolha de tecnologias para estudar, aplicar em um projeto ou solucionar problemas é imensa, principalmente para os ingressantes, essa busca pode ser um tanto quanto custosa. O primeiro passo para a escolha da linguagem adequada deve ser a definição do mercado de atuação (SCUDERO, 2017).

Anualmente são divulgados índices ranqueados sobre as linguagens, mesmo assim as opções são inúmeras. Para não errar na escolha, deve-se optar entre as tecnologias de *front-end* (que estão em pleno crescimento) como *HTML*, *CSS*, *JavaScript* e *frameworks JavaScript* ou linguagens *back-end* (maior empregabilidade), dentre as principais estão *Java*, *C#*, *Python* e *Swift*. A iniciação no mundo de desenvolvimento de *software* com *JavaScript*, que aplica-se tanto em aplicações *front-end* como no *back-end* tende a ser mais indicado para os iniciantes, pois necessita-se apenas de um navegador de internet que todos já têm/tiveram contato, e um editor de texto, o que é quase nada se comparado a instalação e configuração de um ambiente de desenvolvimento integrado e de um conjunto de desenvolvimento de *software* (SCUDERO, 2017). Essa facilidade no início levou o *JavaScript* a ser introduzido em renomadas instituições de ensino estadunidenses, substituindo assim, o lugar do *Java* como linguagem introdutória para muitos alunos e turmas. Tanto *C#* como o *Java* possuem uma curva de aprendizado maior, o que acaba tornando-se pouco atrativo para novatos, mas que em contrapartida possuem comunidades fortes e grandes companhias dando suporte, o que eleva o índice de oferta, procura e contratação (SCUDERO, 2017).

2.1.1 Metodologias Ágeis

Em sua maior parte, todas as técnicas ágeis empregam e recomendam o desenvolvimento dividido/subdividido em pequenas etapas, e estas etapas tendem a ser pouco extensas, o que por sua vez, assegura o parecer contínuo e resultados mais velozes mediante possíveis alterações, tendo em vista substituir técnicas mais antigas e ineficazes, porém ainda utilizadas como a metodologia de desenvolvimento em cascata, conforme Figura 1. (GOMES, 2014).

Figura 1 – Metodologia *Waterfall* (Cascata)



Fonte: (GOMES, 2014)

Dentre as inúmeras técnicas denominadas ágeis presentes e disponíveis no mercado, pode-se destacar:

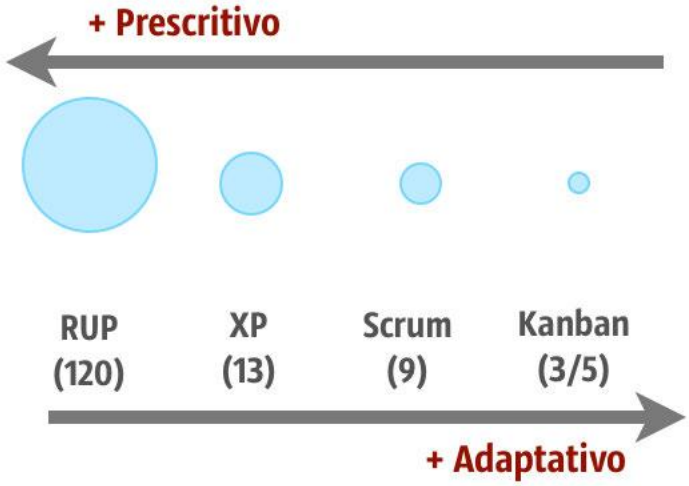
- ***Scrum***: Desenvolvido em meados de 1990 por Ken Schwaber, Jeff Shuterlande Mike Beedle, o *framework Scrum* está entre os mais aplicadas no mercado, com ênfase na administração de projetos, a ferramenta fomenta a acabativa[1] no desenvolvimento de *software* e na maioria dos casos é implementado juntamente com outras ferramentas ágeis (WILDT et al., 2015).

- ***eXtreme Programming***: Criado por Kent Beck, Ward Cunningham e Ron Jeffries, comumente chamado de *XP*, a técnica *eXtreme Programming* foi idealizada para equipes intermediárias, o *XP* foca principalmente na programação e nos testes. Seu propósito é a elevação do nível de confiabilidade, manutenabilidade e reusabilidade dos algoritmos desenvolvidos (WILDT et al., 2015).

- ***Kanban***: Na língua japonesa *Kanban* quer dizer cartão sinalizador. Em contrapartida das outras técnicas ágeis, o *Kanban* não impõe etapas, a ferramenta visa o aumento de produtividade por meio de uma melhor apresentação do *workflow*, separando as atividades em etapas. É a metodologia ágil que menos impõe regras (GOMES, 2014).

Em Scrum, XP e Kanban, há uma menor quantidade de prescrições a se seguir, facilitando a adaptação, conforme Figura 2.

Figura 2 – Métodos ágeis e seus níveis de prescrição e adaptação



Fonte: (GOMES, 2014)

[1] Neologismo criado por Stephen Kanitz, que segundo o autor é "a capacidade que algumas pessoas possuem de terminar aquilo que iniciaram ou concluir o que outros começaram (ADMINISTRADORES, 2015-2019).

2.2 Tecnologias e Linguagens de Programação para Dispositivos Móveis

Quando fala-se em aplicação móvel, na verdade referem-se aos programas que realizam operações específicas em *smartphones*, *tablets* e dispositivos *wearables* (JUNIOR, 2018).

2.2.1 Desenvolvimento Nativo para *Android*

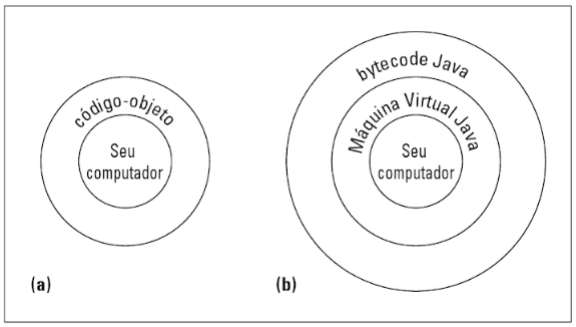
Apesar das aplicações móveis multiplataforma ganharem mais espaço a cada ano, as aplicações mais bem-sucedidos do mercado mundial como o *Facebook Messenger* e *Whatsapp* foram desenvolvidos de forma nativa (JUNIOR, 2018).

2.2.1.1 Linguagem *Java*

A linguagem de programação *Java* já a algum tempo é predominante na *web* (*World Wide Web* - WWW), a linguagem transformou a compreensão sobre desenvolvimento de sistemas, tornando-se indispensável para aqueles que almejam serem especialistas da área. Com fortes heranças de linguagens como *C* e *C++* e incorporando avanços e outras capacidades, *Java* foi criado por James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan em 1991 na até então *Sun Microsystems* atual *Oracle Corporation*. Com o objetivo de se obter uma linguagem portátil, livre de outros programas e que fosse capaz de ser funcional nos aparelhos digitais comuns no dia-a-dia (SCHILDT, 2015).

A linguagem ganhou grande repercussão em 1995 devido a sua mobilidade, possível graças a sua *Java Virtual Machine* (Máquina Virtual *Java* - ***JVM***) que atua como uma tradutora, transformando todo o código ***bytecode*** *Java* em um idioma compreendido pelo computador, independente se o sistema operacional utilizado é um *Windows*, *Linux* ou *macOS*. Com a ausência de uma *JVM*, seria necessário escrever códigos modificados para cada uma das plataformas (BURD, 2014), conforme apresentado na Figura 3.

Figura 3 – Execução de um programa de computador



Fonte: Burd (2014)

2.2.1.2 Plataforma *Android*

Criado pela *Android Inc*, o sistema *Android* foi comprado pela *Google* em 2005, e desde 2007 vem sendo suportado pela *OHA* (*Open Handset Aliance*) grupo formado por gigantes da tecnologia como *HTC*, *Sony*, *Dell*, *Motorola*, *Qualcomm*, *Texas Instruments*, a própria *Google*, *Samsung*, *LG*, *T-mobile*, *Sprint Corporation*, *Nvidia*, *Wind River Systems*, *Asus*, *Intel* e outras (DEITEL; DEITEL; WALD, 2016).

Aplicativos móveis voltados para a plataforma portátil da *Google*, devem ser criados com a linguagem de programação *Java*, escolhida por ser uma das mais utilizadas pelos desenvolvedores, empresas e instituições de ensino e não ter nenhum custo, uma vez que é *open source*[2] Com isso, desenvolvedores com anos, tendem a ter uma curva de aprendizado pequena em relação ao *Android*, se comparado aos programadores que não possuem experiência com a linguagem (DEITEL; DEITEL; WALD, 2016).

Aparelhos com o sistema operacional *Android* possuem funcionalidades como acesso à internet, música, vídeo, Sistema de Posicionamento Global (*Global Positioning System - GPS*), câmera, tela sensível ao toque, dentre outras inúmeras funções. A *Play Store* é a loja oficial para adicionar, avaliar, procurar e realizar o *download* das aplicações disponíveis (DEITEL; DEITEL, 2016).

O sistema operacional da *Google* tem previsão de continuar como o mais vendido até 2022 (IDC, 2018). Conforme Tabela 1 e Figura 4.

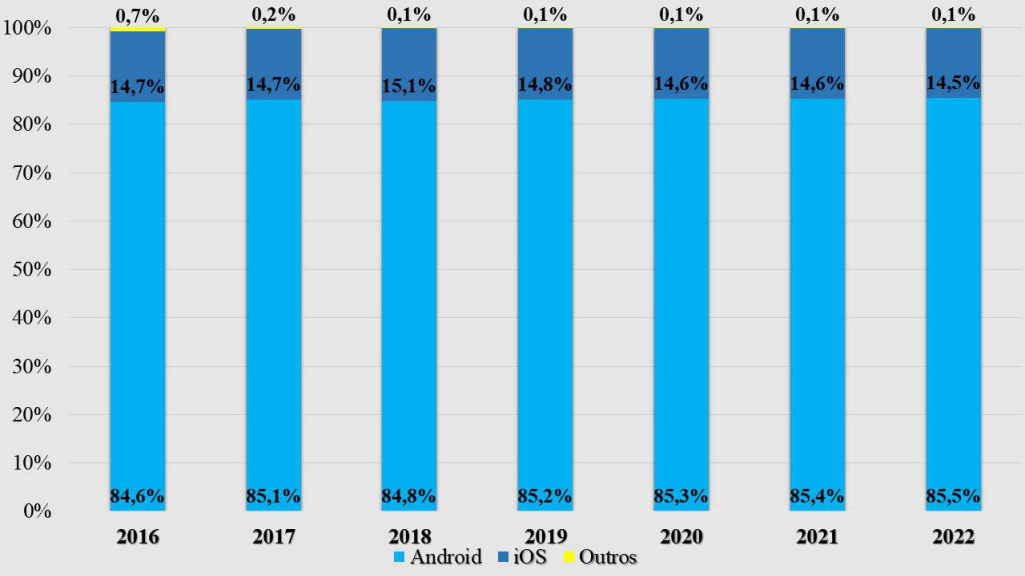
Tabela 1 – Participação no mercado mundial de *smartphones*.

Ano	**2016**	**2017**	**2018**	**2019**	**2020**	**2021**	**2022**
Android	84,6%	85,1%	84,7%	85,2%	85,3%	85,4%	85,5%
iOS	14,7%	14,7%	15,1%	14,8%	14,6%	14,6%	14,5%
Outros	0,7%	0,2%	0,1%	0,1%	0,1%	0,1%	0,1%

Fonte: Adaptado pelo autor - (IDC, 2018)

[2] A expressão *open source* de origem americana, remete a *softwares* que possuem o seu código fonte aberto para contribuições. (CANALTECH, 2017)

Figura 4 – Gráfico de participação no mercado mundial de *smartphones*.



Fonte: Adaptado pelo autor - (IDC, 2018)

2.2.2 Desenvolvimento Nativo para *iOS*

Tanto os usuários como os desenvolvedores que buscam por um *design* mais sofisticado, uma riqueza maior em detalhes, aliados a uma comunicação diferenciada com o aparelho móvel, encontram como opção o *iPhone*, que possui *iOS*, o sistema operacional móvel da empresa americana *Apple Inc.*, o mais utilizado após o *Android* (LECHETA, 2016).

Anunciado em 2007 em uma conferência voltada para programadores, no mesmo ano o *smartphone iPhone* conseguiu ganhar seu lugar de destaque entre os usuários comuns e do mundo da tecnologia, devido a capacidade e qualidade dos recursos presentes no pequeno aparelho móvel, obteve também a atenção dos programadores, uma vez que sua loja oficial, a *Apple Store* prometia uma gratificação, o que até o momento era novidade (LECHETA, 2016).

Apesar de lançado em 2007, somente no ano seguinte tornou-se possível programar voltado para o *iOS*, por meio do Kit de Desenvolvimento de *Software* (*Software Development Kit - SDK*), porém o *SDK* passou a ser gratuito apenas três anos mais tarde. Para utilizar-se o *iOS SDK*, é obrigatório possuir um computador com no mínimo o sistema operacional *macOS X* instalado, caso contrário, não será possível nem mesmo realizar o *download*. Recentemente, o *SDK* passou a ser disponibilizado com o ambiente integrado oficial de desenvolvimento para a plataforma *iOS*, o *Xcode IDE* (STEIL, 2015).

2.2.2.1 Linguagem *Objective-C*

Adquirida e licenciada em 1988 pela empresa *Apple Inc.*, a linguagem *Objective-C* foi fundamental no desenvolvimento de aplicações voltadas para *iOS* e *macOS*. Pode-se dizer que apesar da diferença na escrita de algoritmos, *Objective-C* é na verdade *C* com o acréscimo de várias funcionalidades como o paradigma de programação, portanto *Objective-C* é totalmente interoperável com tudo que já existe e que foi escrito originalmente utilizando-se a linguagem *C* (STEIL, 2015).

Dado ao avanço das linguagens de programação e das tecnologias envolvidas, o *Objective-C* acabou tornando-se um tanto quanto obsoleto quando comparado a outras linguagens mais sucintas e com curva de aprendizado consideravelmente inferiores. Para mudar esse cenário, e fazer com o que a programação voltada para seus produtos voltasse a ser atrativa, a *Apple* lançou a linguagem *Swift* (MACHADO, 2015).

2.2.2.2 Linguagem *Swift*

Com uma grande influência de linguagens mais modernas como *Scala* e *C#*, *Swift* foi desenvolvida para ser fácil e compatível com mais de um paradigma de desenvolvimento (MACHADO, 2015). É voltada para os principais produtos da *Apple*, como *iOS*, *macOS*, *Apple Tv* e *Apple Watch*, possui total compatibilidade com algoritmos legados escritos em *Objective-C*. Aplicações feitas em *Swift* são menos vulneráveis e poupam consideravelmente tempo em seu desenvolvimento (APPLE, 2019).

Grande parte, senão a maioria dos programadores com foco em produtos *Apple* migraram de *Objective-C* para *Swift* e já constroem aplicações usando somente a nova linguagem no *backend*. Aplicações *Swift* chegam a ser **2,6** vezes mais velozes do que aplicações em *Objective-C* e **8,4** vezes mais ágeis em comparação com *Python* na sua versão 2.7. Por ser uma linguagem *open source*, ganhou espaço tanto no meio educacional como profissional (APPLE, 2019).

2.2.3 Desenvolvimento Multiplataforma

Com todo o avanço das tecnologias envolvidas nos dispositivos móveis, a *Google* tinha seu próprio sistema, assim como *Apple*, *Microsoft*, *BlackBerry* e outros que com o passar dos anos acabaram sendo descontinuados, ou ficaram com uma fatia quase inexistente no mercado. Porém cada empresa seguiu por caminhos diferentes como: *frameworks*, linguagens e máquinas virtuais, o que acabou culminando na incompatibilidade das plataformas, fazendo-se necessário a criação de aplicações específicas para cada público (GOIS, 2017).

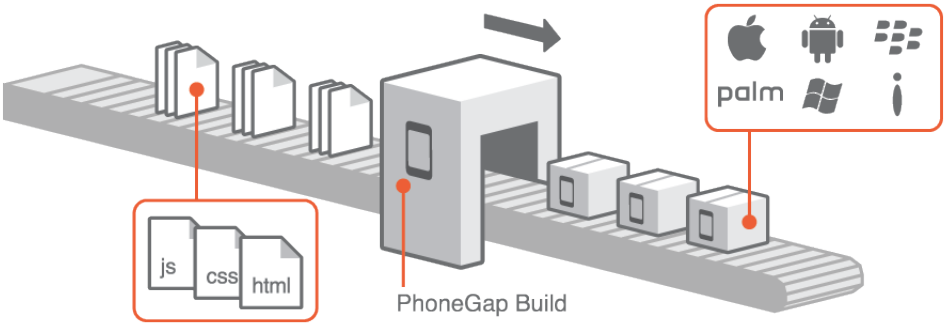
2.2.3.1 *Xamarin Framework*

Utilizando-se todo o potencial e funcionalidades da linguagem *C#* da empresa americana Microsoft, tornou-se possível construir aplicações móveis tanto para a plataforma *iOS*, *Android* e para a recentemente descontinuada *Windows Phone* e até mesmo para o sistema operacional *Windows*. Anteriormente fazia-se necessário pagar pela utilização da ferramenta *Xamarin*, o que acabou mudando após sua venda para a *Microsoft*. Atualmente, é uma ferramenta sem custo e de código *open source* (ARAUJO, 2017). Pode-se aproveitar entre 75% a 100% de qualquer algoritmo já existente para utilização em desenvolvimento para qualquer um dos ambientes suportados, fazendo-se necessárias pouquíssimas alterações para chegar ao objetivo desejado de programação multiplataforma, alcançando-se assim o total aproveitamento do sistema utilizado, da linguagem *C#* e da plataforma *.NET*, já na parte de Interface e Experiência do Usuário (*User Experience* e *User Interface* - *UX/UI*) utiliza-se o *Xamarin Forms*, onde aplica-se *C#* e/ou *XAMAL*, aproveita-se totalmente uma tela em qualquer ambiente suportado pelo *Xamarin* sem nenhuma complicação ou ajuste manual a ser realizado pelo desenvolvedor (ARAUJO, 2017).

2.2.3.2 *IONIC Framework*

O *framework PhoneGap/Cordova* trouxe com ele esperança para empresas e desenvolvedores do ramo da programação conseguirem evitar o **retrabalho** de criação de aplicações móveis específicas para cada fatia do mercado, por meio do aproveitamento dos já consolidados *HTML*, *CSS* e *JavaScript* para criação de uma categoria intermediária. O *IONIC Framework* criado em 2013, apoia-se em ferramentas focadas no *front-end* como *PhoneGap/Cordova* juntamente com o *AngularJS* e outros recursos para criação de aplicações multiplataforma (GOIS, 2017), conforme apresentado na Figura 5.

Figura 5 – Fluxo de conversão do *PhoneGap*



Fonte: Gois (2017)

A programação com base em *PhoneGap/Cordova* por meio de tecnologias de *front-*

end, torna possível a concepção (***build***) de aplicações móveis nativas para as plataformas **alvo**, esses aplicativos desempenham características semelhantes aos programas nativos, pois o *software* está sendo compilado incorporado em uma espécie de navegador de internet, denominado de *webview*, porém, essa abstração com o navegador *webview* é invisível e imperceptível para os seus utilizadores. Esse navegador adaptado beneficia-se primordialmente da linguagem de marcação *HTML*, de uma folha de formatação de estilo *CSS* e do *JavaScript* para sua exibição e funcionamento (VASCONCELLOS, 2017). Para criar-se uma aplicação compatível com mais de um sistema operacional com o *IONIC Framework*, faz-se necessária a instalação do *IONIC Framework CLI*, *JDK* versão 7, *SDK do Android*, *NodeJS* e um editor de texto ou *IDE* compatível (GOIS, 2017).

2.2.4 Linguagem *Kotlin*

Em 2017, foi divulgado que seria possível desenvolver aplicações móveis nativas voltadas para o sistema operacional *Android* utilizando não somente a linguagem *Java*, mas também desenvolver com *Kotlin*, todo o público presente no evento *Google I/O* apoiou e aprovou o pronunciamento. Com tal possibilidade de se escolher entre as duas linguagens e até mesmo mesclá-las nos mesmos projetos, classes e afins, o desenvolvimento *Android* tornou-se muito mais proveitoso (LECHETA, 2018).

A motivação para criação do Kotlin foi basicamente suprir as necessidades encontradas pela equipe da JetBrains durante os anos de desenvolvimento de ambientes integrados como *IntelliJ*, *PyCharm*, *RubyMine* e *WebStorm* usando *Java* como linguagem principal e também *Scala*. A linguagem foi batizada com o nome da ilha Russa de *Kotlin* localizada próxima a São Petersburgo (LECHETA, 2018).

Tanto *Kotlin* como o *Android Studio* que é a *IDE* oficial de desenvolvimento *Android*, foram criados pela empresa russa *JetBrains*. Sua sintaxe de fácil compreensão proporciona uma programação mais prazerosa. Possui total compatibilidade com a linguagem *Java* pois ambas utilizam a *JVM* (LECHETA, 2018).

A linguagem começou a ser desenvolvida em 2011, com base no vasto conhecimento da empresa *JetBrains* no desenvolvimento e suporte de *IDE's* tanto para *Java* como para vários outros tipos de tecnologias, com a intenção de chegar-se a uma linguagem que poderia ser executada e/ou usada nos mesmos projetos onde *Java* era utilizado e não somente para o ecossistema *Android*[4]. Em meados de 2006, já havia a possibilidade de se realizar o *download* do complemento e instalar na *IDE Android Studio* para programar em *Kotlin* para *Android* (RESENDE, 2018).

[4] O ecossistema Android é composto pelas linguagens suportadas pela *JVM* referente a programação Android nativa, ambiente (*IDE*) e kit de desenvolvimento e a loja oficial na qual os aplicativos são disponibilizados (INFOQ, 2011)

A seguir os pontos que tornam a linguagem tão relevante no mercado de desenvolvimento (RESENDE, 2018):

- **Sucinta:** A característica de ser uma linguagem com sintaxe bem sucinta em relação ao *Java*, significa que ao converter um arquivo/classe *Java* para *Kotlin* o resultado final será o mesmo, porém em uma abordagem com *Kotlin* teria-se menos códigos desnecessários, como uma classe sem métodos acessores e modificadores explícitos por exemplo, o que facilita em muito a leitura e manutenção de código, essa conversão pode ser realizada tanto de forma manual ou com recursos da própria *IDE Android Studio*.

- **Protegida contra exceções:** Uma das exceções mais encontradas na programação *Java*, com certeza é *NullPointerException* que se trata de uma menção a algo que ainda não está disponível a nível de programa. Em *Kotlin* **não** é possível atribuir o valor nulo a uma variável, para forçar uma variável a receber um valor nulo é necessário adicionar um **ponto de interrogação** na declaração do tipo.

- **Desejada:** 2º linguagem de desenvolvimento mais querida, a 4º mais pesquisada e a 22º mais admirada do ano de 2018, segundo pesquisa realizada com mais de 100 mil usuários de todas as partes do mundo.

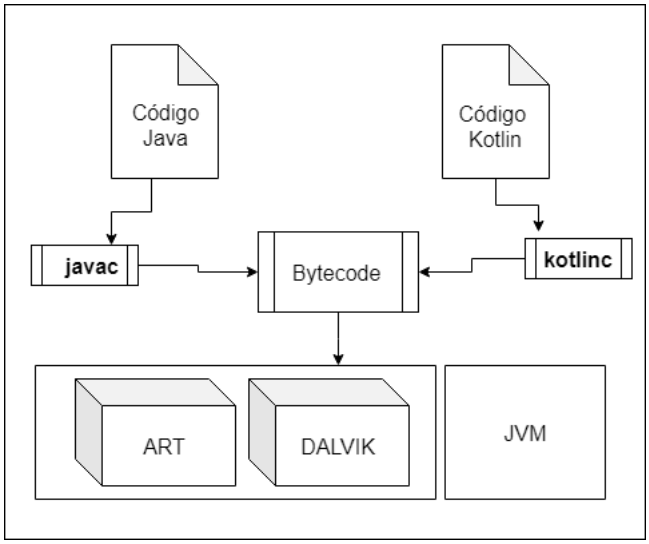
- **Integrada:** Desde a versão 3.0 da *IDE Android Studio*, fez-se desnecessária a instalação de *plugins Kotlin*, pois o complemento passou a ser disponibilizado totalmente integrado com o ambiente de desenvolvimento.

- **Compatível**[3]**:** A total compatibilidade entre a linguagem da empresa *JetBrains* e *Java*, permite a troca de dados entre arquivos tanto na extensão .kt como .java.

Os arquivos que possuem extensão .kt foram desenhados e idealizados desde sua concepção para possibilitarem sem maiores dilemas e complicações inúteis as chamadas aos arquivos de extensão .java de forma totalmente nativa e eficaz, conforme apresentado na Figura 6.

[3] ***ART*** *Android Runtime*: Ambiente de tempo de execução a partir do *Android* 5 e superior. ***DALVIK*** *Android Runtime*: Ambiente de tempo de execução em versões inferiores ao *Android* 5. (GOOGLE, 2017-2019)

Figura 6 – Fluxo de conversão de códigos Kotlin e Java para *bytecode*



Fonte: Adaptado pelo autor
- (KOENIG, 2016)

2.3 Qualidade de Código

Qualquer profissional da área de tecnologia e programação tem plena de consciência da qualidade dos seus códigos, porém, em projetos consideravelmente maiores, e com muitos responsáveis, pode ocorrer a perda da qualidade no projeto. Para sanar essas adversidades, existem diversos programas, que realizam de forma automática ou não, o controle e a análise de qualidade, produzindo relatórios técnicos, entretanto, nem sempre se chega ao resultado esperado (BATISTA, 2016).

2.3.1 Refatoração de Código

A técnica de refatoração propõe examinar os códigos já existentes, com o objetivo de se reduzir repetições (desnecessárias) e incoerências presentes no sistema (LINDSTROM, 2017). Com a falta da prática de se refatorar, os projetos acabam tornando-se muito complexos, inviabilizando futuras alterações e correções, chegando ao ponto de se cogitar a total reescrita de um sistema, independente do seu tamanho e importância. Em seu livro Código Limpo, o renomado autor Robert C. Martin definiu alguns parâmetros para uma refatoração eficiente (ZANETTE, 2017):

- **Nomenclaturas**: Deve-se declarar operações, propriedades, parâmetros e classes de forma condizente com sua função, para que ao realizar a leitura, seja fácil identificar o seu verdadeiro propósito. As nomenclaturas devem ser as mais fiéis possíveis, mesmo que para se alcançar o objetivo, seja necessária uma nomenclatura relativamente grande. Operações necessitam obrigatoriamente serem declaradas com nomenclatura

de verbos, e classes e suas instâncias necessitam estar declaradas como substantivos (ZANETTE, 2017).

- **Operações**: Robert C. Martin, diz que as operações devem ser curtas e descomplicadas, o quanto menor for uma operação, melhor. Uma operação só pode executar exclusivamente uma atividade, isso vai oportunizar o reaproveitamento da operação, dessa forma será mais fácil a revisão e conservação do código (ZANETTE, 2017).

- **Comentários/Anotações**: Deve-se comentar o mínimo possível, pois apesar dos sistemas estarem sempre em modificações, os comentários acabam sendo esquecidos e pode vir a prejudicar futuramente. Portanto ao utilizá-lo, deve ser sempre revisado (ZANETTE, 2017).

- **Duplicidades**: Cada parte de compreensão do código, necessita ser exclusiva, pois é prejudicial trechos semelhantes de código que executam a mesma operação em lugares diferentes, pois torna-se muito mais difícil e passível de esquecimento realizar uma mesma modificação em múltiplos locais dentro de um programa. Mesmo que nem toda duplicidade afete razoavelmente, aconselha-se livrar-se delas (ZANETTE, 2017).

- **Prevenção**: Deve-se aproveitar ao máximo os recursos de tratamentos e exceções e erros, oferecidos pela maioria das linguagens de programação disponíveis, tomando *Java* como exemplo, temos disponíveis *Exceptions* e blocos *try-catch*. Um sistema necessita estar pronto para as situações adversas das idealizadas por seu criador. Orienta-se também, evitar os valores nulos em qualquer parte, sempre que possível (ZANETTE, 2017).

- **Reescrita**: Deve-se criar a rotina de reescrita de código, com isso o sistema estará sempre atual, o ideal é reescrever enquanto a raciocínio utilizado ainda está recente na mente do desenvolvedor, mas deve-se tentar evitar sempre a criação de possíveis novos *bugs* (ZANETTE, 2017).

- **Verificação**: Os testes de *software* necessitam ser eficazes, ágeis, individuais, sem duplicidades e efetuar sempre a operação que para a qual foi-se desenhado e programado para realizar. Orienta-se a aplicação do *TDD*, ou seja, primeiro criando a operação de testes, em seguida o trecho de código a ser verificado e após a conclusão, deve-se refatorar (ZANETTE, 2017).

2.3.2 Ferramentas para Análise de Qualidade de Código

Entende-se por Analisadores Estáticos de Códigos (AEC), os programas capazes de identificar, enumerar e até mesmo solucionar imperfeições e falhas existentes em um *software*, o que possibilita que desenvolvedores e gerentes de projeto preocupem-se com

outros fatores de um sistema, uma vez que se tem uma ferramenta específica para realização de uma análise mais detalhada (AZEVEDO, 2018).

Com a utilização de um AEC, as falhas e possíveis ameaças são prontamente detectadas, e isso é realizado dentre outras funcionalidades pela análise de:

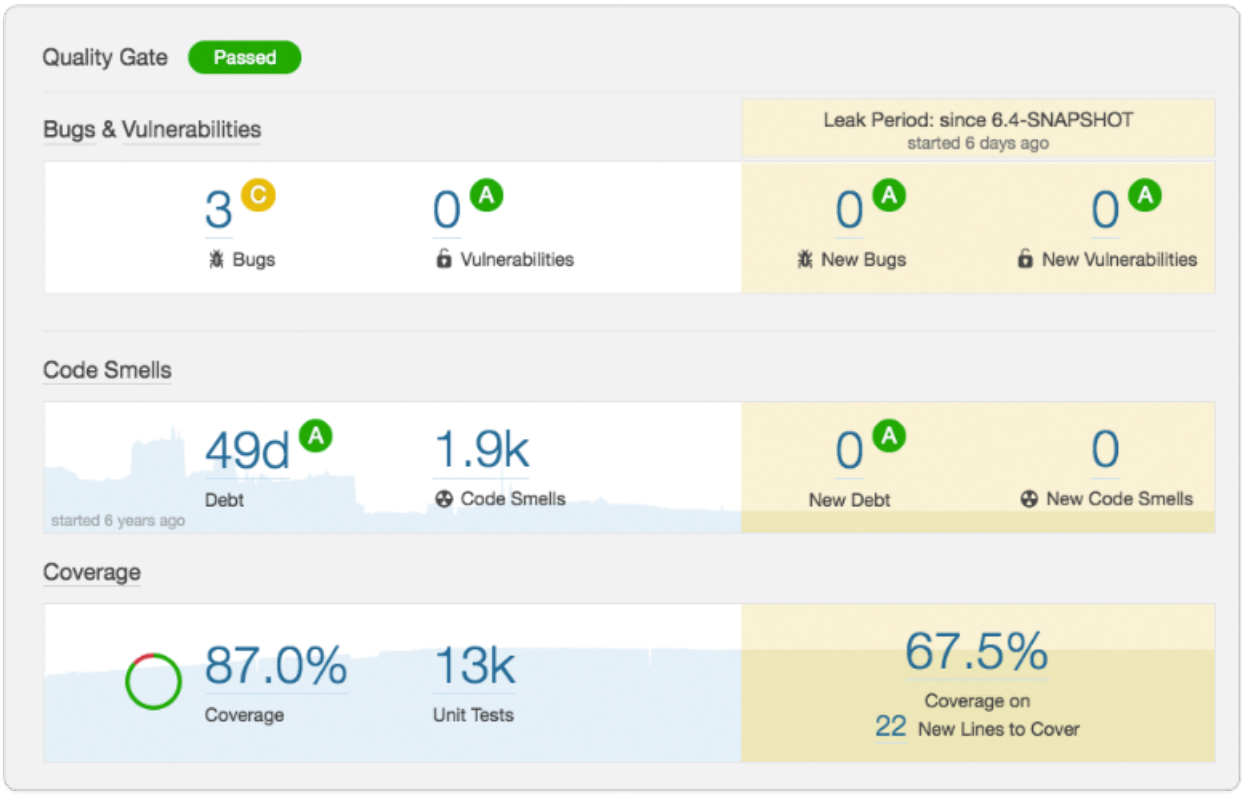
- **Quantidade de linhas de código** (*Lines of code - LOC*) utilizadas, o que pode variar de acordo com as linguagens de programação (mais ou menos verbosas) empregadas no projeto (KEIRóZ, 2015).
- **Quantidade de instruções condicionais e de repetição** (*Cyclomatic Complexity - CC*) aconselha-se que esse número seja sempre o mais baixo possível segundo Robert C. Martin, autor do livro *Clean Code*. (KEIRóZ, 2015).
- **Coerência das operações** (*Lack of Cohesion of Methods - LCOM*) no qual são quantificadas as responsabilidades exercidas pelo conteúdo de uma classe. (KEIRóZ, 2015).

2.3.2.1 *SonarQube*

Anteriormente chamado apenas de *Sonar*, o *SonarQube* desenvolvido pela *SonarSource SA*, é um sistema *web* de código fonte aberto que realiza a verificação das características de um programa e exibe os resultados obtidos por meio de um painel (AZEVEDO, 2018), conforme Figura 7. Dentre suas muitas possibilidades de análise, destacam-se as verificações de padrões de projeto, duplicidade de código e prevenção contra as possíveis inconsistências (KEIRóZ, 2015).

Além de ser personalizável ao gosto do usuário, o programa oferece ainda a possibilidade de ser utilizado juntamente com diversos ambientes de desenvolvimento integrado disponíveis no mercado e também com sistemas de versionamento de projeto como o *GitHub* da *Microsoft*, possibilitando uma oportunidade para efetuar-se correções e ajustes de problemas antecedendo futuras alterações e problemas relacionados ao versionamento (KEIRóZ, 2015).

Figura 7 – Parte do Painel do *SonarQube*

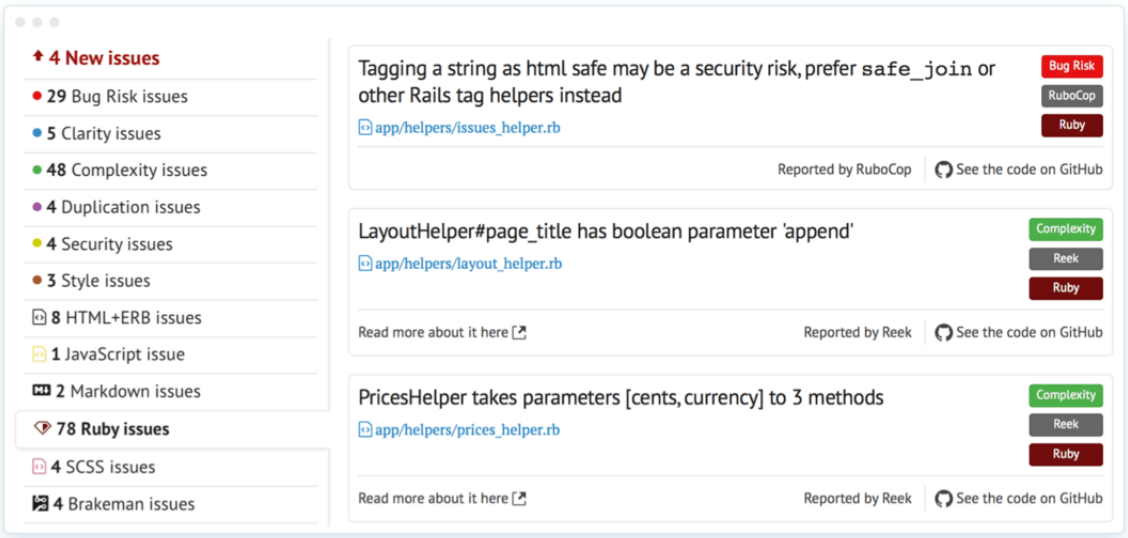


Fonte: (SONARQUBE, 2008-2019)

2.3.2.2 *Ebert*

O *Ebert* é um sistema pago e desenvolvido pela *Caliper Metrics* parte da empresa brasileira Plataformatec que provê uma análise tanto estática como contínua dos códigos desempenhados, por meio de uma total integração com o sistema versionamento de projetos *GitHub*, com a sua utilização a equipe torna-se muito mais apta para visualizar os erros, recebendo sugestões para solução e também dicas para desviar-se deles pelo decorrer de todo o projeto. O sistema abstrai dos envolvidos a necessidade de uma verificação de alterações de códigos submetidos para entrar em produção, notificando instantaneamente sobre brechas de segurança, duplicidades, incoerências desnecessárias, incompatibilidades e também permite vários ajustes e modificações referentes às verificações de código apresentadas, conforme apresentado na Figura 8 (EBERT, 2017-2019).

Figura 8 – Interface do *Ebert*



Fonte: (EBERT, 2017-2019)

2.4 Trabalhos Relacionados a esta Pesquisa

Nesta subsecção, serão apresentados alguns trabalhos relacionados ao tema principal da pesquisa: a linguagem de programação *Kotlin* e a qualidade de *software* por meio de técnicas e tecnologias.

2.4.1 Desenvolvimento de Aplicativo para Adoção de Animais Abandonados Utilizando a Linguagem de Programação *Kotlin* e Programação Reativa (FILHO, 2017)

O intuito do trabalho realizado na Universidade Tecnológica Federal do Paraná nos Departamentos de Acadêmicos de Eletrônica e Informática no curso Engenharia da Computação é de se minimizar o número de animais desabrigados nas grandes cidades como Curitiba, com a criação de uma aplicação móvel nativa, destinada para usuários do sistema operacional *Android*, utilizando os recursos oferecidos pela linguagem de programação *Kotlin* para obter-se mais uma opção para auxiliar as ferramentas já existentes destinadas a facilitar o encontro e a adoção dos animais de rua (FILHO, 2017).

Apesar das dificuldades em utilizar algumas tecnologias de armazenamento de dados mais atuais como o *Firebase*, as metas desejadas foram alcançadas com a criação de uma aplicação móvel em uma linguagem de desenvolvimento ainda pouco conhecido para os meios acadêmicos mas que ofereceu inúmeros aspectos que facilitaram em muito a programação, o projeto encabeçou aumentar a integração das *ONG's* com os interessados em adoção. Como ideia para próximos passos ficaram a criação de um sistema destinado para usuários do sistema operacional *iOS* e a resolução de alguns *bugs* simples, mas que

facilitaram a usabilidade do sistema. No decorrer do desenvolvimento da pesquisa, *Kotlin* passou a ser a nova linguagem de desenvolvimento oficial para *Android* (FILHO, 2017).

2.4.2 *Framework* para Avaliação da Qualidade de Código: Uma Abordagem Baseada em Valor (FERRAZ, 2016)

A meta da pesquisa realizada na Universidade de Brasília, era a análise e o acompanhamento de algumas metodologias denominadas ágeis, juntamente com técnicas de testes de unidade de *software* para obtenção de sistemas menos propensos a futuro/possíveis erros provenientes da falta de averiguação por parte dos envolvidos (FERRAZ, 2016).

Com o cumprimento da meta geral e das específicas como primeira etapa do trabalho, nessa fase inicial da pesquisa, foram planejados todos os possíveis passos para aplicação e utilização do *framework ágil Scrum* e/ou *XP* e dos testes de unidade, com intuito de resultar em melhores acabativas nos projetos desempenhados pela Controladoria Geral do Distrito Federal e do Laboratório Fábrica de *Software* - Campus UnB Gama, ambos localizados em Brasília. Nas etapas futuras, planeja-se colocar em prática os conceitos idealizados (FERRAZ, 2016).

2.4.3 Proposta de Plano de Garantia da Qualidade de *Software* para o Laboratório de Criação e Aplicação de *Software* (CAUZZI, 2015)

Neste artigo desenvolvido na Universidade de Caxias do Sul - Centro de Computação e Tecnologia da Informação, tinha-se como objetivo para a pesquisa, analisar as atuais técnicas de programação empregadas no Laboratório de Criação e Aplicação de *Software* da instituição, para que através dessa futura análise torna-se possível implementar pelo menos parte das ferramentas e técnicas corretas com o propósito de garantir que todos os programas desenvolvidos obtivessem maior qualidade em suas entregas, tanto para os programadores, quanto para os responsáveis técnicos pelos projetos e principalmente para os utilizadores finais. (CAUZZI, 2015).

Os propósitos da pesquisa foram cumpridos, mediante acompanhamento e implantação de ferramentas abertas para gerir o processo de desenvolvimento por parte dos programadores, evidenciou-se que é praticamente impossível realizar um desenvolvimento sem imperfeições e defeitos. Para trabalhos futuros, indicou-se a elaboração de uma inspeção para maiores reconhecimentos dos ganhos obtidos (CAUZZI, 2015).

2.4.4 Estimando Atributos de Qualidade de *Software* Utilizados e Desejados pelas Empresas Certificadas no *MPS-SW* no Brasil (MENDES, 2018)

Nesta pesquisa desenvolvida na Universidade Federal de Ouro Preto, Departamento de Computação e Sistemas, Colegiado de Engenharia de Computação, o intuito era de se elaborar uma equiparação entre a característica dos programas e sistemas entregues e características almejadas pelas empresas participantes do estudo (MENDES, 2018).

Alcançou-se o objetivo do trabalho, mesmo com da falta de comprometimento das empresas solicitadas em se envolverem e responderem algumas perguntas e questionários simples. Constatou-se que as empresas na sua maioria, contam com metodologias ágeis no seu dia-a-dia de programação e mais instrumentos automáticos que por sua vez, executam a geração e o acompanhamento de métricas referentes às tarefas desempenhadas, principalmente os códigos desenvolvidos. (MENDES, 2018).

3 METODOLOGIA

Para que seja possível cumprir os objetivos desse trabalho de conclusão de curso, no presente capítulo serão apresentados todos os passos a serem desenvolvidos. Todos os trechos de código utilizados nesta seção para comparação entre as linguagens *Kotlin* e *Java* apresentam-se disponíveis para visualização e consulta no sistema de gerenciamento e versionamento de projetos *GitHub* no endereço <https://github.com/wendreof/tcc>.

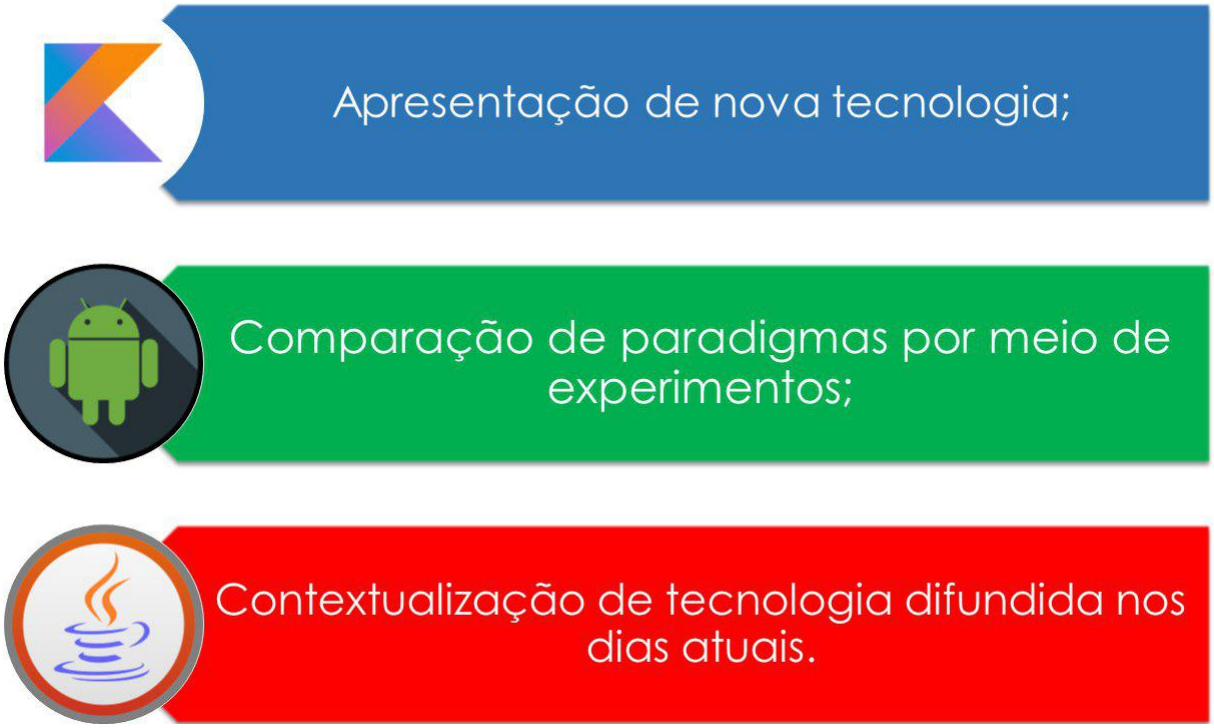
Para a organização dos códigos utiliza-se as formatações, endentações e otimização de importações oferecidas pela própria *IDE Android Studio* e para a estruturação dos projetos utiliza-se o padrão de desenvolvimento com nomenclatura de classes, variáveis, operações e relacionados somente na língua inglesa. Em cada um dos experimentos, explica-se de forma resumida o conteúdo de cada trecho de código, e as particularidades referentes, principalmente ao *Kotlin*, pois o presente trabalho não tem por objetivo, se aprofundar em todos os conceitos relacionados às tecnologias aplicadas.

Todos os comentários utilizados nos trechos de códigos no decorrer do capítulo, apresentam-se sem seus devidos acentos e pontuações, devido a falta de compatibilidade com o Formato de Transformação Unicode 8-bit (*Unicode Transformation Format – 8-bit - UTF-8*).

3.1 Panorama Geral da Pesquisa

O panorama geral desta pesquisa, conforme representado a seguir na Figura 9, propõe-se a esclarecer o motivo da escolha da apresentação de uma nova tecnologia para o desenvolvimento de *software* voltado para os dispositivos móveis por meio de métricas para comparação entre duas linguagens de programação, sendo elas *Java* e *Kotlin*, que por sua vez fazem parte do ecossistema da plataforma móvel *Android* e o desenvolvimento e análise de experimentos baseados em trechos de códigos que desempenham as mesmas funcionalidades.

Figura 9 – Panorama Geral da Pesquisa



Fonte: Elaborado pelo autor

De acordo com o panorama geral da pesquisa, apresentado na Figura 9 acima, a seguir listam-se todas as etapas a serem operacionalizadas, para cumprir-se os objetivos gerais e específicos deste trabalho de conclusão de curso.

- **Etapa 01**: Justificativa para Adoção do *Kotlin* com Destaque nos Aspectos de Usabilidade;
- **Etapa 02**: Métricas de Usabilidade para Comparação;
- **Etapa 03**: Definição de Experimentos para Comparação entre *Kotlin* e *Java*;
- **Etapa 04**: Comparação entre as Linguagens Kotlin e Java para os Experimentos Definidos.

3.2 Etapas para o Desenvolvimento da Pesquisa

3.2.1 Etapa 01: Justificativa para Adoção do *Kotlin* com Destaque nos Aspectos de Usabilidade

Como reforçado nos capítulos anteriores dessa pesquisa, mais especificamente no segundo capítulo, a linguagem de programação Java ainda é o principal e mais utilizada para

desenvolvimento de aplicações móveis destinadas para os usuários do sistema operacional *Android*. Desde seu anúncio oficial, o *Kotlin* chegou para oferecer aos desenvolvedores mais uma opção no momento de escolha de tecnologias para elaboração de novos aplicativos assim como a manutenção de sistemas legados, ou seja, *Kotlin* não vai e nem pretende substituir o *Java*, mas sim **coexistir** de forma homogênea no ecossistema *Android*. Munido de recursos mais modernos, inspirados em tecnologias reconhecidas no mercado de desenvolvimento de software, com *Kotlin* torna-se possível escrever algoritmos compreendidos pela *JVM*, de modo mais simples, facilitando todas as etapas que envolvem a programação e possibilitando o aproveitamento de tudo que já foi escrito em *Java* como classes, projetos e bibliotecas.

Ao término da pesquisa, pretende-se evidenciar as vantagens da programação com foco em dispositivos móveis, utilizando como principal tecnologia, a linguagem de programação *Kotlin*, apoiando-se em seus aspectos modernos e enxutos, resultando em uma menor quantidade de código clichê e desnecessário que em nada auxiliam a aplicação e acabam dificultando a manutenção.

3.2.2 Etapa 02 - Métricas de Usabilidade para Comparação

Baseando-se em um dos conceitos descritos pelo escritor e engenheiro de *software* americano Robert C. Martin, autor do renomado livro *Clean Code: A Handbook of Agile Software Craftsmanship*, no qual se é descrito que na programação lê-se muito mais vezes do que se escreve um código propriamente dito, as etapas estabelecidas para equiparar *Kotlin* e *Java* são apresentadas na Tabela 2:

Tabela 2 – Métricas de comparação

Métrica	**Definição**
1	Quantidade de caracteres utilizados
2	Quantidade de linhas de código
3	Tamanho do arquivo em quilobytes (Kb)

Fonte: Elaborada pelo autor

Em linguagens de programação mais verbosas, faz-se necessária a utilização de uma maior quantidade de caracteres e símbolos, da mesma forma podem-se aumentar a quantidade de palavras chaves e palavras mais longas que o normal para que se torne possível se expressar a real intenção de um algoritmo, o que gera a inclusão de mais linhas no projeto, e consequentemente aumenta o tamanho final do arquivo utilizado em caso de linguagens compiladas como o *Java* e sua *JVM*. Entende-se que uma determinada linguagem que necessita de menos de caracteres, expressões e linhas de código é menos verbosa, mais expressiva e de maior facilidade referente a sua manutenção.

Assim, para os experimentos desta pesquisa, ao comparar as linguagens *Kotlin* e *Java*, serão analisadas as métricas apresentadas na Tabela 2.

3.2.3 Etapa 03 - Definição de Experimentos para Comparação entre *Java* e *Kotlin*

Abaixo os experimentos a serem desenvolvidos no decorrer do capítulo para que seja possível atingir o objetivo de comparação:

Experimento I - *Hello World*!: Apresentação de um pequeno trecho de código que ao ser executado, apresenta uma saída para o usuário;

Experimento II - Operação Aritmética Básica: Elaboração de uma função que tem por intuito realizar a somar de dois números inteiros e interpolar o resultado com um texto;

Experimento III - Classe *Student*: Implementação de uma classe modelo que abstrai algumas das características presentes em um estudante;

Experimento IV - Classe *ThermometricScale*: Apresentação da classe base para o desenvolvimento da aplicação de medidas termométricas e declaração de suas três funções;

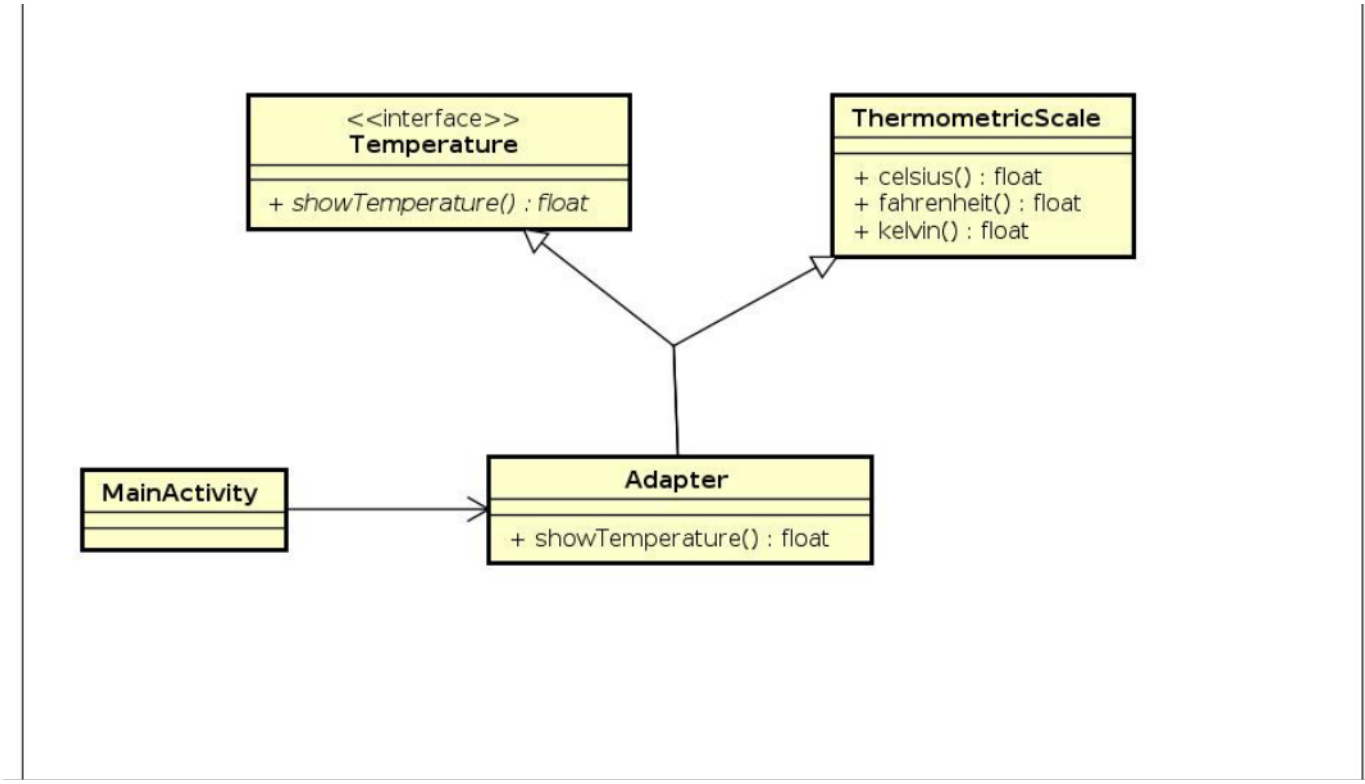
Experimento V - Interface *Temperature*: Implementação de uma interface e um método que recebe dois valores por parâmetros e retorna um único valor;

Experimento VI - Classe *Adapter*: Utilização das técnicas de se extender e implementar uma classe e uma interface;

Experimento VII - *Activity Main*: Criação da principal classe da aplicação de conversão e implementação de todos os eventos de clique e substituição dos textos apresentados na interface.

Nos experimentos I, II, II trabalham-se trechos desconexos de códigos de caráter mais introdutórios das duas linguagens, entretanto, nos experimentos IV, V, VI e VII com o intuito de aprofundar-se mais especificamente na plataforma *Android* e no seu ecossistema, apresentam-se duas classes, uma interface e uma terceira classe do tipo *activity*, ambas referentes a uma aplicação que realiza a conversão de valores termométricos de acordo com o diagrama em Linguagem de Modelagem Unificada (*Unified Modeling Language - UML*) apresentado na Figura 10:

Figura 10 – Diagrama *UML* de Aplicação de Conversão de Medidas Termométricas



Fonte: Elaborado pelo autor

No diagrama *UML* disponibilizado na Figura 10 acima, apresenta-se um exemplo de implementação do padrão de projeto *Adapter* em uma aplicação móvel, na Tabela 3 detalha-se cada uma das classes e suas respectivas funções:

Tabela 3 – Classes e Definições do diagrama UML - Figura 10

Classe	Definição
Adapter	Adaptar a interface *Temperature* e a classe *ThermometricScale*
Temperature	Definir a interface de domínio específica a ser adaptada
MainActivity	Colaborar com objetos da a interface *Temperature*
ThermometricScale	Definir uma interface existente que necessita ser adaptada

Fonte: Elaborada pelo autor

3.2.4 Etapa 04 - Comparação entre as Linguagens *Kotlin* e *Java* para os Experimentos Definidos

3.2.4.1 Experimento I - *Hello World!*

Como início da comparação, apresenta-se nessa subseção, duas versões de um *Hello World*, que nada mais é, do que um trecho de código básico que normalmente é empregado de forma educacional para iniciantes para se exemplificar os princípios de uma determinada

linguagem. A função de um *Hello World* é exibir o texto *Hello World!* (Olá Mundo!) de forma visual por meio de uma saída, seja ela *IDE*, terminal ou tela.

Em versões anteriores do *Kotlin*, fazia-se necessária a implementação de um parâmetro do tipo *Array* de *Strings* na função *main* assim como faz-se necessário em *Java*.

Exemplo de implementação de *Hello World* em *Kotlin*

```kotlin
fun main(){
    println("Hello World!")
}
```

Exemplo de implementação de *Hello World* em *Java*

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

3.2.4.2 Experimento II - Operação Aritmética Básica

Nessa subseção, apresentam-se duas formas de se implementar a função aritmética de soma, declarando-se uma variável inteira que recebe a soma de dois valores inteiros, sendo eles os números 99 e 1 respectivamente, obtendo-se um total de 100. Os trechos de código após execução exibem um texto concatenando com a variável que recebeu o valor da soma.

Em Kotlin disponibilizam-se para utilização dois tipos de variáveis, sendo eles as variáveis mutáveis do tipo ***var*** que podem ter seus valores alterados ao decorrer da execução do programa e as variáveis imutáveis do tipo ***val*** que podem receber apenas uma atribuição de valor, sendo assim variáveis finais.

Exemplo de implementação de função aritmética de soma em *Kotlin*

```kotlin
fun main() {
    val soma = 99 + 1
    println("O resultado da soma e: $soma")
}
```

Exemplo de implementação de função aritmética de soma em *Java*

```java
public class Operations {
    public static void main(String args[]) {
        int soma = 99 + 1;
        System.out.println("O resultado e: " + soma);
    }
}
```

3.2.4.3 Experimento III - Classe *Student*

Nessa subseção foram implementam-se de formas diferentes uma mesma versão de uma classe *Student* na qual implementam-se quatro atributos, sendo eles um atributo name do tipo *string*, outro atributo denominado *tccTitle* também do tipo *string*, *approved* do tipo *boolean* e *yearsOld* do tipo *integer*, e as operações disponíveis são métodos acessores e de configuração *getters* e *setters*, *toString*, *equals* e *hashCode*, todos eles comumente usados em classes do tipo *Model* e na programação *Java*.

Classes do tipo *data class* já possuem por padrão os métodos *getter*, *setter*, *toString*, *equals*, *hashCode* e *copy* implícitos, tornado desnecessária em *Kotlin* a re-implementação das funções. Para obter-se um resultado parecido em *Java* seria necessário configurar o *Project Lombok* e utilizar as anotações *@Getter*, *@Setter*, *@ToString* e *@EqualsAndHashCode* ou a anotação *@Data* no início da implementação da classe ou alguma outra ferramenta com o mesmo propósito.

Exemplo de implementação de uma classe *Student* em *Kotlin*

```
1 data class Student(var name: String,
2                    var tccTitle: String,
3                    var approved: Boolean,
4                    var yearsOld: Int)
```

Exemplo de implementação de uma classe *Student* em *Java*

```
1  import java.util.Objects;
2
3  public class Student {
4
5      private String name;
6      private String tccTitle;
7      private Boolean approved;
8      private int yearsOld;
9
10     public Student(String name, String tccTitle, Boolean approved, int
           yearsOld) {
11         this.name = name;
12         this.tccTitle = tccTitle;
13         this.approved = approved;
14         this.yearsOld = yearsOld;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
```

```
23      }
24
25      public String getTccTitle() {
26          return tccTitle;
27      }
28
29      public void setTccTitle(String tccTitle) {
30          this.tccTitle = tccTitle;
31      }
32
33      public Boolean getApproved() {
34          return approved;
35      }
36
37      public void setApproved(Boolean approved) {
38          this.approved = approved;
39      }
40
41      public int getYearsOld() {
42          return yearsOld;
43      }
44
45      public void setYearsOld(int yearsOld) {
46          this.yearsOld = yearsOld;
47      }
48
49      @Override
50      public boolean equals(Object o) {
51          if (this == o) return true;
52          if (!(o instanceof Student)) return false;
53          Student student = (Student) o;
54          return getYearsOld() == student.getYearsOld() &&
55                  getName().equals(student.getName()) &&
56                  getTccTitle().equals(student.getTccTitle()) &&
57                  getApproved().equals(student.getApproved());
58      }
59
60      @Override
61      public int hashCode() {
62          return Objects.hash(getName(), getTccTitle(), getApproved(),
                getYearsOld());
63      }
64
65      @Override
66      public String toString() {
67          return "Student{" +
68                  "name='" + name + '\'' +
```

```
                ", tccTitle='" + tccTitle + '\'' +
                ", approved=" + approved +
                ", yearsOld=" + yearsOld +
                '}';
    }
}
```

3.2.4.4 Experimento IV - Classe *ThermometricScale*

A classe *ThermometricScale* não possui atributos, apenas três funções, sendo elas *celsius*, *fahrenheit* e *kelvin*, cada uma das operações recebe e retorna um único valor do tipo *float* que é utilizado na fórmula específica de cada uma das escalas termométricas implementadas.

Em *Kotlin* é possível declarar uma função, seu tipo e respectivo retorno na mesma linha de forma mais simplificada e enxuta.

Classe *ThermometricScale* em *Kotlin*

```
package com.example.wlf.ktempconverter.classes.ThermometricScale

open class ThermometricScaleKotlin {
    fun celsius(value: Float): Float = (value - 32) * 5 / 9

    fun fahrenheit(value: Float): Float = (value * 9 / 5) + 32

    fun kelvin(value: Float): Float = value + 273.15f
}
```

Classe *ThermometricScale* em *Java*

```
package com.example.wlf.ktempconverter.classes.ThermometricScale;

public class ThermometricScaleJava {

    protected Float celsius(Float value) {
        return (value - 32) * 5 / 9;
    }

    protected Float fahrenheit(Float value) {
        return (value * 9 / 5) + 32;
    }

    protected Float kelvin(Float value) {
        return value + 273.15f;
    }
}
```

3.2.4.5 Experimento V - Interface *Temperature*

A interface *Temperature* possui apenas um único método denominado de *viewTemperature*, que recebe por parâmetro um valor também do tipo *float* e um único caractere.

Interface *Temperature* em *Kotlin*

```kotlin
package com.example.wlf.ktempconverter.classes.Interface

interface TemperatureKotlin {
    fun viewTemperature(value: Float, s: Char): Float
}
```

Interface *Temperature* em ***Java***

```java
package com.example.wlf.ktempconverter.classes.Interface;

public interface TemperatureJava {
    public Float viewTemperature(Float value, Character s);
}
```

3.2.4.6 Experimento VI - Classe *Adapter*

Na classe *Adapter* implementada em *Java*, torna-se necessária a criação da variável auxiliar r do tipo *Float*, para que seja possível atribuir e retornar ao final da estrutura de decisão condicional *Switch* o valor de forma correta. No entanto, na implementação em *Kotlin*, a própria estrutura *When* responsabiliza-se em retornar por padrão um valor, dispensando a criação de uma variável auxiliar.

A expressão *when* está para a linguagem *Kotlin* assim como *If* e *Switch* estão para o *Java* e todas as demais linguagens derivadas da linguagem de programação *C*.

Adapter em *Kotlin*

```kotlin
package com.example.wlf.ktempconverter.classes.Adapter

import com.example.wlf.ktempconverter.classes.Interface.
    TemperatureKotlin
import com.example.wlf.ktempconverter.classes.ThermometricScale.
    ThermometricScaleKotlin

class AdapterKotlin : TemperatureKotlin, ThermometricScaleKotlin() {

    override fun viewTemperature(value: Float, s: Char): Float {
        return when (s) {
            'c' -> ThermometricScaleKotlin().celsius(value)

            'f' -> ThermometricScaleKotlin().fahrenheit(value)

```

```
14              'k' -> ThermometricScaleKotlin().kelvin(value)
15
16              else -> 0f
17          }
18      }
19  }
```

Adapter em *Java*

```
1  package com.example.wlf.ktempconverter.classes.Adapter;
2
3  import com.example.wlf.ktempconverter.classes.Interface.TemperatureJava;
4  import com.example.wlf.ktempconverter.classes.ThermometricScale.
       ThermometricScaleJava;
5
6  public class AdapterJava extends ThermometricScaleJava implements
       TemperatureJava {
7
8      @Override
9      public Float viewTemperature(Float value, Character s) {
10         Float r;
11         switch (s) {
12             case 'c':
13                 r = celsius(value);
14                 break;
15             case 'f':
16                 r = fahrenheit(value);
17                 break;
18             case 'k':
19                 r = kelvin(value);
20                 break;
21             default:
22                 r = 0f;
23         }
24         return r;
25     }
26 }
```

3.2.4.7 Experimento VII - *Activity Main*

As duas versões de classe *Activity Main* (classe principal) apresentadas a seguir, são de usabilidade simples e possuem como principal responsabilidade dentro do contexto da aplicação de conversão de medidas termométricas, capturar (vincular-se com) todos os *widgets* (objetos/ferramentas) declarados no respectivo arquivo de Linguagem de Marcação Extensível (*Extensible Markup Language - XML*) utilizado na interface, assim como todas as funções de clique em botões como o de limpar tela e converter a temperatura, *Radio*

Buttons para escolha de apenas uma única medida termométrica por vez seja ela *Fahrenheit* ou *Kelvin*, a apresentação/substituição de caixas de textos em campos do tipo *EditText* e *TextView*, exibição de mensagens para auxiliar o usuário na usabilidade da aplicação por meio de *Snackbar* (caixas de textos temporárias na parte inferior do dispositvo) e por fim a validação (verificar se há um texto/valor presente) da caixa de texto do tipo *EditText* na qual o usuário deve inserir um determinado valor em *Celsius*.

Conforme apresentado na Tabela 3, a classe *Activity Main* dada a estrutura do projeto, tem por obrigação colaborar em conformidade com todos todos os objetos da interface *Temperature*.

Activity Main em *Kotlin*

```
1  package com.example.wlf.ktempconverter.views
2
3  import android.os.Bundle
4  import android.support.design.widget.Snackbar
5  import android.support.v7.app.AppCompatActivity
6  import android.view.View.GONE
7  import android.view.View.VISIBLE
8  import android.view.WindowManager
9  import com.example.wlf.ktempconverter.R
10 import com.example.wlf.ktempconverter.classes.Adapter.AdapterKotlin
11
12 /* A importacao abaixo dispensa a declaracao de findViewById's para cada
       um dos widgets
13  declarados no arquivo XML de interface */
14 import kotlinx.android.synthetic.main.activity_main_kotlin.*
15
16 class MainActivityKotlin : AppCompatActivity() {
17     private val tempAdapter = AdapterKotlin()
18
19     /* Declara-se na mesma linha que se trata de uma funcao de
           sobreescrita */
20     override fun onCreate(savedInstanceState: Bundle?) {
21         super.onCreate(savedInstanceState)
22         setContentView(R.layout.activity_main_kotlin)
23
24         /* Pontos e virgulas ao final sao opcionais e desencorajados
               pela propria IDE */
25         window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)
26         window.addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN)
27
28         /* Acesso direto a operacao de click e chamada as funcoes de
               forma mais simples */
29         converterButton.setOnClickListener { convert() }
30         clear.setOnClickListener { clearFields() }
```

```
31      }
32
33      private fun convert() {
34
35          /* Pode-se capturar o atributo texto diretamente sem o metodo
                get */
36          val temp: String = valorTemp.text.toString()
37
38          if (temp.isNotEmpty() && temp != "0") {
39              val temp: Float = valorTemp.text.toString().toFloat()
40
41              when {
42                  fahrenheitRadio.isChecked -> {
43                      converted.text = tempAdapter.viewTemperature(temp, '
                            f').toString()
44                      viewFormula(temp, 'F', tempAdapter.viewTemperature(
                            temp, 'f'))
45                  }
46
47                  KelvinRadio.isChecked -> {
48                      converted.text = tempAdapter.viewTemperature(temp, '
                            k').toString()
49                      viewFormula(temp, 'K', tempAdapter.viewTemperature(
                            temp, 'k'))
50                  }
51              }
52          } else {
53              showMSG(getString(R.string.please_type_a_temp))
54
55              /* Pode-se atribuir um texto sem utilizar o set */
56              valorTemp.error = getString(R.string.type_a_valid_value)
57          }
58      }
59
60      private fun viewFormula(w: Float, e: Char, b: Float): String {
61
62          /* Acesso direto a propriedade visibiliy, sem uso de set */
63          formula.visibility = VISIBLE
64
65          /* A estrutura When ja ira retornar o devido valor */
66          return when (e) {
67
68              'F' -> formula.setText("Formula (${w} C      9/5) + 32 = ${b}
                    $e ").toString()
69
70              /* A interpolacao de variaveis e textos com ${} dispensa a
                    utilizacao de String.format*/
```

```
71              'K' -> formula.setText("Formula ${w} C  + 273.15 = ${b} $e "
                   ).toString()
72
73              else -> "0"
74          }
75      }
76
77      /* Funcoes com logica e retorno simples, podem ser declarados na
           mesma linha */
78      private fun showMSG(msg: String) = Snackbar.make(scrollview, msg,
           Snackbar.LENGTH_LONG).show()
79
80      private fun clearFields() {
81          formula.visibility = GONE
82          formula.text = ""
83          valorTemp.setText("")
84          converted.text = ""
85      }
86  }
```

Activity Main em *Java*

```
1   package com.example.wlf.ktempconverter.views;
2
3   import android.os.Bundle;
4   import android.support.design.widget.Snackbar;
5   import android.support.v7.app.AppCompatActivity;
6   import android.view.View;
7   import android.view.WindowManager;
8
9   /* Necessaria a importa  o dos widgets */
10  import android.widget.Button;
11  import android.widget.EditText;
12  import android.widget.RadioButton;
13  import android.widget.ScrollView;
14  import android.widget.TextView;
15
16  import com.example.wlf.ktempconverter.R;
17  import com.example.wlf.ktempconverter.classes.Adapter.AdapterJava;
18
19  import static android.view.View.GONE;
20
21  public class MainActivityJava extends AppCompatActivity {
22      private AdapterJava tempAdapter = new AdapterJava();
23
24      /* Necessaria a criacao de atributos de mesmo tipo dos widgets
           declarados no arquivo XML de
25       interface para utilizar-se na funcao findViewById */
```

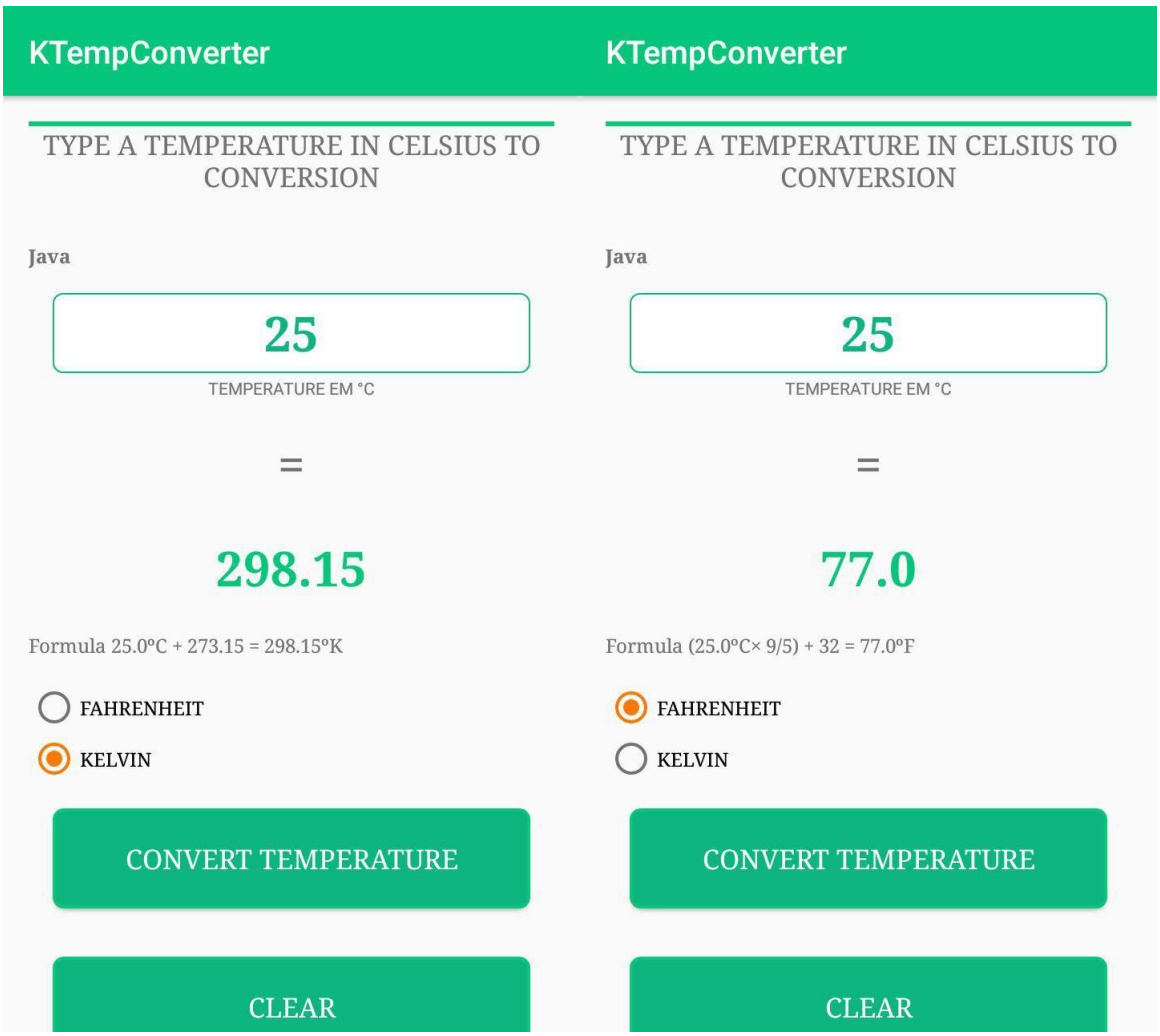
```
26      private EditText valorTemp;
27      private TextView formula;
28      private TextView converted;
29      private Button clear;
30      private Button converterButton;
31      private RadioButton fahrenheitRadio;
32      private RadioButton KelvinRadio;
33      private ScrollView scrollView;
34
35      @Override
36      protected void onCreate(Bundle savedInstanceState) {
37          super.onCreate(savedInstanceState);
38          setContentView(R.layout.activity_main_java);
39
40          /* Pontos e virgulas obrigatorios ao final */
41          getWindow().addFlags(WindowManager.LayoutParams.
                FLAG_KEEP_SCREEN_ON);
42          getWindow().addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN)
                ;
43
44          /* Necessaria a realizacao da captura de cada um dos widgets de
                interface declarados
45           no arquivo XML de interface */
46          valorTemp = findViewById(R.id.valorTemp);
47          formula = findViewById(R.id.formula);
48          converted = findViewById(R.id.converted);
49          clear = findViewById(R.id.clear);
50          converterButton = findViewById(R.id.converterButton);
51          KelvinRadio = findViewById(R.id.KelvinRadio);
52          fahrenheitRadio = findViewById(R.id.fahrenheitRadio);
53      }
54
55      /* Obrigatorio o recebimento de um parametro do tipo View para ser
            utilizado
56       no atributo onClick pelos widgets de interface */
57      public void convert(View v) {
58
59          final String temp = valorTemp.getText().toString();
60
61          if (!temp.isEmpty() && !temp.equals("0")) {
62
63              /* Necessaria a criacao de variavel auxiliar para atribuicao
                    de valor */
64              final Float tempF = Float.parseFloat(valorTemp.getText().
                    toString());
65
66              if (fahrenheitRadio.isChecked()) {
```

```
67                  converted.setText(tempAdapter.viewTemperature(tempF, 'f'
                        ).toString());
68                  viewFormula(tempF, 'F', tempAdapter.viewTemperature(
                        tempF, 'f'));
69
70              } else if (KelvinRadio.isChecked()) {
71                  converted.setText(tempAdapter.viewTemperature(tempF, 'k'
                        ).toString());
72                  viewFormula(tempF, 'K', tempAdapter.viewTemperature(
                        tempF, 'k'));
73              } else {
74                  showMSG(getString(R.string.please_type_a_temp));
75                  valorTemp.setError(getString(R.string.type_a_valid_value
                        ));
76              }
77          }
78      }
79
80      private void viewFormula(Float w, Character e, Float b) {
81          formula.setVisibility(View.VISIBLE);
82
83          if (e == 'F') {
84
85              /* Necessaria a utilizacao de String.format para facilitar
                    interpolacao de variaveis
86              e textos */
87              formula.setText(String.format("Formula (%s  C   9/5) + 32 =
                    %s  %s", w, b, e));
88          } else if (e == 'K') {
89              formula.setText(String.format("Formula %s  C + 273.15 = %s
                    %s", w, b, e));
90          } else {
91              formula.setText("0");
92          }
93      }
94
95      private void showMSG(String msg) {
96          Snackbar.make(scrollView, msg, Snackbar.LENGTH_LONG).show();
97      }
98
99      public void clearFields(View v) {
100         formula.setVisibility(GONE);
101         formula.setText("");
102         valorTemp.setText("");
103         converted.setText("");
104     }
105 }
```

3.2.4.8 Aplicação de Conversão de Medidas Termométricas em *Java* e *Kotlin*

Como resultado dos códigos desempenhados nesta seção, obtém-se as aplicações demonstradas na Figura 11 e Figura 12, referentes ao *Java* e *Kotlin*, respectivamente. A aplicação de conversão de temperatura encontra-se disponível para *download* na *Play Store*, através do *link* <https://bitlybr.com/qAwN7>

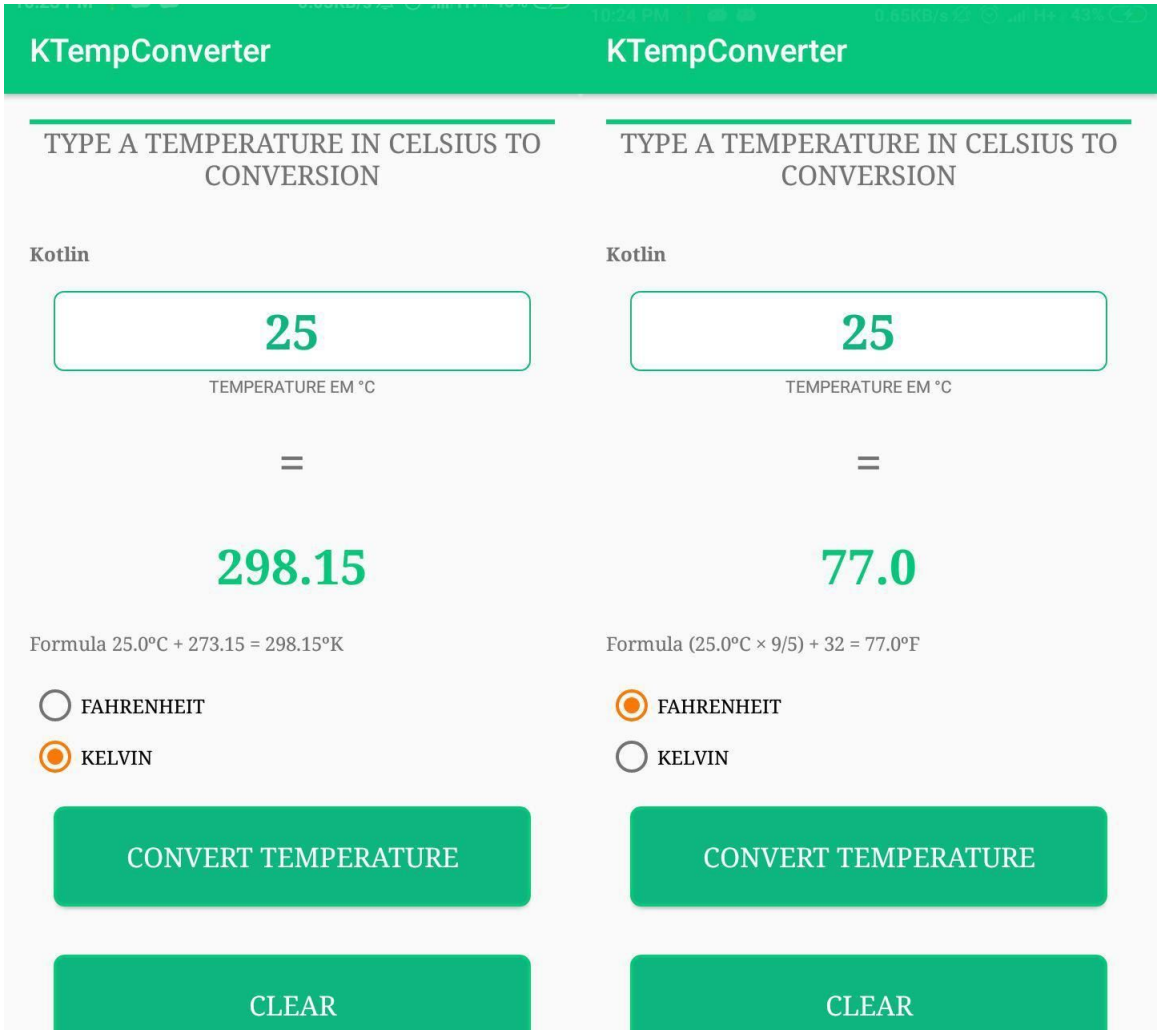
Figura 11 – Aplicação de Conversão de Medidas Termométricas escrito em *Java*



Fonte: Elaborado pelo autor

As duas aplicações desempenham as mesmas funções e possuem os mesmos comportamentos, características e também o mesmo *layout* (cores e disposição).

Figura 12 – Aplicação de Conversão de Medidas Termométricas escrito em *Kotlin*



Fonte: Elaborado pelo autor

Tanto a aplicação *Java* como a aplicação *Kotlin* são capazes de realizar a conversão de uma medida em *Celsius* inserida pelo usuário para *Fahrenheit* ou *Kelvin*.

4 RESULTADOS E DISCUSSÃO

4.1 Análise Qualitativa para Comparação entre *Kotlin* e *Java*

Neste capítulo, apresenta-se análise a respeito do desempenho de cada uma das linguagens analisadas em âmbito geral, levando em consideração cada um dos experimentos em *Kotlin* e *Java*, desenvolvidos na seção anterior.

4.1.1 Comparação das Métricas

Ao decorrer da presente seção, serão apresentados índices baseados no artigo: Análise Comparativa de Linguagens de Programação a partir de Problemas Clássicos da Computação, publico na revista Revista de Sistemas e Computação - RSC, desenvolvido na Universidade Federal de Itajubá do estado de Minas Gerais, no qual se estabelece um estudo comparativo sobre o tamanho dos algoritmos, velocidade de execução e tamanho em *bytes* entre *C*, *Java* e *Python*, ambas linguagens que encontram-se sempre no topo de pesquisas de popularidade. O estudo foi realizado por meio de experimentos matemáticos com diversas fórmulas amplamente conhecidas, os autores concluem com base nas comparações desenvolvidas, que a linguagem *C* possui mais características para ser a escolhida.

Com o intuito de se estabelecer indicadores de qualidade, a fim de se efetuar uma avaliação referente aos algoritmos apresentados nos experimentos I, II, III, IV, V, VI e VII, foram definidos, como base, três índices qualificadores para avaliação, sendo eles:

- Índice **A** (satisfatório): Implementação de algoritmo que realiza a utilização de poucos caracteres especiais, símbolos, assim como baixa quantidade de linhas de código;
- Índice **B** (parcialmente satisfatório): Implementação de algoritmo que se utiliza de mais caracteres especiais, símbolos ou maior quantidade de linhas de código;
- Índice **C** (insatisfatório): Implementação de algoritmo que se utiliza de maior quantidade de caracteres especiais, símbolos e linhas de código.

De acordo com os índices de qualidade exemplificados a seguir na Tabela 4, apresentam-se as métricas referentes aos códigos desempenhados no decorrer do capítulo 4:

Tabela 4 – Métrica Geral dos Experimentos

Métrica	***Kotlin***	***Java***
Caracteres utilizados	A	B
Linhas de código	A	B
Tamanho em *bytes*	A	A

Fonte: Elaborada pelo autor

Conforme apresentado na Tabela 4, sobre a linguagem *Kotlin* baseando-se unicamente nos algoritmos implementados e na formatação utilizada, facilita a menor utilização de caracteres especiais, símbolos e linhas de código, em comparação com a linguagem de programação *Java*, recebendo assim, o índice de qualidade A em todos os quesitos analisados, levando em consideração os experimentos de introdução, sendo eles: I - *Hello World*, II - operação aritmética de soma e III - classe *Student*, também como os experimentos que demonstraram a implementação de aplicação de conversão de medidas termométricas, sendo eles: IV - classe *ThermometricScale*, V - interface *Temperature*, VI - classe *Adapter* e VII - classe *Activity Main*.

4.1.2 Pontos Positivos e Negativos da Adoção do Kotlin

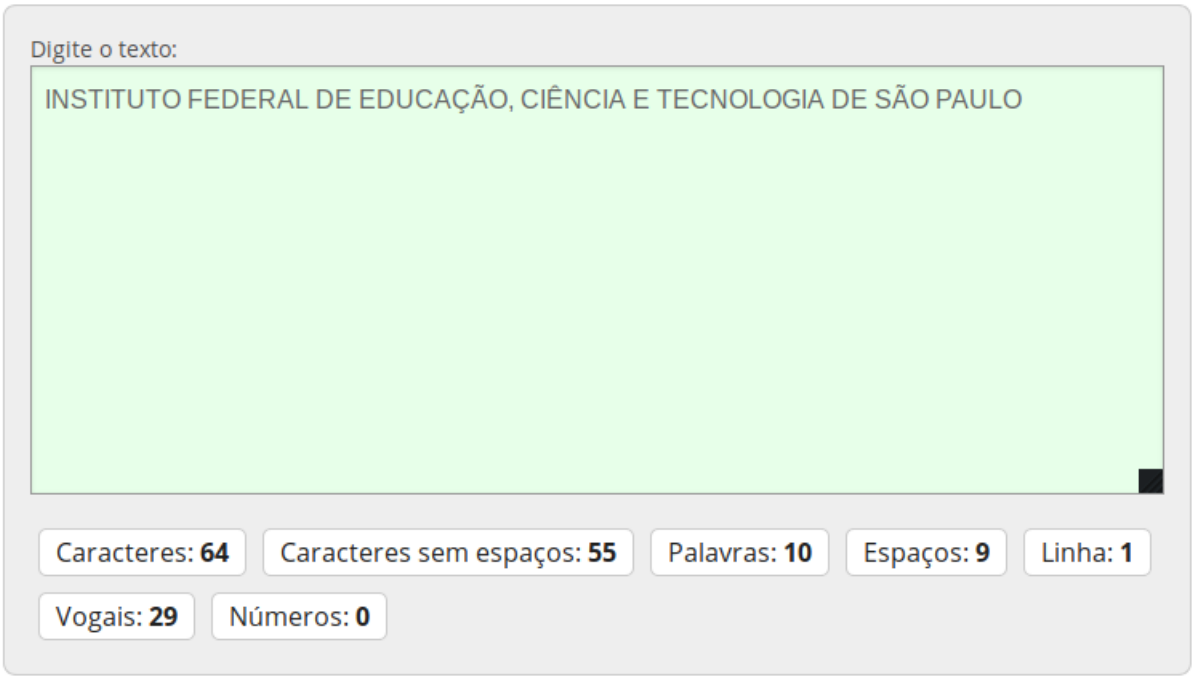
No ato de integrar-se em um projeto já existente, a compatibilidade com a linguagem de programação *Kotlin*, observa-se um aumento de alguns segundos no tempo de execução da aplicação, e em caso de uma aplicação escrita somente em *Kotlin*, o tempo de execução também é maior se comparada a uma aplicação desenvolvida em *Java*. Por mais que seja possível literalmente misturar códigos em *Java* e *Kotlin* no mesmo projeto e até no mesmo arquivo, observa-se que para aproveitar-se todos os recursos oferecidos pela nova linguagem, como tornar o código mais enxuto e aumentar sua flexibilidade, necessita-se de consultas e estudos a documentação, uma vez que o recurso de conversão de *Java* para *Kotlin* que é disponibilizado nativamente pela *IDE Android Studio* em versões mais recentes e por meio de *plugins* em versões anteriores, não realiza a conversão da forma mais enxuta e direta possível, mas sim da forma mais verbosa, como por exemplo, a não utilização do recurso de importação de todos os *widgets* utilizados no arquivo *XML* de *layout*, conforme demonstrado no experimento VII *activity main*.

O fato de *Kotlin* ser totalmente integrada com a *IDE Android Studio* torna a sua utilização/adoção bem atrativa e descomplicada. A compatibilidade da linguagem com *Java* torna o desenvolvimento mais leve, deixando a cargo do desenvolvedor, escolher qual sintaxe, estilo e padronização de escrita de código utilizar, como por exemplo optar por utilizar ponto e vírgula ao final de cada instrução (*Java*) ou não (*Kotlin*). A curva de aprendizado apresenta-se baixa para desenvolvedores com conhecimento prévio em *Java* e nas particularidades do desenvolvimento móvel voltado para o dispositivos móveis e o ecossistema da plataforma *Android*.

4.2 Análise Quantitativa para Comparação entre *Kotlin* e *Java*

Para efetuar a comparação entre as linguagens *Java* e *Kotlin*, utiliza-se ferramenta disponibilizada de forma gratuita no endereço *web* <https://www.4devs.com.br/analisar_textos>, que realiza a funcionalidade de contabilização de caracteres, palavras e de linhas de código. Conforme apresentado na Figura 13 a seguir, após inserir o algoritmo a ser analisado, logo abaixo do campo de texto, apresentam-se os campos onde são disponibilizados os valores.

Figura 13 – Ferramenta de contagem de caracteres, palavras e linhas



Fonte: <https://www.4devs.com.br/analisar_textos>

Nas subseções posteriores, apresentam-se as tabelas com as devidas métricas de quantidade de caracteres, linhas de código e tamanho em *bytes*, ambas as métricas obtidas por meio da disponibilização dos trechos de códigos utilizados nos experimentos, conforme demonstrado na Figura 13.

4.2.1 Experimento I

No exemplo de implementação de um simples e introdutório algoritmo de *Hello World*, *Kotlin* atingiu o objetivo, com uma menor quantidade de caracteres utilizados, linhas de código e consequentemente de *bytes*, conforme Tabela 5 a seguir.

Tabela 5 – Métricas *Hello World*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	39	116
Linhas de código	03	05
Tamanho em *bytes*	39	117

Fonte: Elaborada pelo autor

4.2.2 Experimento II

De acordo com a Tabela 6, no experimento de implementação de operação de soma, que retorna a soma de dois valores fixos e inteiros, interpolados com um pequeno trecho de texto, *Kotlin* obteve os menores índices em todas as métricas analisadas.

Tabela 6 – Métricas Operação Aritmética de Soma

Métrica	***Kotlin***	***Java***
Caracteres utilizados	81	159
Linhas de código	5	6
Tamanho em *bytes*	80	159

Fonte: Elaborada pelo autor

4.2.3 Experimento III

No experimento de implementação de uma classe que abstrai de maneira simples, as características de um estudante, *Kotlin* demonstra a possibilidade de evitar códigos desnecessários em algoritmos comumente utilizados, possibilitando a economia de mais de mil caracteres e 60 linhas de implementação, conforme Tabela 7.

Tabela 7 – Métricas Classe *Student*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	163	1.607
Linhas de código	4	67
Tamanho em *bytes*	163	1.600

Fonte: Elaborada pelo autor

4.2.4 Experimento IV

Com a implementação da classe *ThermometricScale*, de acordo com a Tabela 8, observa-se por meio da implementação da declaração de funções, parâmetros e seus respectivos retornos disponível em *Kotlin*, economiza-se diversas linhas de código, podendo variar de acordo com a abordagem do(s) desenvolvedor(s).

Tabela 8 – Métricas Classe *ThermometricScale*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	283	363
Linhas de código	9	16
Tamanho em *bytes*	285	364

Fonte: Elaborada pelo autor

4.2.5 Experimento V

No experimento no qual implementa-se a função da interface *Temperature* a ser implementada, *Kotlin* e *Java* utilizam a mesma quantidade de linhas de código, *Kotlin* obtém os resultados menores na métrica de caracteres utilizados e tamanho em *bytes*, conforme Tabela 9.

Tabela 9 – Métricas Interface *Temperature*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	143	155
Linhas de código	5	5
Tamanho em *bytes*	144	156

Fonte: Elaborada pelo autor

4.2.6 Experimento VI

De acordo com a Tabela 10, na implementação da classe que tem por objetivo ser um adaptador no aplicativo de conversão, demonstra-se que Java necessita de maiores quantidades de linhas de código, caracteres e um tamanho superior em *bytes*.

Tabela 10 – Métricas Classe *Adapter*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	605	741
Linhas de código	19	26
Tamanho em *bytes*	608	742

Fonte: Elaborada pelo autor

4.2.7 Experimento VII

Na classe principal do programa de conversão de Celsius para Fahrenheit e Kelvin, de acordo com a implementação utilizada, *Java* demonstra a necessidade da utilização de mais caracteres, aumentando a quantidade de linhas de código , consequentemente, o tamanho em *bytes*, conforme Tabela 11.

Tabela 11 – Métricas Classe *Activity Main*

Métrica	***Kotlin***	***Java***
Caracteres utilizados	2.404	3.179
Linhas de código	74	93
Tamanho em *bytes*	3.200	3.800

Fonte: Elaborada pelo autor

5 CONCLUSÕES

Este trabalho abordou um estudo analítico entre tecnologias diferentes, mas que servem a um mesmo propósito, e um estudo de usabilidade sobre a linguagem de programação *Kotlin*, referente à todo o ecossistema que envolve a plataforma *Android*.

Adicionalmente, foram apresentados, em caráter introdutório, as principais tecnologias que fazem parte do desenvolvimento voltado para os dispositivos móveis, desde os meios para a elaboração de aplicações tanto nativas e específicas para apenas uma plataforma, como aplicações que atingem ambos os públicos com tecnologias denominadas híbridas/multiplataforma. Também foram apresentados os conceitos de qualidade de código por meio de ferramentas e da reescrita, e detalharam-se alguns trabalhos com segmento semelhante ao contexto deste trabalho.

Baseando-se em três índices qualificadores de código, sendo eles, a quantidade de caracteres, as linhas de código utilizadas e o tamanho do arquivo em *bytes*, foram implementados 07 experimentos práticos, desde alguns algoritmos mais simples como operação matemática, *Hello World* e classe *model*, finalizando com uma aplicação móvel *Android* que realiza a conversão de medidas termométricas de Celsius para Fahrenheit e/ou Kelvin, por meio da interação com o usuário, baseada um padrão de projeto. De acordo com os códigos desempenhados foi realizada a comparação entre a linguagem *Kotlin* e *Java*, no qual, *Kotlin* mostrou-se, tão eficaz quanto Java, chegando ao mesmo resultado, com menor necessidade de caracteres, linhas de código e da mesma forma tamanho final em *bytes*, sendo superior em todos os experimentos implementados.

Somando-se todos os trechos de código demonstrados nos experimentos, em *Java* foram necessários: **6.320** caracteres, **218** linhas de código e **6.938** bytes. Por outro lado, com *Kotlin*, obteve-se os mesmos resultados finais, por meio de: **3.718** caracteres, **119** linhas de código e **4.480** bytes.

Foram apresentados os pontos negativos e positivos na adoção da tecnologia *Kotlin* como linguagem de programação, onde destacou-se como negativo o aumento no tempo de execução de projetos *Kotlin* e como positivo, a sua fácil adaptação para programadores com conhecimento prévio.

Para trabalhos futuros, o conceito da diminuição de utilização de códigos desnecessários no desenvolvimento de *software* apresentados neste trabalho pode motivar novas pesquisas relacionadas, como a comparação de tecnologias que propõe-se a entregar aplicações móveis multiplataforma e nativas, ou seja, para *Android* e *iOS*. As tecnologias e linguagens adotadas e desenvolvidas por grandes empresas que são referência no meio tecnológico, como a ferramenta *Flutter* que utiliza a linguagem *Dart* ambos projetos

criados pela *Google* e o *framework React Native* criado pelo *Facebook* que utiliza *JavaScript* são exemplos de alternativas emergentes no momento em que esse trabalho foi realizado. Outra opção de futuros projetos relacionados a este trabalho de conclusão de curso seria uma nova comparação entre as linguagens *Java* e *Kotlin*, mas com foco em *CRUD's*[1], desenvolvimento da Sequência de *Fibonacci*, operações matemáticas e cálculos mais elaborados e de algoritmos de encriptação para se medir a complexidade em cada uma das abordagens.

[1] Acrônimo em inglês para operações básicas de um sistema, sendo elas: C de *create* (inclusão), R de *read* (leitura), U de *update* (atualização) e D de *delete* (remoção) (CODEACADEMY, 2015-2019)

REFERÊNCIAS

ADMINISTRADORES. **O poder das metas case reunião de MCI**. 2015–2019. Disponível em: <https://administradores.com.br/artigos/o-poder-das-metas-case-reuniao-de-mci>. Acesso em: 19 de dez. de 2019. Citado na página 31.

APPLE. **Swift. Uma linguagem aberta e poderosa para todo mundo criar apps incríveis.** 2019. Disponível em: <https://www.infoq.com/br/articles/apple-swift>. Acesso em: 13 de abr. de 2019. Citado na página 35.

ARAUJO, E. C. d. **Xamarin Forms: Desenvolvimento de aplicações móveis multiplataforma**. São Paulo: Casa do Código, 2017. 309 p. Citado na página 36.

AZEVEDO, M. **Existe avaliação e garantia de qualidade no seu código durante o desenvolvimento e depois do deploy?** 2018. Citado na página 41.

BATISTA, S. H. d. A. **Qualidade de Software: dicas para escrever um código de qualidade**. 2016. Disponível em: <http://www.linhadecodigo.com.br/artigo/3460/qualidade-de-software-dicas-para-escrever-um-codigo-de-qualidade.aspx>. Acesso em: 27 de abr. de 2019. Citado na página 39.

BRITO, M. **Ciência da computação: é difícil aprender a programar?** 2018. Disponível em: <http://blog.unipe.br/graduacao/ciencia-da-computacao-e-dificil-aprender-a-programar>. Acesso em: 20 de abr. de 2019. Citado na página 29.

BURD, B. **Começando a Programar em Java Para Leigos**. Porto Alegre: Alta Books, 2014. 452 p. Citado na página 32.

CANALTECH. **O que é open source?** 2017. Disponível em: <https://canaltech.com.br/produtos/O-que-e-open-source/>. Acesso em: 03 de maio de 2019. Citado na página 33.

CAUZZI, E. J. **Proposta de plano de garantia da qualidade de software para o laboratório de criação e aplicação de software**. Caxias do SUl: UNIVERSIDADE DE CAXIAS DO SUL, CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO, BACHARELADO EM SISTEMAS DE INFORMAÇÃO, 2015. Citado 2 vezes nas páginas 23 e 44.

CODEACADEMY. **What is CRUD?** 2015–2019. Disponível em: <https://www.codecademy.com/articles/what-is-crud>. Acesso em: 19 de dez. de 2019. Citado na página 72.

DEITEL, P.; DEITEL, H. **Java como Programar**. 10. ed. São Paulo: Pearson Education do Brasil, 2016. 968 p. Citado na página 33.

DEITEL, P.; DEITEL, H.; WALD, A. **Android 6 para Programadores - 3ª Edição: Uma Abordagem Baseada em Aplicativos**. 3. ed. Porto Alegre: Bookman Editora, 2016. 456 p. Citado na página 33.

EBERT. **Get thorough code reviews and ship with confidence**. 2017–2019. Citado 2 vezes nas páginas 42 e 43.

FERRAZ, M. H. d. S. **Framework para Avaliação da Qualidade de Código: Uma Abordagem Baseada em Valor**. Brasília: Universidade de Brasília, 2016. Citado 2 vezes nas páginas 23 e 44.

FERREIRA, G. **O Guia do Programador Iniciante: Programar não é só escrever código**. 2018. Disponível em: <http://gabsferreira.com/o-guia-do-programador-iniciante-programar-nao-e-so-escrever-codigo/>. Acesso em: 20 de abr. de 2019. Citado na página 29.

FILHO, G. L. F. d. S. **Desenvolvimento de aplicativo para adoção de animais abandonados utilizando a linguagem de programação Kotlin e programação reativa**. Curitiba: Universidade Tecnológica Federal do Paraná - Câmpus Curitiba, 2017. Citado 3 vezes nas páginas 23, 43 e 44.

FOWLER, M. **Refatoração: Aperfeiçoamento e Projeto**. Porto Alegre: Bookman Editora, 2009. 365 p. Citado 2 vezes nas páginas 25 e 26.

GOIS, A. **Ionic Framework: Construa aplicativos para todas as plataformas mobile**. São Paulo: Casa do Código, 2017. 162 p. Citado 3 vezes nas páginas 35, 36 e 37.

GOMES, A. F. **Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio**. São Paulo: Casa do Código, 2014. 176 p. ISBN 9788566250992. Disponível em: <https://books.google.com.br/books?id=zHCCCwAAQBAJ>. Citado 2 vezes nas páginas 30 e 31.

GOOGLE. **Android Runtime (ART) and Dalvik**. 2017–2019. Disponível em: <https://source.android.com/devices/tech/dalvik>. Acesso em: 19 de dez. de 2019. Citado na página 38.

GUERRA, E. **Design Patterns com Java: Projeto orientado a objetos guiado por padrões**. São Paulo: Casa do Código, 2018. 305 p. Citado na página 26.

IDC, I. D. C. **Smartphone Market Share**. 2018. Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>. Acesso em: 11 de nov. de 2018. Citado 2 vezes nas páginas 33 e 34.

INFOQ. **O Ecossistema Android e além, de acordo com Tim Bray**. 2011. Disponível em: <https://www.infoq.com/br/news/2011/07/ecossistema-android/>. Acesso em: 19 de dez. de 2019. Citado na página 37.

JUNEAU, J. **Project Lombok: Clean, Concise Java Code**. 2009–2019. Disponível em: <https://www.oracle.com/corporate/features/project-lombok.html>. Acesso em: 28 de fev. de 2019. Citado na página 26.

JUNIOR, A. **Aplicativos e desenvolvimento mobile híbrido x nativo**. 2018. Disponível em: <https://imasters.com.br/desenvolvimento/aplicativos-e-desenvolvimento-mobile-hibrido-x-nativo>. Acesso em: 20 de abr. de 2019. Citado na página 32.

KEIRóZ, K. **DevQA: Como medir qualidade de código ?** 2015. Citado na página 41.

KOENIG, K. **Coding for Android on steroids with Kotlin**. 2016. Disponível em: <https://www.slideshare.net/AgentK/coding-for-android-on-steroids-with-kotlin>. Acesso em: 03 de maio de 2019. Citado na página 39.

LECHETA, R. R. **Desenvolvendo para iPhone e iPad**. 6. ed. São Paulo: Novatec, 2016. 520 p. Citado na página 34.

LECHETA, R. R. **Android Essencial com Kotlin**. 2. ed. São Paulo: Casa do Código, 2018. 536 p. Citado 2 vezes nas páginas 27 e 37.

LINDSTROM, S. **Refatoração de CSS: Organize suas folhas de estilo com sucesso**. São Paulo: NOVATEC, 2017. ISBN 9788575225370. Disponível em: <https://books.google.com.br/books?id=BJruDQAAQBAJ>. Citado na página 39.

MACHADO, G. **A linguagem de programação Swift**. 2015. Disponível em: <https://www.infoq.com/br/articles/apple-swift>. Acesso em: 13 de abr. de 2019. Citado na página 35.

MARTIN, R. C. **Código Limpo Habilidades Práticas do Agile Software - Edição Revisada**. Jacaré: Alta Books, 2009. 456 p. Citado 2 vezes nas páginas 25 e 26.

MENDES, F. P. **Estimando atributos de qualidade de software utilizados e desejados pelas empresas certificadas no MPS-SW no Brasil**. João Monlevade: Universidade Federal de Ouro Preto Departamento de Computação e Sistemas, Colegiado de Engenharia de Computação, 2018. Citado 2 vezes nas páginas 24 e 45.

RESENDE, K. **Kotlin com Android: Crie aplicativos de maneira fácil e divertida**. São Paulo: Casa do Código, 2018. 364 p. Citado 3 vezes nas páginas 26, 37 e 38.

SCHILDT, H. **Java para Iniciantes**. 6. ed. Porto Alegre: Bookman Editora, 2015. 704 p. Citado na página 32.

SCUDERO, E. **Como escolher a primeira linguagem de programação?** 2017. Disponível em: <https://becode.com.br/primeira-linguagem-de-programacao/>. Acesso em: 20 de abr. de 2019. Citado 2 vezes nas páginas 29 e 30.

SONARQUBE. **Continuous Inspection**. 2008–2019. Disponível em: <https://www.sonarqube.org/features/clean-code/>. Acesso em: 03 de maio de 2019. Citado na página 42.

STEIL, R. **iOS: Programe para iPhone e iPad**. São Paulo: Casa do Código, 2015. 264 p. Citado 2 vezes nas páginas 34 e 35.

TUCKER, A.; NOONAN, R. **Linguagens de Programação - 2.ed.: Princípios e Paradigmas**. 2. ed. Porto Alegre: AMGH Editora, 2009. 611 p. Citado na página 25.

TURINI, R. **Desbravando Java e Orientação a Objetos: Um guia para o iniciante da linguagem**. São Paulo: Casa do Código, 2014. 225 p. Citado na página 26.

VASCONCELLOS, L. **Apps Híbridas com Cordova e Ionic**. 2017. Disponível em: <https://bit.ly/2PJBenj>. Acesso em: 03 de maio de 2019. Citado na página 37.

WILDT, D. et al. **eXtreme Programming: Práticas para o dia a dia no desenvolvimento ágil de software**. São Paulo: Casa do Código, 2015. ISBN 9788555191077. Disponível em: <https://books.google.com.br/books?id=S2qCCwAAQBAJ>. Citado na página 31.

ZANETTE, A. **Clean Code: boas práticas para manter o seu código limpo!** 2017. Disponível em: <https://becode.com.br/clean-code/>. Acesso em: 26 de abr. de 2019. Citado 2 vezes nas páginas 39 e 40.