

DESENVOLVIMENTO MOBILE



EDITORA
IFB

Reitora
Luciana Miyoko Massukado

Pró-Reitora de Ensino
Veruska Ribeiro Machado

Pró-Reitor de Extensão e Cultura
Paulo Henrique Sales Wanderley

Pró-Reitora de Pesquisa e Inovação
Giovanna Megumi Ishida Tedesco

Pró-Reitor de Administração
Rodrigo Maia Dias Ledo

Pró-Reitor de Gestão de Pessoas
José Anderson de Freitas Silva

Coordenação de Publicações
Mariana Carolina Barbosa Rêgo

Produção Executiva

Sandra Maria Branchine

Conselho Editorial
Ana Paula Caetano Jacques
Francisco Das Chagas Roque Machado
Girlane Maria Ferreira Florindo
Jocenio Marquios Epaminondas
Josué de Sousa Mendes
Juliana Rocha de Faria Silva
Juliana Estanislau de Ataíde Mantovani
Larissa Dantas de Oliveira
Maurilio Tiradentes Dutra
Mariana Carolina Barbosa Rêgo
Nívia Aniele Oliveira
Tatiane Alves de Melo

Diretora de Educação a Distância - DEaD
Jennifer de Carvalho Medeiros

Pedagoga
Carolina Novaes Xavier de L. Reynaldo

Apoio administrativo
Cláudia Sabino Fernandes
Joscélia Moreira de Azevedo
Noeme César Gonçalves

Coordenador Geral do Programa Novos Caminhos
Hênio Delfino Ferreira de Oliveira

Coordenadora Adjunta Administrativo
Cláudia Sabino Fernandes

Coordenadora Adjunta de Ensino
Luciana Brandão Dourado

Coordenadora Adjunta de Produção de Conteúdos Educacionais para EaD
Sylvana Karla da Silva de L. Santos

Coordenador Adjunto de Tecnologia
Hugo Silva Faria

Revisão de conteúdo
Elaine Cavalcante Menezes
Karina Mendes Nunes Viana
Lidiane Szerwinsk Camargos

Diagramação/Ilustrações
Erika Ventura Gross
Fábio Lucas Vieira
Márlon Cavalcanti Lima

Organização
Cláudia Sabino Fernandes
Joscélia Moreira de Azevedo
Noeme César Gonçalves

EDITORA

Reitoria – SGAN Qd 610, módulos D, E, F, G
CEP: 70860-100 Brasília-DF
www.ifb.edu.br
Fone: +55 (61) 2103-2108
editora@ifb.edu.br

2020 Editora IFB

A exatidão das informações, as opiniões e os conceitos emitidos nos capítulos são de exclusiva responsabilidade dos autores. Todos os direitos desta edição são reservados à Editora IFB. É permitida a publicação parcial ou total desta obra, desde que citada a fonte. É proibida a venda desta publicação.

D451 Desenvolvimento mobile / organizadoras: Cláudia Sabino Fernandes, Joscélia Moreira de Azevedo, Noeme César Gonçalves – Brasília: Editora IFB, 2022.
1 E-book : 387 p. : il. ; PDF.

Inclui bibliografia.
ISBN 978-85-64124-88-2

1. Dispositivos móveis. 2. Computação móvel. 3. Programação. 4. Arquitetura de sistemas operacionais. I. Fernandes, Cláudia Sabino. II. Azevedo, Joscélia Moreira de. III. Gonçalves, Noeme César. IV. Título.

CDU 004.42

Sumário

MÓDULO I: INTRODUÇÃO AO DESENVOLVIMENTO MOBILE

APRESENTAÇÃO 9

UNIDADE 1 – INTRODUÇÃO AO MUNDO DOS APLICATIVOS MÓVEIS 10

1.1 Entendendo o mundo à nossa volta 10
1.2 Configurando o ambiente 18
1.3 Executando o primeiro projeto 29

UNIDADE 2 – REPARANDO O PROJETO DE APLICATIVOS MÓVEIS 43

2.1 Entendendo a arquitetura da aplicação 43
2.2 Criando a primeira aplicação 52

UNIDADE 3 – DESENVOLVENDO UM APLICATIVO COMPLETO 70

3.1 Construindo o projeto 71
3.2 Trabalhando o *layout* 75
3.3 Criando as funcionalidades 91

UNIDADE 4 – PUBLICANDO O APLICATIVO 104

4.1 Últimos ajustes da aplicação 104
4.2 Gerando o aplicativo instalável 125
4.3 Publicando aplicativos nas lojas 132

REFERÊNCIAS 136

CURRÍCULO DO PROFESSOR AUTOR 137

MÓDULO II: ARQUITETURA DE SISTEMAS MOBILE

UNIDADE 1 - SISTEMAS OPERACIONAIS - HISTÓRIA E PLATAFORMAS **140**

1.1 Introdução aos Sistemas Operacionais 140
1.2 História dos Sistemas Operacionais 148
1.3 História da Mobilidade 155
1.4 Sistemas Operacionais *Desktop* 170
1.5 Sistemas Operacionais para Dispositivos Móveis e Mercado 174

UNIDADE 2 - CONCEITOS BÁSICOS DOS SISTEMAS OPERACIONAIS **188**

2.1 Abstração e gerência de recursos 190
2.2 Tipos de sistemas operacionais 191
2.3 Funcionalidades e conceitos de *hardware* 195
2.4 Estrutura de um sistema operacional 204
2.5 Arquiteturas de sistemas operacionais 205
2.6 Gerência de processos 215
2.7 Gerência de memória 219
2.8 Gerência de entrada e saída 220
2.9 Sistemas de arquivos 221

UNIDADE 3 - ARQUITETURA DOS SISTEMAS OPERACIONAIS DE DISPOSITIVOS MÓVEIS **227**

3.1 Arquitetura do Sistema *Android* 228
3.2 Arquitetura do Sistema *Apple iOS* 236
3.3 Arquitetura do *Windows Phone* 238
3.4 Desenvolvimento de Aplicativos para Diferentes Sistemas Operacionais 241

UNIDADE 4 - VIRTUALIZAÇÃO **251**

4.1 Fundamentos da Virtualização 253
4.2 A construção de máquinas virtuais 258
4.3 Tipos de máquinas virtuais 263
4.4 Técnicas de virtualização 266
4.5 Ambientes de máquinas virtuais 268
4.6 Máquina Virtual *Android* (Dalvik, ART (*Android Runtime*)) 272

REFERÊNCIAS **279**

MÓDULO III: ANÁLISE E PROJETO DE SISTEMAS

UNIDADE 1 – LEVANTAMENTO, ANÁLISE E NEGOCIAÇÃO DE REQUISITOS **282**

1.1 Projeto de *software* 282

1.2 A importância da Engenharia de Requisitos no projeto de *software* 285

1.3 Principais conceitos: atores (organizações, sistemas, clientes) , *stakeholders*, fronteiras, restrições 289

1.4 Método 5W2H (*What, Who, Why, When, Where, How, How much*) 295

1.5 Classificação de requisitos 297

UNIDADE 2 - MODELAGEM, ESPECIFICAÇÃO, VALIDAÇÃO E VERIFICAÇÃO DE REQUISITOS **301**

2.1 A importância da modelagem e suas ferramentas 301

2.2 Técnicas de levantamento de requisitos 306

2.3 Técnicas de especificação de requisitos - Orientação a objetos e métodos ágeis 312

2.4 Processo de verificação e validação de requisitos 328

Questões de autoaprendizagem 333

UNIDADE 3 - CARACTERIZAÇÃO E APLICAÇÃO DE METODOLOGIAS E FERRAMENTAS DE MODELAGEM DE SISTEMAS ORIENTADOS A OBJETOS **335**

3.1 Conceitos de modelagem de sistemas, modelos e suas perspectivas 335

3.2 Etapas do processo de desenvolvimento de *software* e metodologias 349

Questões de autoaprendizagem 359

UNIDADE 4 – UML E SEUS DIAGRAMAS E INTRODUÇÃO A DESIGN PATTERNS **361**

4.1 Linguagem de Modelagem Unificada (UML) 361

4.2 Diagrama de Caso de uso e descrição de caso de uso 363

4.3 Diagrama de atividades 365

4.4 Diagrama de classes 371

4.5 Outros diagramas 377

4.6 Introdução a padrões do projeto 379

Questões de autoaprendizagem 384

REFERÊNCIAS **386**

CURRÍCULO DA PROFESSORA AUTORA **388**

Apresentação

Querido leitor e querida leitora.

É um prazer para nós saber que está dedicando seu precioso tempo para a leitura desse e-book. Mas, antes que você inicie essa jornada, podemos te contar um pouco de como foi a produção desse material?

Esse *e-book* é o resultado da compilação de material elaborado para o curso Técnico em Informática, ofertado pelo Instituto Federal de Educação, Ciência e Tecnologia de Brasília.

Esse curso, realizado na modalidade a distância, por meio de fomento da bolsa-formação do PRONATEC, foi ofertado entre os anos de 2017 e 2019 e a elaboração do material prezou por uma linguagem dialogada com estudantes que estavam do outro lado da tela do computador ou do celular.

Por isso, te convidamos a fazer essa leitura aproveitando a linguagem dialogada e sentindo-se parte da caminhada que fizemos com nossos estudantes.

Ele está organizado da seguinte maneira: no Módulo I, veremos, Introdução ao Desenvolvimento *Mobile*, no Módulo II, Arquitetura de Sistemas *Mobile* e, no Módulo III, Análise e Projeto de Sistemas .

Desejamos que esse *e-book* seja uma jornada de aprendizagem para você!

E, além de aproveitar um pouquinho do material de nossos cursos, quem sabe não te incentivamos a vir nos conhecer melhor e estudar conosco, não importa se na modalidade presencial ou a distância!

Boa leitura!

MÓDULO I

INTRODUÇÃO AO DESENVOLVIMENTO MOBILE

Daniel Celestino de Freitas Pereira



O Módulo I desta obra, Introdução do Desenvolvimento *Mobile*, pretende apresentar você ao mundo da programação para dispositivos móveis! Ele está estruturado em 4 unidades temáticas com tópicos específicos, conforme descrito abaixo.

Unidade 1: trata das evoluções tecnológicas para entender em que mundo estamos e sobre os clientes que nos esperam. Serão abordados também os aspectos básicos para criação de um ambiente para desenvolvimento das aplicações, utilizando o sistema operacional *Android* e a linguagem de programação Java.

Unidade 2: apresenta a arquitetura das aplicações para Sistema Operacional *Android* e as etapas de criação do primeiro projeto funcional.

Unidade 3: trata do desenvolvimento de um aplicativo completo, trabalhando desde os detalhes do *layout* às funcionalidades da aplicação.

Unidade 4: ensina a gerar os aplicativos e como publicá-los nas lojas.

Vamos começar?

Unidade 1

INTRODUÇÃO AO MUNDO DOS APLICATIVOS MÓVEIS

Nesta unidade, iremos dar os primeiros passos no mundo dos aplicativos móveis, mas, para isso, vamos também buscar uma visão panorâmica do mundo em que vivemos, entendendo o contexto tecnológico à nossa volta. Isso é importante para planejarmos e entendermos os novos rumos da tecnologia e, é claro, para compreendermos qual a interferência disso em nossas aplicações. Veremos ainda nesta unidade, a configuração inicial necessária para o nosso processo de desenvolvimento de outros aplicativos móveis. Vamos lá!

1.1 ENTENDENDO O MUNDO A NOSSA VOLTA

Então, pessoal, como já falamos, nosso principal objetivo aqui é incentivá-los a construir jogos *mobile*, mas já pararam para se perguntar o por quê?

— Como assim, professor?

—Quero levá-los a pensar um pouco antes de entrarmos na resposta.



Como chegamos ao estágio atual de nossa tecnologia mundial? Como era a tecnologia antigamente? Qual a era em que vivemos atualmente?

Para ajudá-los a refletir sobre isso, gostaria que vocês assistissem ao vídeo abaixo:



Você está preparado para a Transformação Digital?

https://www.youtube.com/watch?v=VMguBZEkylM

Interessante, não é mesmo? Mas começamos de trás para frente. Que tal voltarmos um pouco e entendermos a história da evolução tecnológica. Vamos lá?



Indústria 4.0 - A Quarta Revolução industrial.

https://www.youtube.com/watch?v=0SYlggPuOM8

O que podemos analisar desse breve histórico?

Nos primeiros séculos da civilização pós-Cristo, o homem dependia de força animal para sua produção. Eram os animais que funcionavam como força para o trabalho mecânico, como: moinhos de água, moedura de trigo e outros cereais e moinhos de vento.

Foi então que, em meados do século XVIII, com a descoberta do ferro e do carvão, em um processo iniciado na Inglaterra, o homem foi capaz de inventar a máquina a vapor, tendo como uma das principais aplicações, a fabricação de fios e de tecidos.

Essa geração de riquezas trouxe mudanças na capacidade e na velocidade de produção, o que consequentemente acarretou modificações significativas na economia e na sociedade como um todo, inclusive no que tange ao número de pessoas e ao espaço geográfico, pois as cidades começaram a crescer. Nesse cenário, aumentou a quantidade de profissões, foram criadas ferrovias expandindo a capacidade de trânsito de mercadorias e de pessoas, o campo começou a receber mecanização, novas mercadorias e novas fábricas foram sendo criadas. Podemos dizer que essa revolução é marcada pelo carvão e pelo ferro ou, melhor dizendo, a máquina a vapor mecanizando a produção.

—Estou lembrado de ter estudado esse assunto, professor!
—Muito bem! Vamos em frente!

Na segunda metade do século XIX, após a entrada de outros países nesse processo de Revolução Industrial, iniciou-se um período marcado pela descoberta da energia elétrica e do uso de petróleo como combustível. Basta olhar para os os dias atuais que conseguimos entender qual o impacto dessas duas descobertas para o mundo! Não conseguiríamos pensar nos demais avanços da sociedade, sem passar pelas descobertas desse período, não é mesmo!?

—Podemos então dizer, professor, que, para a segunda Revolução Industrial, estamos falando de energia elétrica e petróleo como marcos desse período?
—Isso mesmo, pessoal! Vamos para a terceira Revolução Industrial?

Após a Segunda Guerra Mundial, em meados do século XX, iniciou-se um período em que grandes mudanças na indústria também puderam ser percebidas. Tais mudanças se devem ao uso de tecnologia na produção, ou melhor, a tecnologia foi além da produção! Esse período foi marcado pelo surgimento da Internet. Entre essas mudanças, podemos destacar: utilização de várias fontes de energia (petróleo, energia hidrelétrica, nuclear, eólica etc), uso crescente de tecnologia (informática, nesse caso) na produção (robótica por exemplo), surgimento dos computadores pessoais (PC's), telefones celulares, tablets, biotecnologia, dentre outras. Importante destacar que a ciência e a tecnologia passam a andar cada vez mais de mão dadas. Vocês poderiam então me dizer qual o grande marco dessa Revolução?

—Sim, professor, é o uso da tecnologia!

—Isso mesmo! Tanto que ela é chamada de Revolução Tecnocientífica!



Pessoal, vocês acham que a terceira Revolução Industrial teve seu fim quando?

—Não sei, professor!

—Nem eu! Rs. Brincadeira! Na verdade, ainda estamos na terceira Revolução Industrial! Ainda estamos utilizando a tecnologia e, cada vez mais, aumentando a produção, demonstrando que há um relacionamento cada vez maior entre ciência e tecnologia. A robótica vem ganhando mais espaço a cada dia, ou seja, estamos ainda na terceira Revolução!

—Ué professor, já ouvimos falar da 4ª Revolução Industrial. Então, ela não existe?

—Vou explicar!

Pessoal, ainda não concluímos a terceira Revolução Industrial e já entramos na quarta. É isso mesmo! Elas andam em paralelo, segundo muitos teóricos e estudiosos do ramo. A quarta Revolução Industrial é, na verdade, uma resposta às grandes mudanças da sociedade atual. Os PC's deram lugar aos smartphones e a informação, que já era facilitada por meio da Internet e da globalização, agora é compartilhada com um número cada vez maior de pessoas e em velocidades cada vez mais altas.

As transformações sociais também são cada vez mais rápidas! Uma ideia que, antes, era escrita em um livro e gradualmente refletia impactos na sociedade, agora, pode transformar uma sociedade por meio de algumas mensagens de celular, que se espalham rapidamente e de uma forma incontrolável. As empresas, se quiserem continuar existindo, precisam estar prontas para se adaptar às rápidas mudanças sociais e para se reinventar. Cada vez mais, a inteligência artificial tem produzido grandes resultados. As peças e, até mesmo casas, são produtos de impressoras 3D. A Internet, agora, passa a conectar também as coisas e não apenas as pessoas (Internet das coisas). Criou-se o dinheiro virtual (*Bitcoin*) e já estamos falando em inteligência artificial afetiva! É nesse cenário que vivemos hoje!



Figura 1: Meme sobre IOT.

Fonte:: https://www.gerarmemes.com.br/memes-galeria/66-me-solta/140

Este pequeno resumo sobre as quatro revoluções pode ser aprofundado nos seguintes links:

https://educacao.uol.com.br/disciplinas/geografia/revolucoes-industriais-primeira-segunda-e-terceira-revolucoes.htm

http://www.administradores.com.br/artigos/marketing/as-4-revolucoes-industriais/59504/

https://conaenge.com.br/4-revolucoes-industriais-processos-fabricacao/

Vamos ver mais um vídeo sobre a Quarta Revolução Industrial?



Quarta Revolução Industrial.
https://www.youtube.com/watch?v=jTLpqipsw0g

—É, professor! Isso realmente impressiona! Mas o senhor me permite uma pergunta: o que esse assunto tem a ver com nossa matéria de Introdução ao Desenvolvimento Mobile?
—Boa pergunta!

Pessoal, estou conduzindo vocês na busca do conhecimento! Na verdade, é preciso entender o cenário de forma contextualizada, sempre que formos resolver um problema, e é isso que pretendo fazer um pouco com vocês aqui. É preciso aprender a pensar, aprender a pesquisar, aprender a aprender! Nada é mais sensato que entender nossa sociedade atual, pois é nela que estarão nossos clientes de aplicativos! E vou mais longe, se não soubermos entender as evoluções, não conseguiremos entender as expectativas dos usuários de nossas aplicações! Entenderam?

—Entendi, professor! Agora faz mais sentido!

—Por falar em clientes, saberiam como os estudiosos classificam os nossos clientes dos tempos atuais? Vamos entender um pouco disso antes de entrarmos no mundo dos jogos? Venham comigo!

A primeira classificação de consumidor é o 1.0. O Consumidor 1.0 surgiu nos anos em que a Internet nascia. Podemos até lembrar da Internet discada! (Talvez, alguns de vocês não a tenham conhecido! Rs). Nessa época, os consumidores tinham acesso a uma espécie de lista telefônica on-line, isto é, os serviços começaram a ser oferecidos de forma estática e com pouca interação. Dessa forma, o cliente dependia totalmente das empresas, pois eram estas que disponibilizavam os serviços da maneira que bem entendessem.

O Consumidor 2.0 passou a ser exigente e começou a tomar o controle da relação com as empresas. Eles, os consumidores, passaram a procurar por serviços mais baratos e na qualidade que desejassem. Mas, mesmo assim, nesse cenário, o consumidor precisava ir até a empresa.

Então, chegamos ao consumidor 3.0, ou seja, ao consumidor que temos hoje em dia. Aqui a relação mudou de vez! O Consumidor que manda na relação e a empresa deve ir atrás do consumidor para continuar existindo. Com o consumidor altamente conectado e com a disseminação das mídias sociais, uma simples mensagem pode acabar com a reputação de uma empresa e, inclusive, levá-la à falência!



Figura 2: Quadrinhos sobre consumidores.
Fonte: https://br.pinterest.com/pin/16255248627290912

Vejam o vídeo abaixo:



A Evolução do Consumidor.
https://www.youtube.com/watch?v=zWm40F79As4

Entendem então qual o cliente que nos espera, não é, pessoal? Mas antes de darmos nossos primeiros passos no mundo dos aplicativos móvies, queria deixar um último vídeo como reflexão. OK?



Evolução da Tecnologia.
https://www.youtube.com/watch?v=eJTQGym1Hl4

Vale a pena refletir um pouco sobre isso, não é mesmo!?

Então, pessoal, entenderam por que estudamos essa introdução? Podemos seguir em frente conhecendo nosso cliente? Podemos entrar no mundo dos aplicativos *Mobile*?!?!

SIM!!!

1.2 CONFIGURANDO O AMBIENTE

Neste tópico, vamos aprender a configurar o ambiente no qual iremos trabalhar. Começaremos com a instalação do Java.

1.2.1 Instalando ou atualizando o Java

Pessoal, vamos começar com uma pergunta:



Sabemos que vamos programar para *Android* e com a linguagem Java. Mas como verificar se temos java na máquina e qual a versão?

— Eu já vi isso, professor, em outras disciplinas, mas não estou lembrando!

— Sem problemas, pessoal! Temos basicamente duas formas: uma via cmd (command do Windows) e outra por meio das aplicações. Explico melhor...

A primeira coisa a ser feita é verificar qual versão do Java está instalada na máquina. Para fazer isso, por meio de linha de comando, precisamos ir até a linha de comando e digitar "cmd" e, na linha de comando, digitar: "java -version".

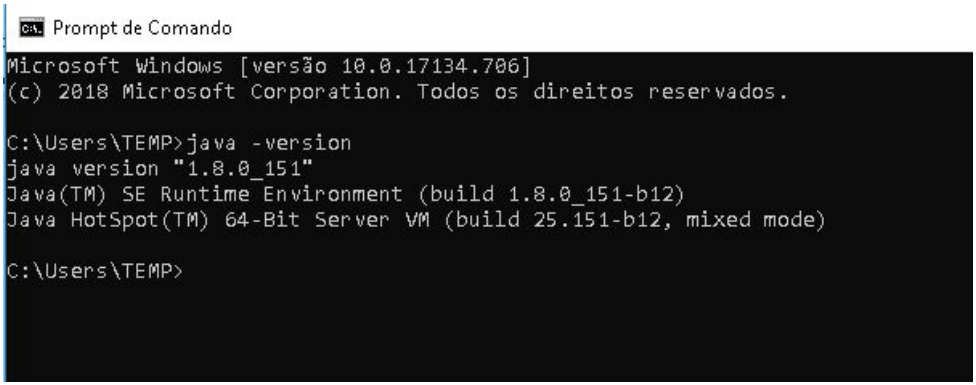


Figura 3: Verificando a versão do Java por linha de comando.

A segunda é por meio da barra de tarefa do Windows (apenas versões mais recentes permitem isso). Basta digitar java, na barra de tarefa, e selecionar o “Sobre o Java”.

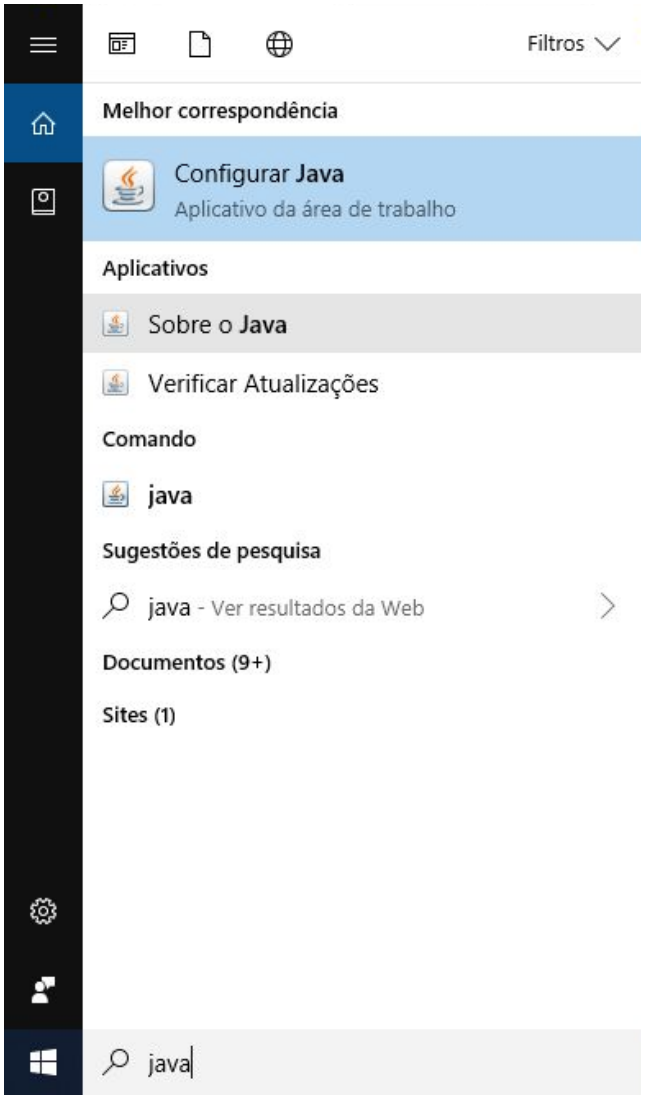


Figura 4: Tela da barra de tarefa pesquisando "java".

Feito isso, uma tela similar a esta aparecerá:



Figura 5: Tela do "Sobre o Java".

Para equalizar a versão do Java ou instalar (caso não tenha na máquina), basta pesquisar no Google por “java sdk” ou ir ao link:

https://www.oracle.com/technetwork/pt/java/javase/downloads/index.html.

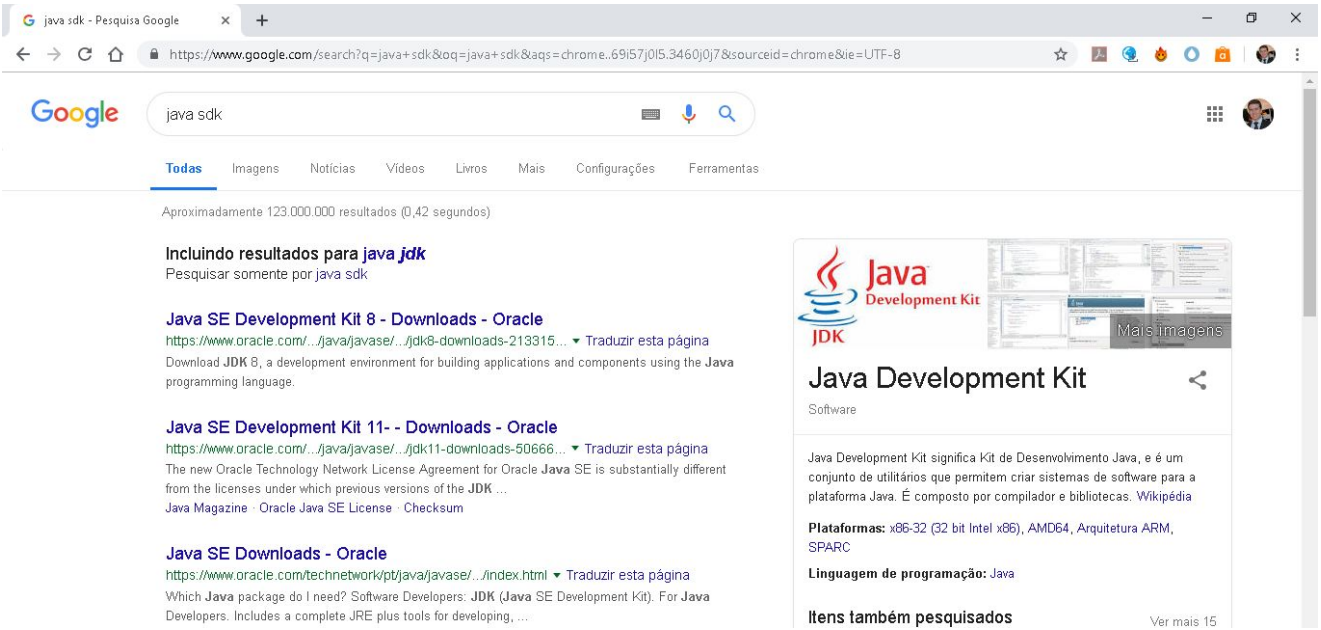


Figura 6: Pesquisa por "java sdk".

Na tela de *download* do Java, não é necessário baixar o Netbeans, pois não utilizaremos em nossos estudos.

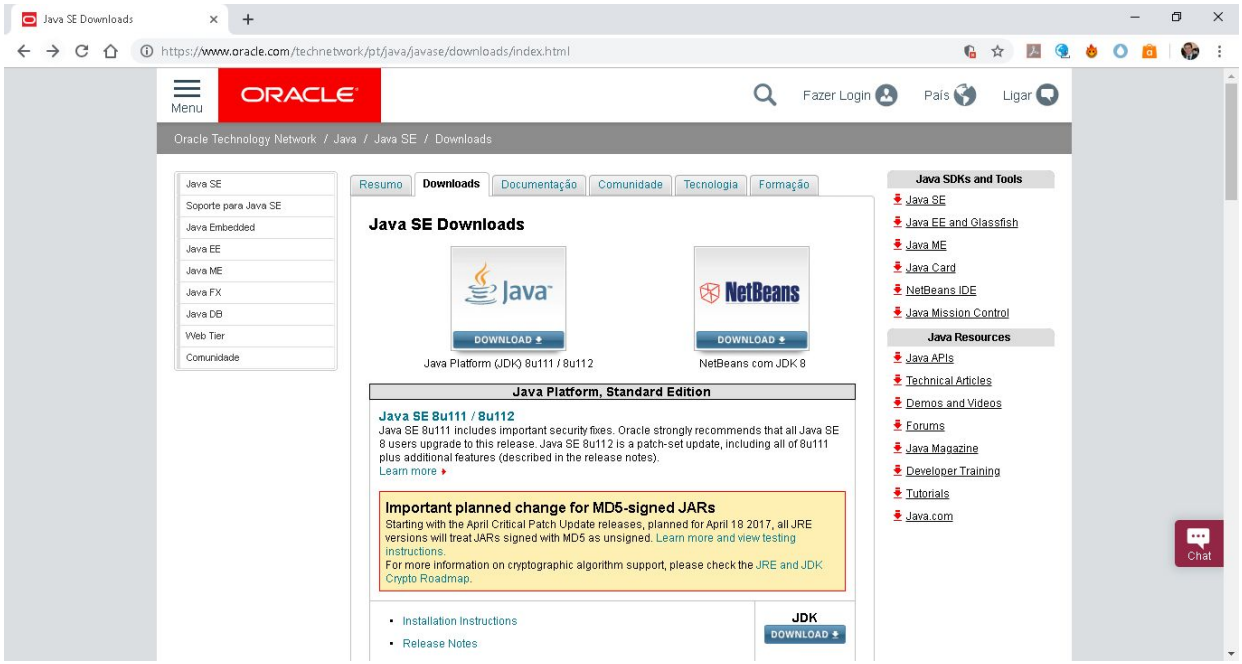


Figura 7: Download do Java.

Ao clicar no Java, aparecerá uma outra tela e, então, selecione:

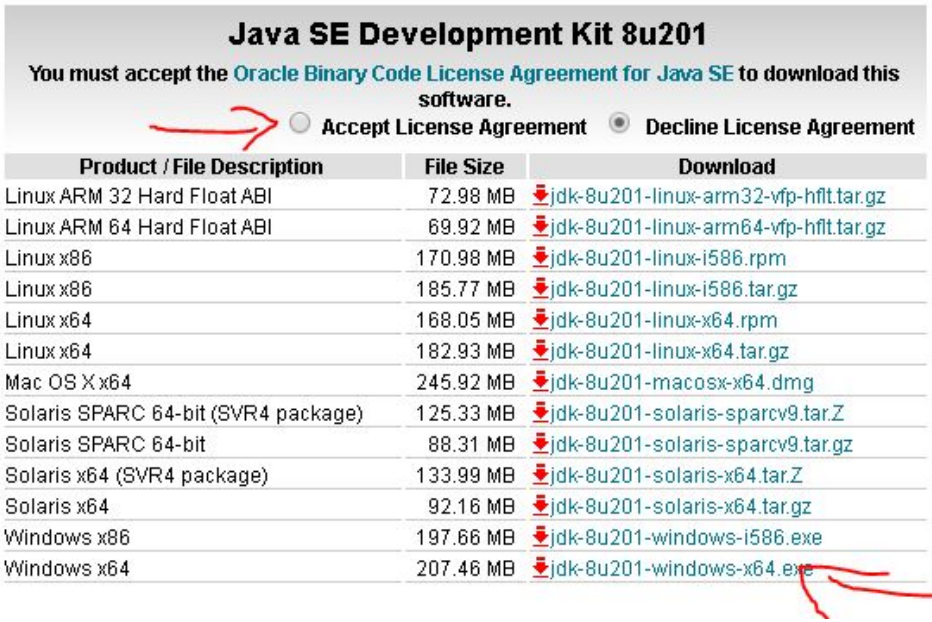


Figura 8: Tela de configurações do *download* do Java.

A instalação do Java, após concluído o *download*, é padrão “next....next”. A grande dica é não alterar a pasta padrão do Java.

1.2.2 Instalando *Android* Studio

Após a instalação do Java, é preciso instalar o *Android* Studio. Para isso, pesquise no Google o termo: *android studio*” ou vá ao link: https://developer.android.com/studio/?hl=pt-br.

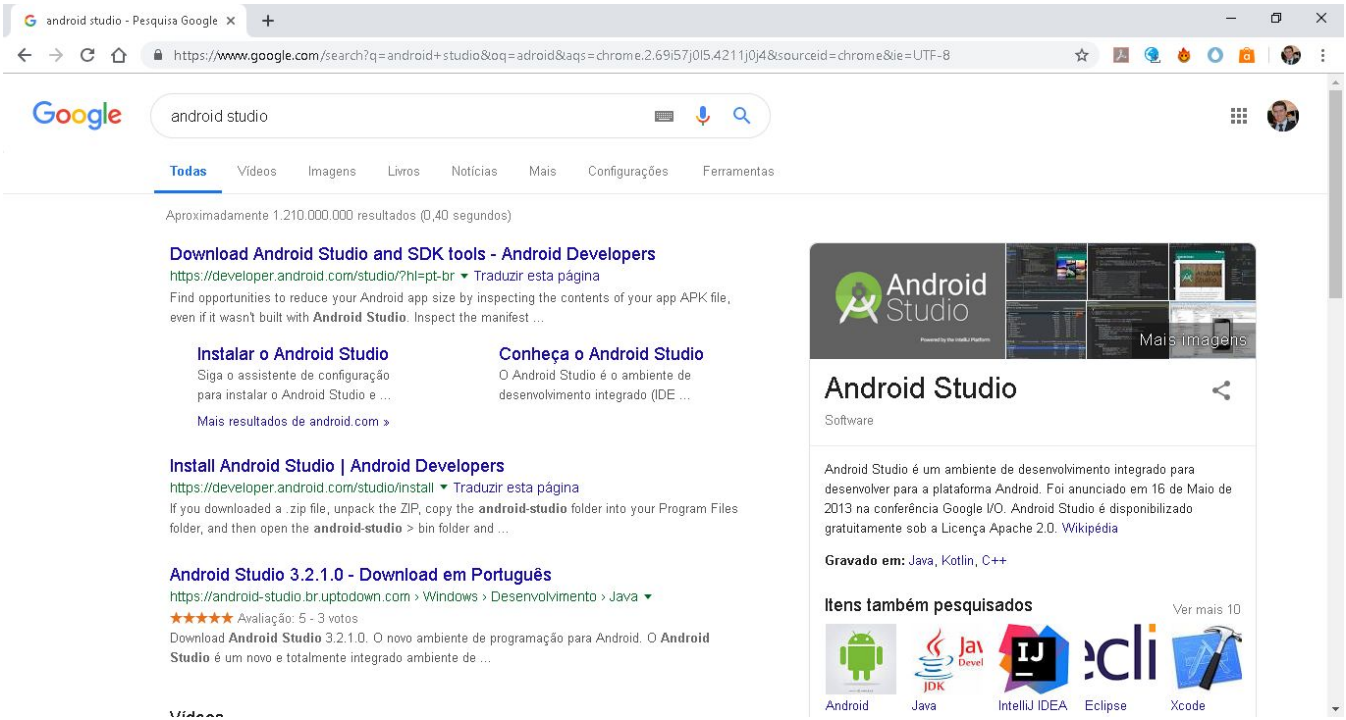


Figura 9: Tela de pesquisa do Android Studio.

Será aberta a página do *Android Studio*, conforme a imagem abaixo:

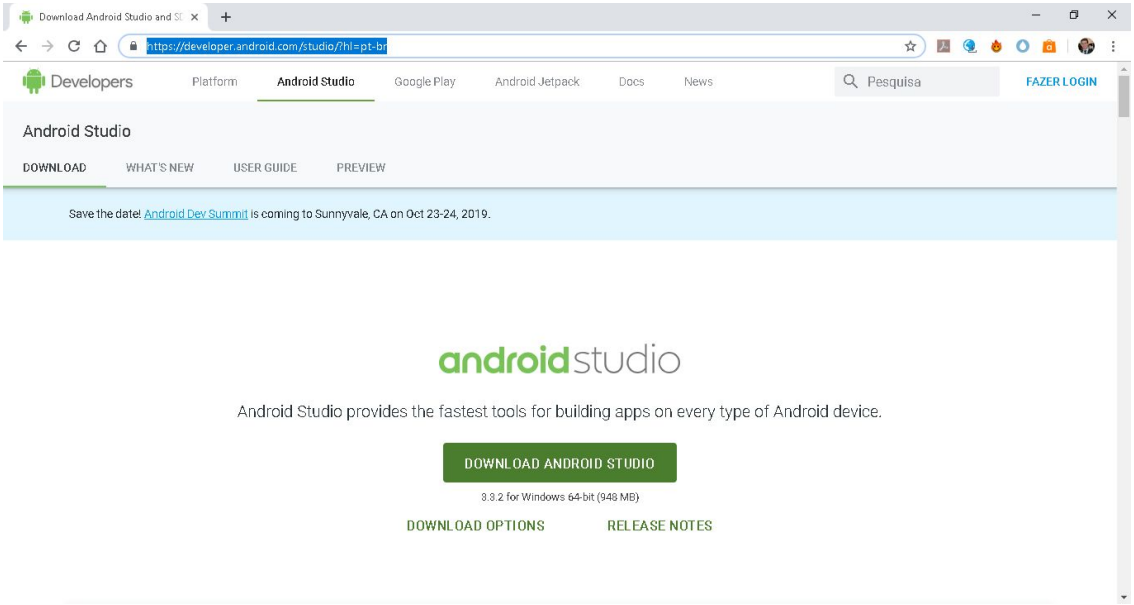


Figura 10: Página do *Android Studio.*

Clique em *download*. Nesse momento, estamos baixando a versão 3.3.2. Então, é preciso concordar com os termos de uso para prosseguir o *download*.

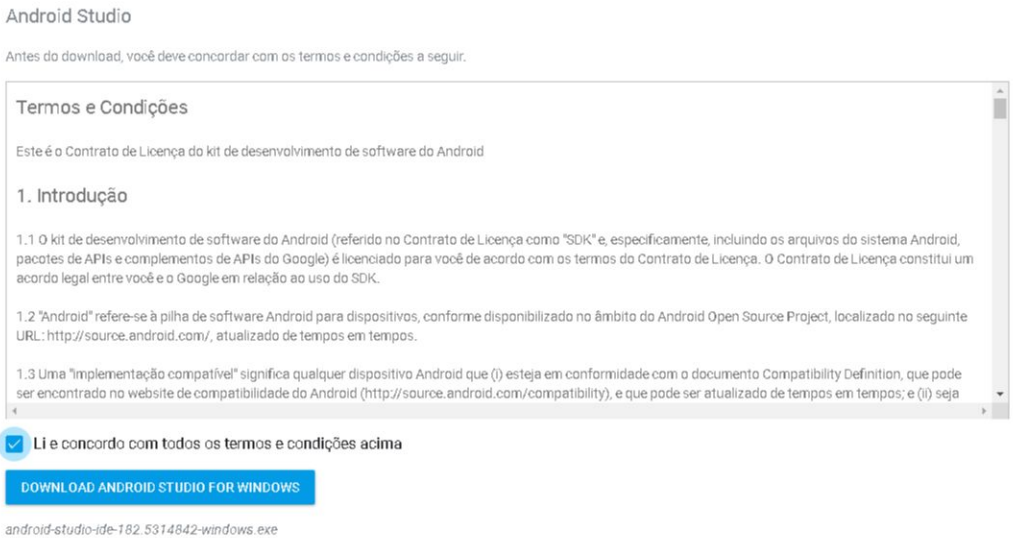


Figura 11: Termo de uso do *Android Studio.*

Libere o *firewall* do Windows, caso seja perguntado, e então inicie a instalação.



Figura 12: Tela inicial da instalação do *Android Studio.*

Há algumas configurações que podem ser feitas na instalação, mas, para efeitos de nosso estudo, apenas faça a instalação padrão “next...next”.

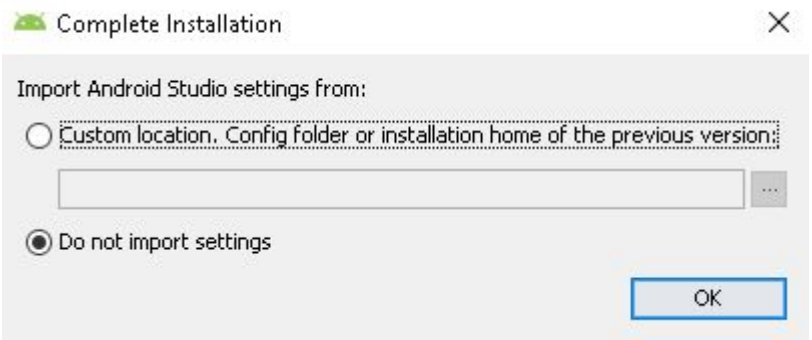


Figura 13: Tela de importação de configurações do *Android Studio.*

Na tela acima, poderíamos importar um arquivo de configurações do Android *Studio*. Em nossa instalação, não importaremos nada. Então, sigamos em frente!

Após a instalação inicial, faremos os ajustes finais da instalação do Android *Studio*. Temos a seguinte tela:

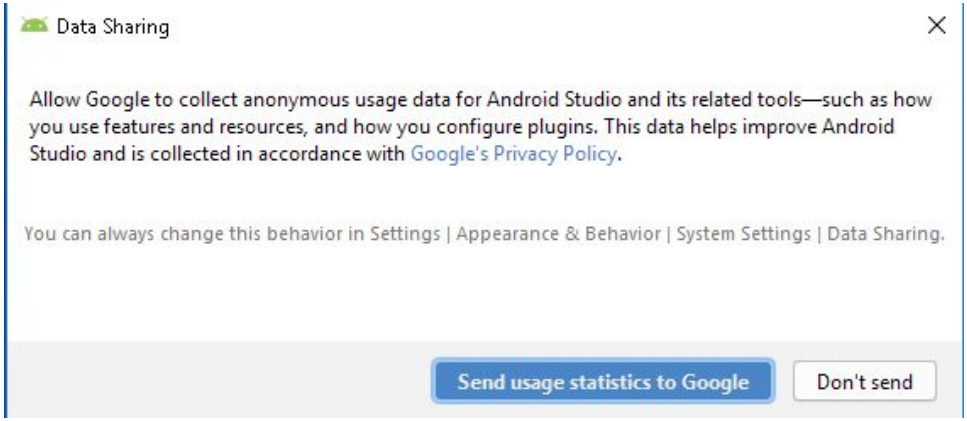


Figura 14: Tela de política de privacidade do Android *Studio.*

Fique à vontade para escolher a opção. Após isso, é aberta a tela do Wizard de instalação do *Android Studio.*

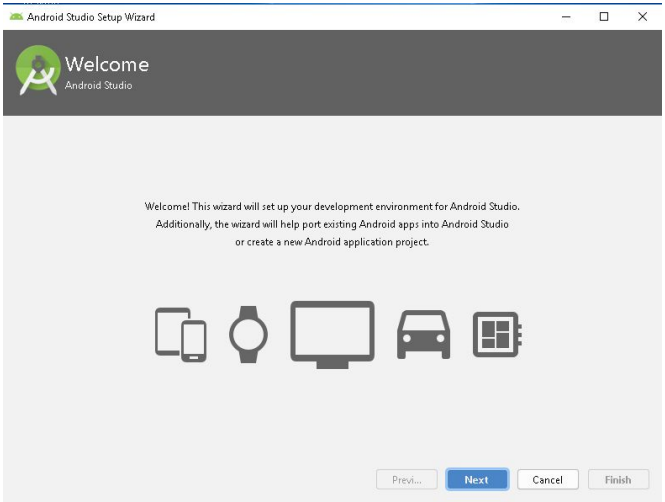


Figura 15: Tela inicial da Wizard de instalação do *Android Studio.*

Agora é necessário escolher o tipo de instalação. Adotaremos o *Standard* (Padrão).

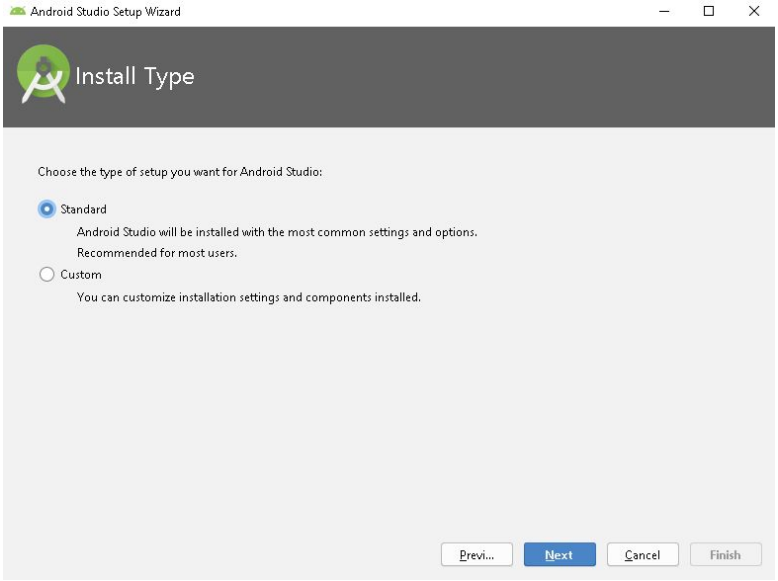


Figura 16: Tela de Seleção do tipo de instalação do Android *Studio.*

Depois será a vez de escolher o tema do *Android Studio.*

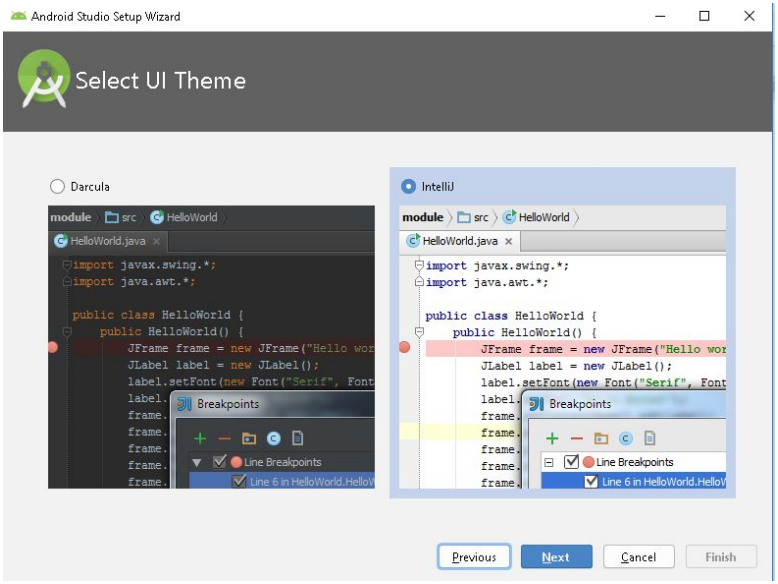


Figura 17: Tela de seleção do tema do *Android Studio.*

É mostrada, então, uma tela das configurações do Android *Studio* para verificação.

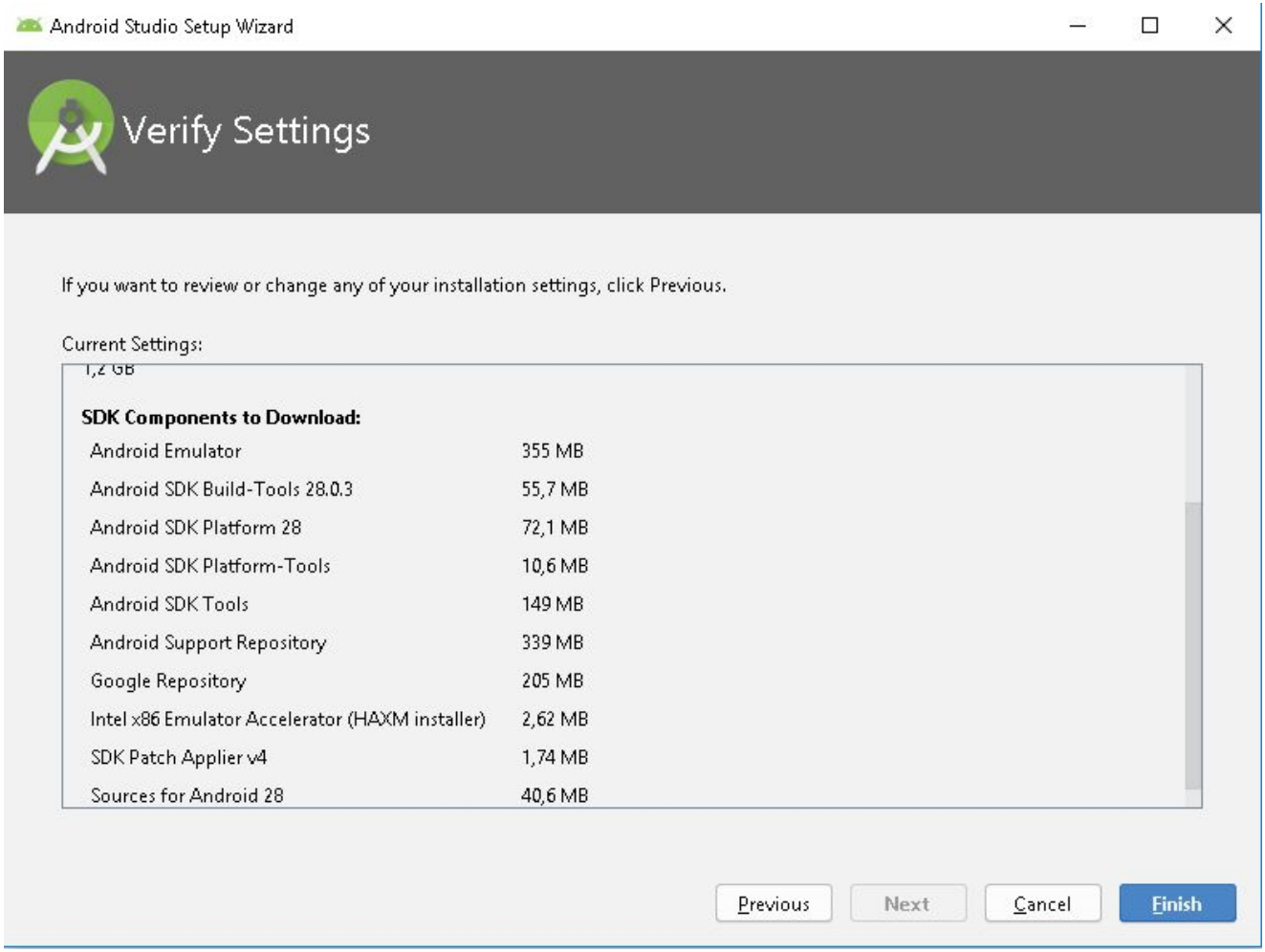


Figura 18: Tela de configuração do *Android Studio.*

Após confirmação, é iniciada a tela para *download* dos recursos configurados.

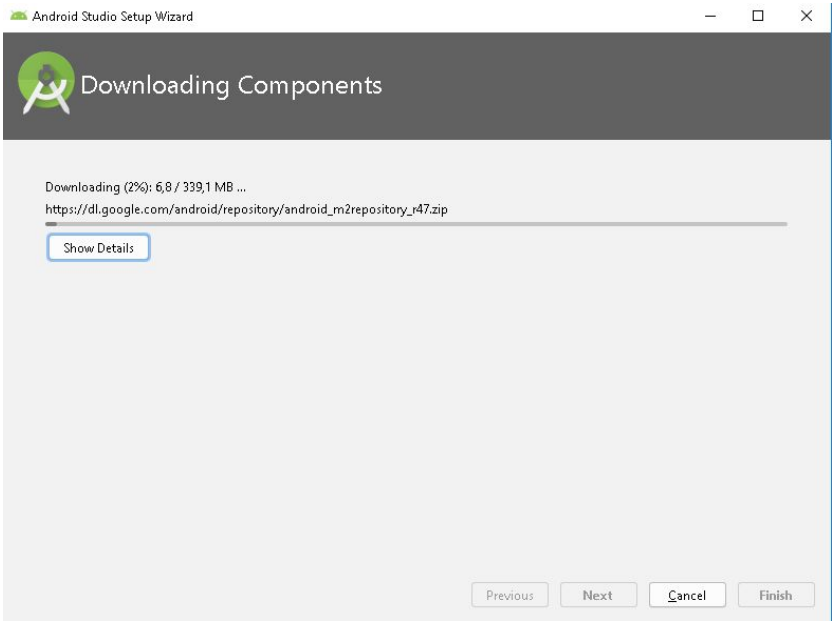


Figura 19: Download de componentes do *Android Studio.*

Agora é só aguardar a finalização. Isso pode demorar um pouco.

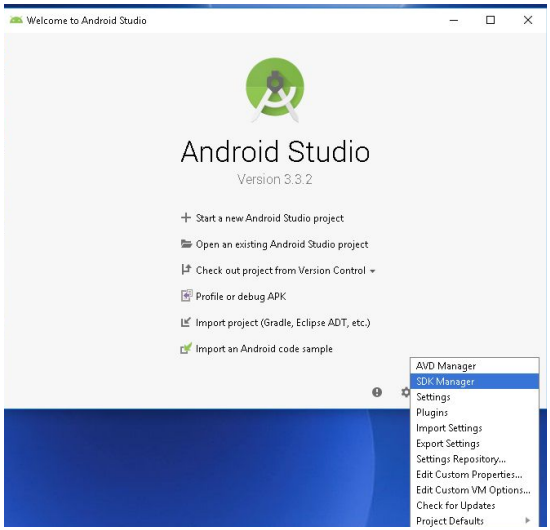


Figura 20: Tela inicial do *Android Studio.* Instalação Concluída!

1.3 EXECUTANDO O PRIMEIRO PROJETO

Neste tópico, criaremos nosso primeiro aplicativo. Para isto, começaremos pelo projeto. Vamos lá?

1.3.1 Criando novo projeto (Hello Word)

Pessoal, para criar um aplicativo, é necessário criar um novo projeto. Abaixo, a tela inicial do *Android Studio*.

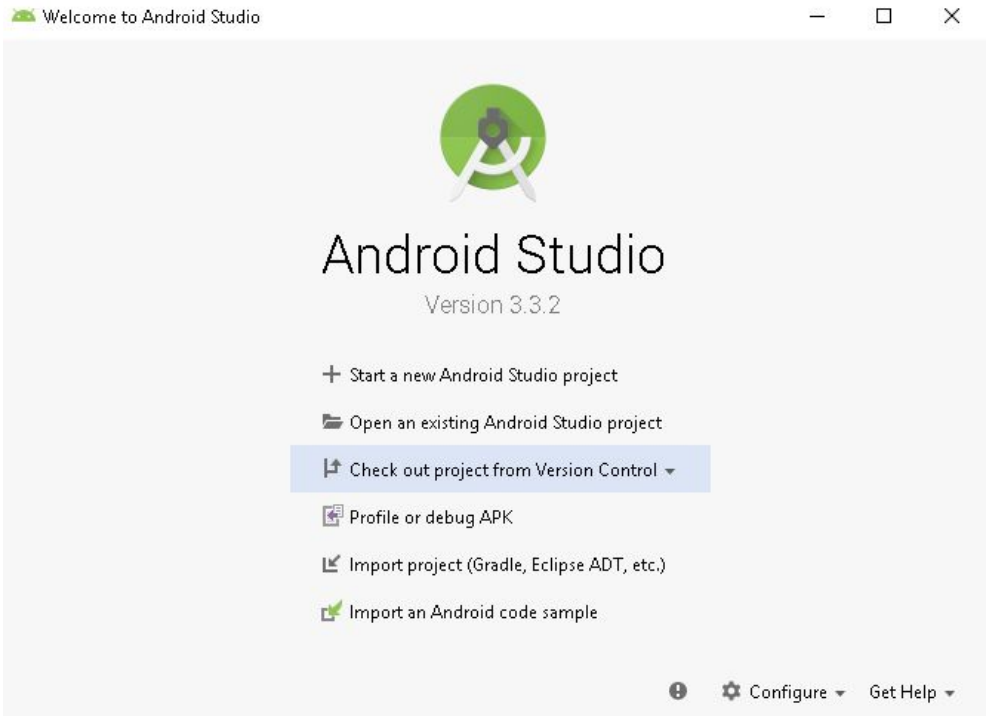


Figura 21: Tela inicial do Androd *Studio.*

Clique em "*Start a new Android Studio Project*" e aparecerá a tela de escolha das configurações básicas do projeto.

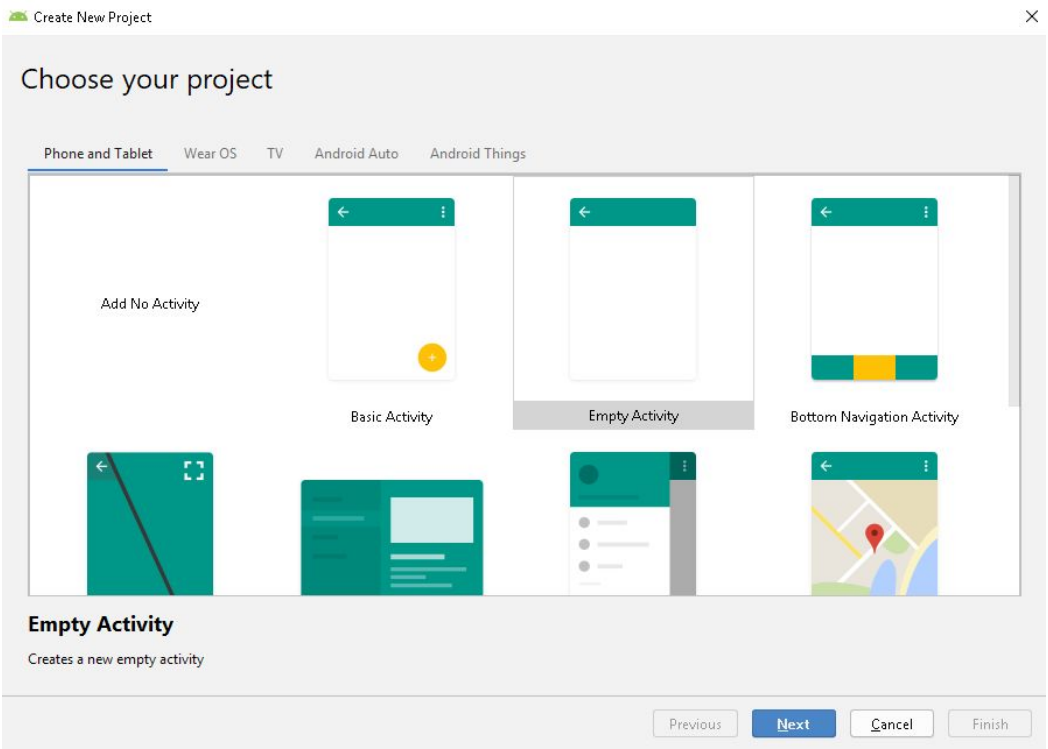


Figura 22: Tela de configurações iniciais do *Android Studio.*

Adotaremos o *layout "Empty Project".*

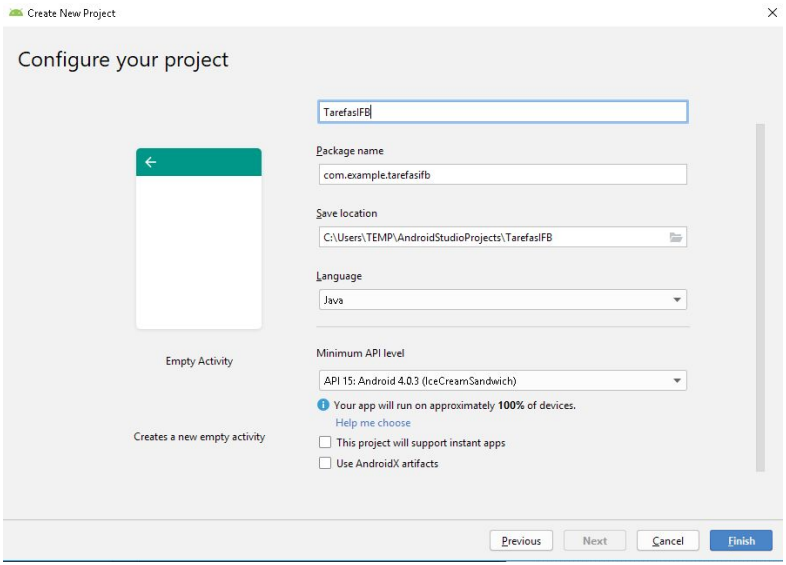


Figura 23: Configurações do projeto.

Clique em finish.

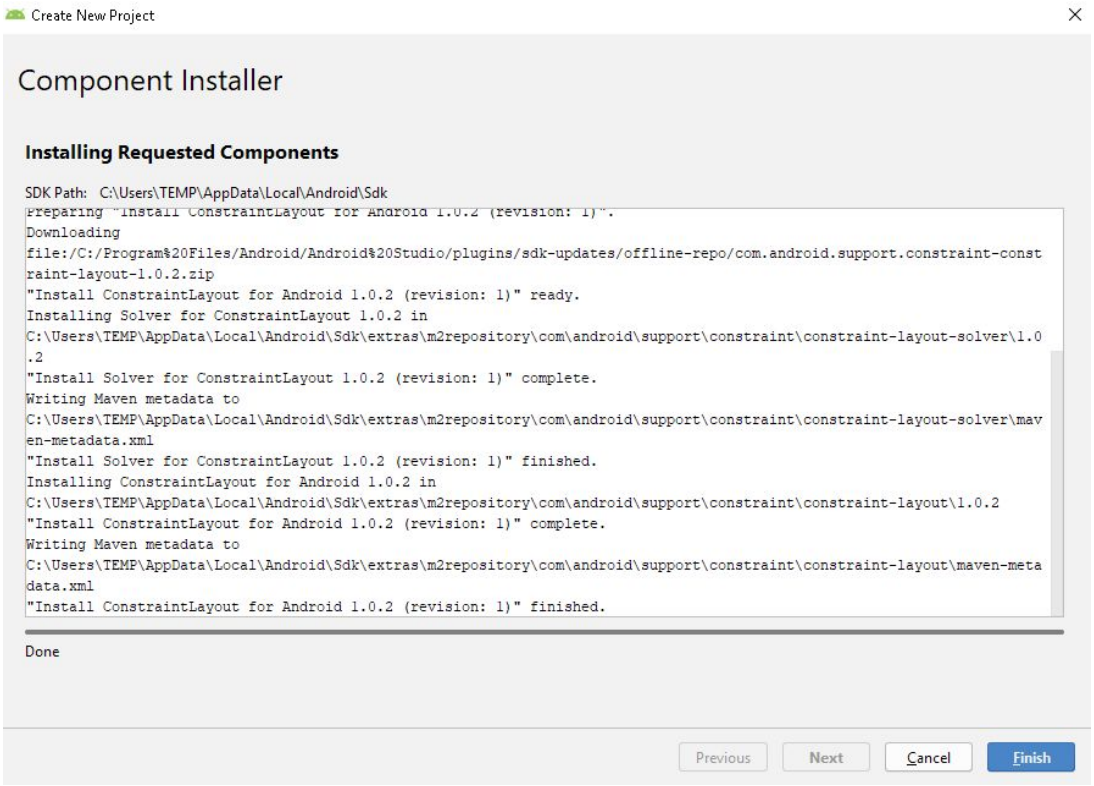


Figura 24: Tela de relatório de instalação,

Caso seja questionado pelo *Firewall* do Windows, faça a liberação:

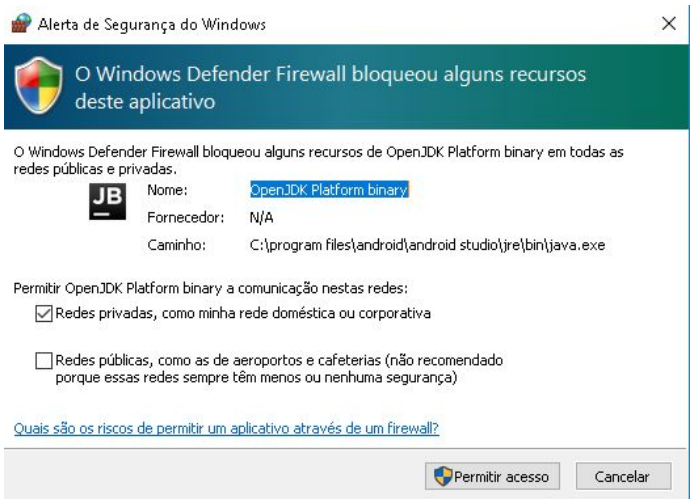


Figura 25: Tela de Liberação do *Firewall*

Aí está nosso projeto criado!

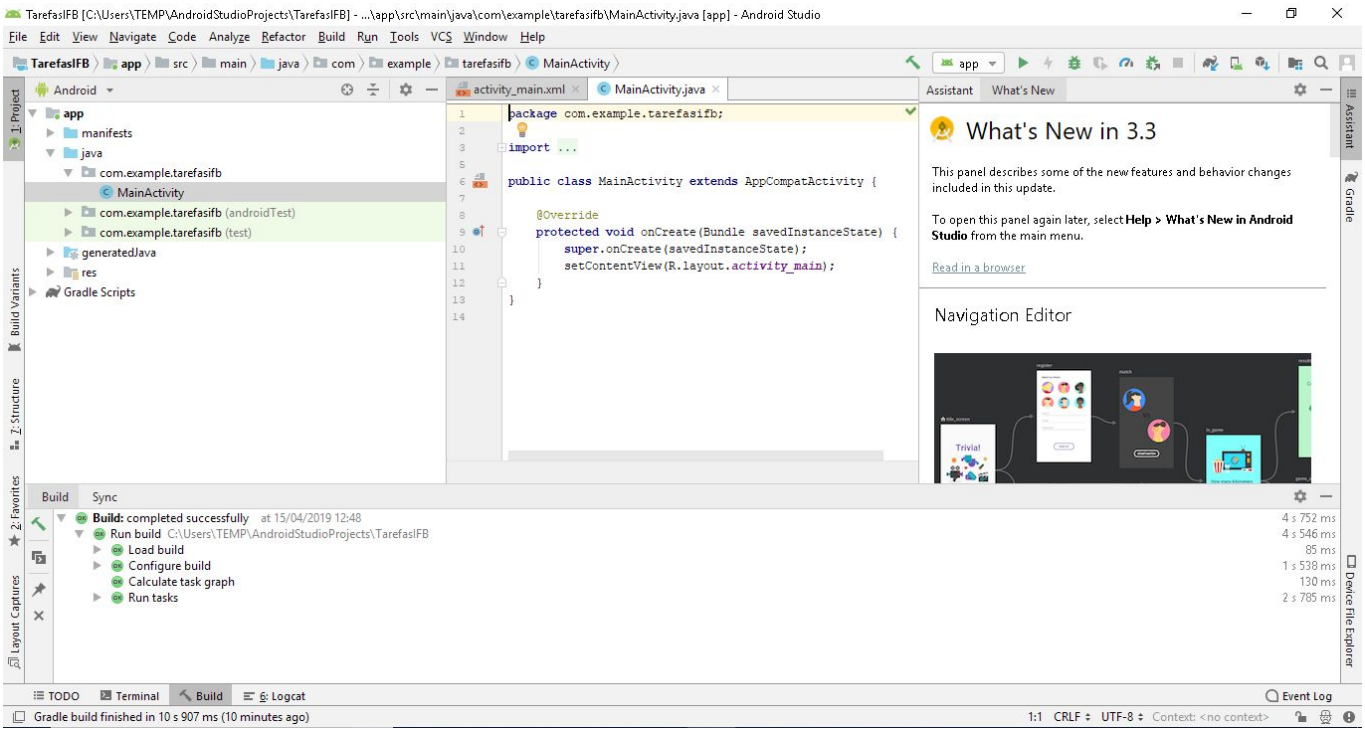


Figura 26: Tela do projeto criado no *Android Studio.*

1.3.2 Configurando o Emulador

Para rodar nosso projeto, vá ao Run > Run App ou Shift + F10.

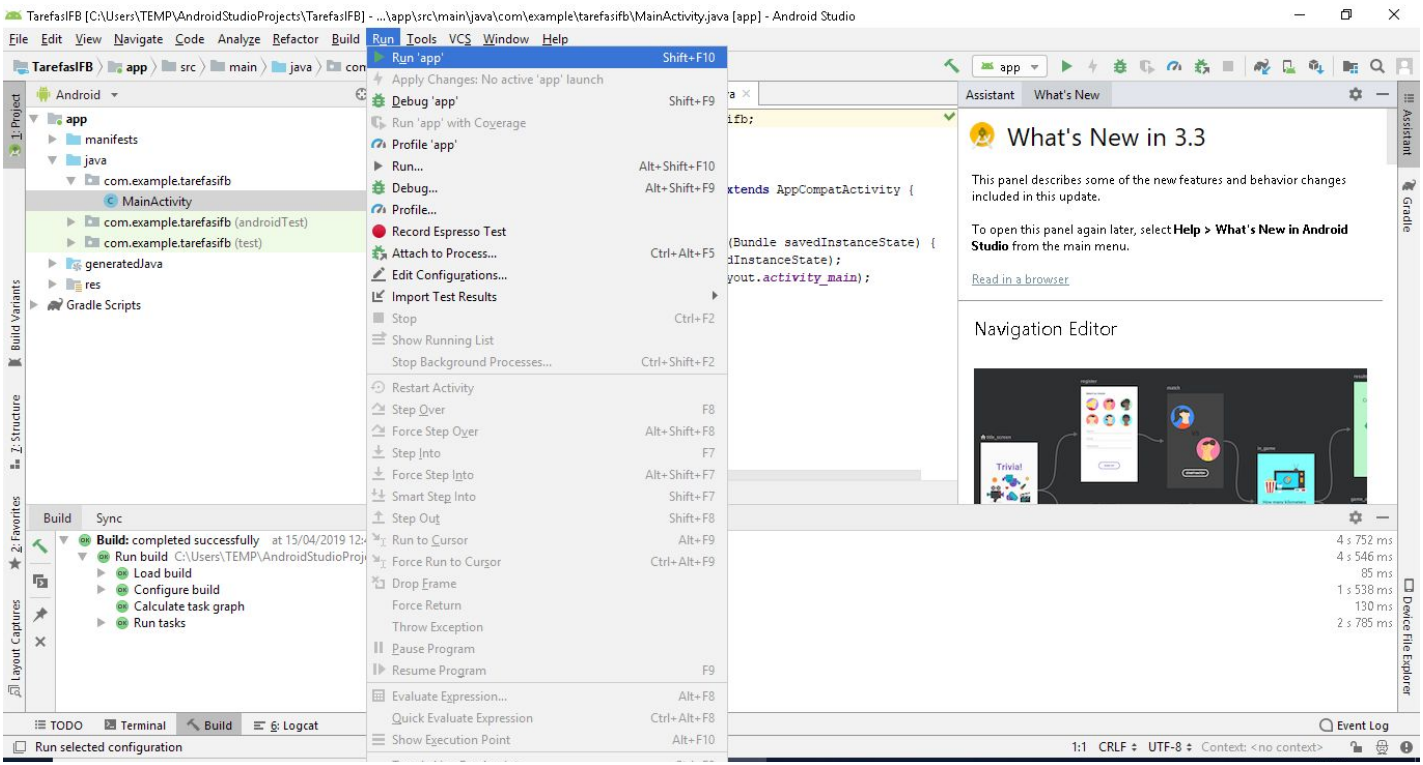


Figura 27: Iniciando execução do projeto.

Ao buscar um dispositivo capaz de executar o APP, o *Android Studio* verifica que, no primeiro momento, não temos nenhum configurado. Precisaremos então configurar o primeiro dispositivo virtual. Para isso, clique em "*Create New Virtual Device*".

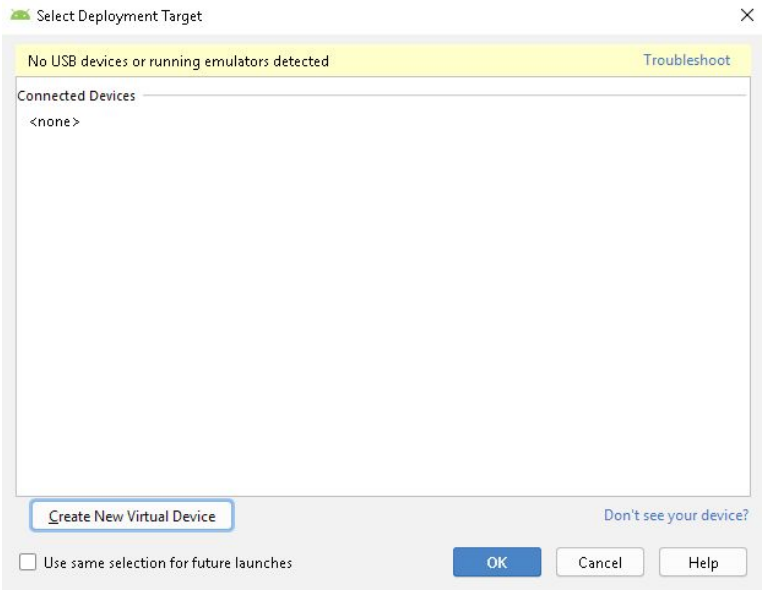


Figura 28: Criando novo dispositivo virtual.

Na próxima tela, uma série de opções de *hardware* virtual são disponibilizadas. Essa escolha deve ser de acordo com a categoria do dispositivo que emulará a aplicação.

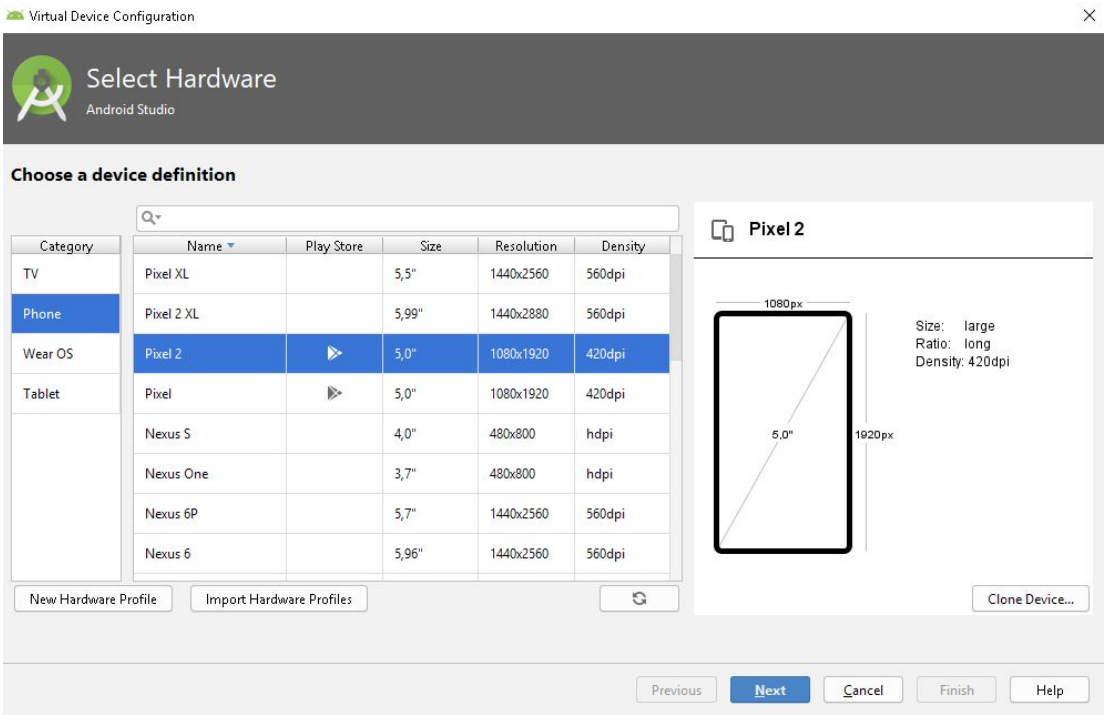


Figura 29: Uma série de opções de hardware virtual.

Fique à vontade para escolher, apesar de que adotaremos, em nosso estudo, o Pixel 2. Em seguida, precisamos indicar qual versão do *Android* que o emulador vai utilizar durante a execução. Para cada versão, é necessária a realização de *download* na primeira vez. Em nosso estudo, utilizaremos a versão *Android Nougat* (*Android* 7). Após escolher a versão desejada, clique em *Next* novamente:

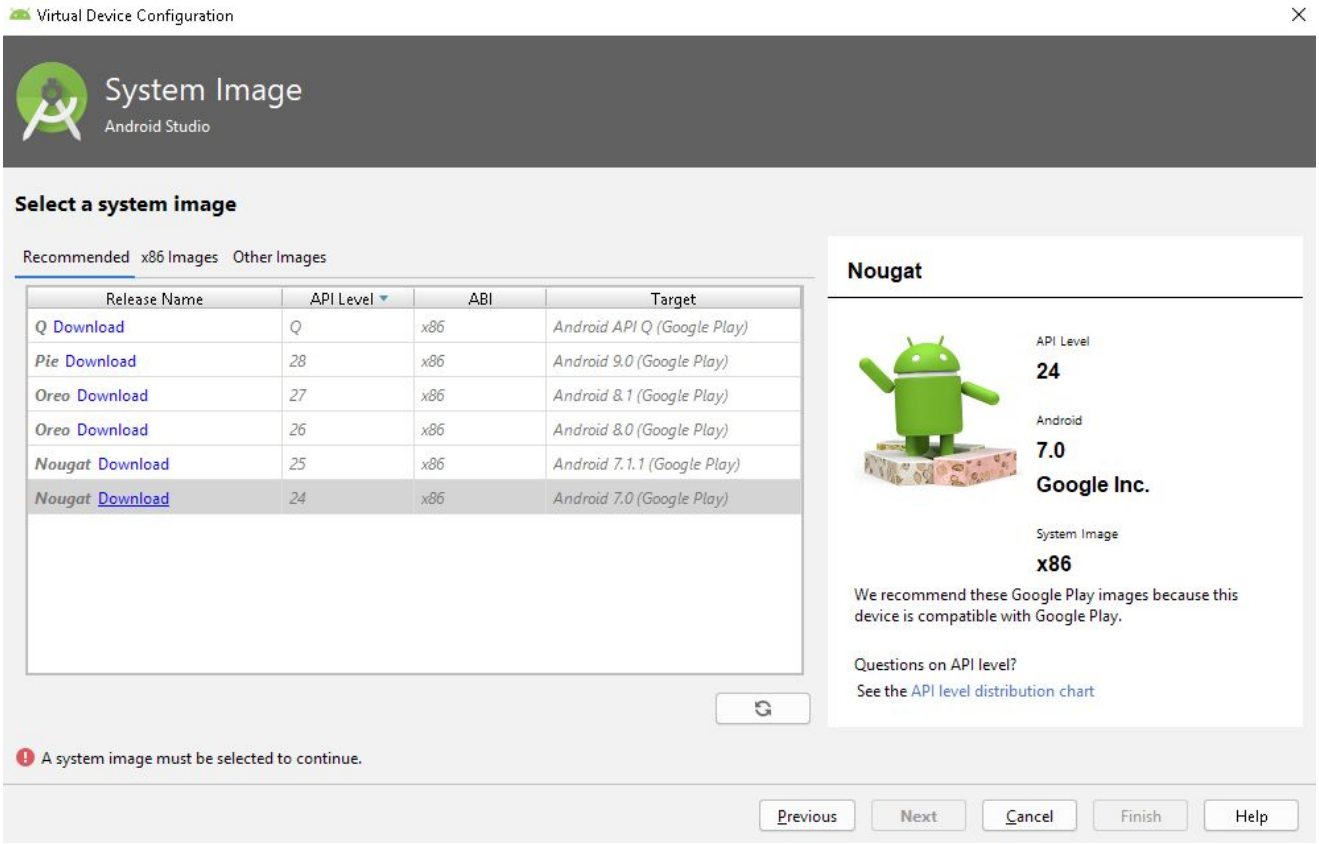


Figura 30: Tela de seleção da versão do Sistema Operacional.

Temos, então, a tela de licenciamento da solução referente ao Sistema Operacional escolhida. Basta aceitar os termos.

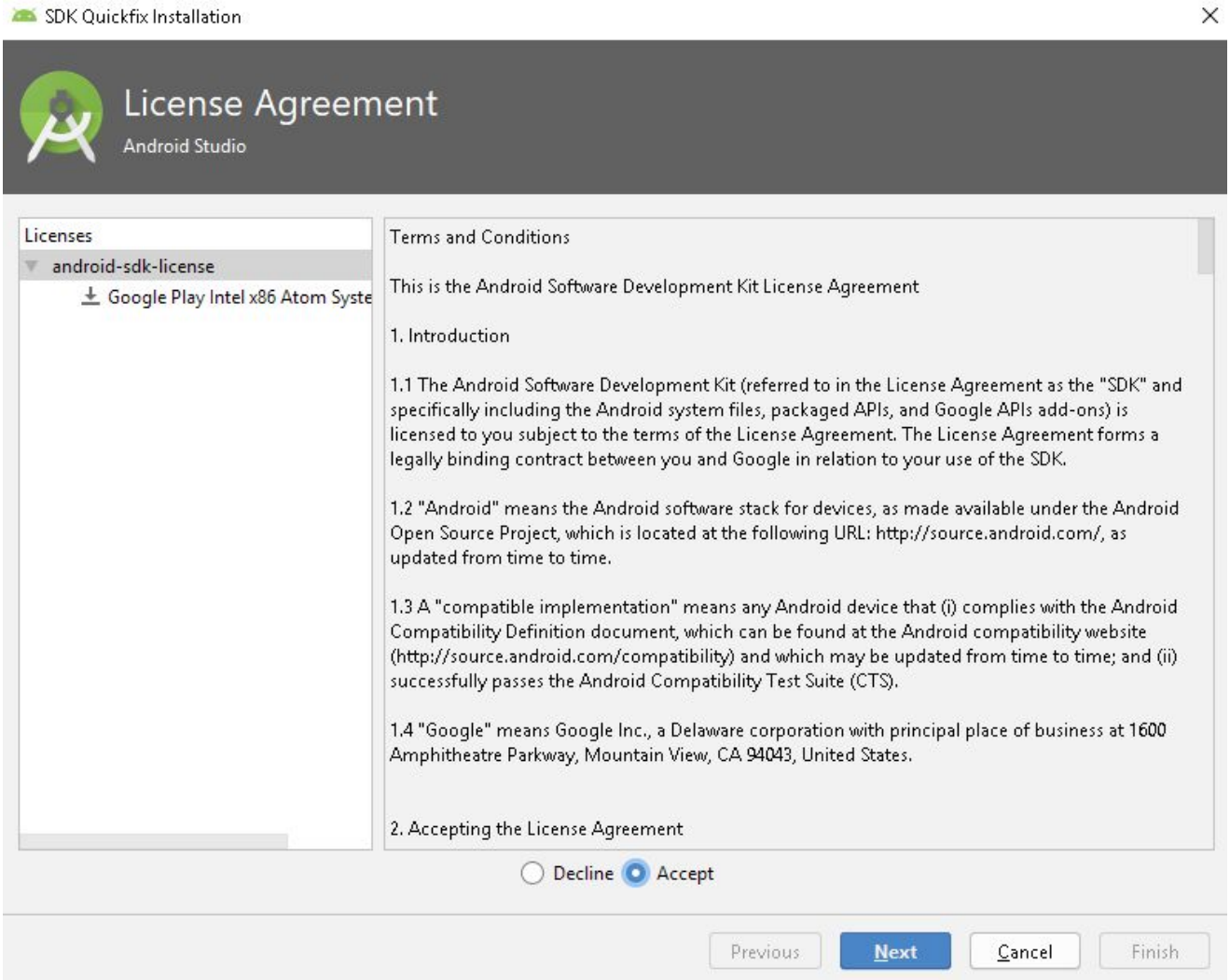


Figura 31: Tela de licenciamento da versão escolhida de Sistema Operacional.

Após isso, é só aguardar o *download* e a instalação da versão.

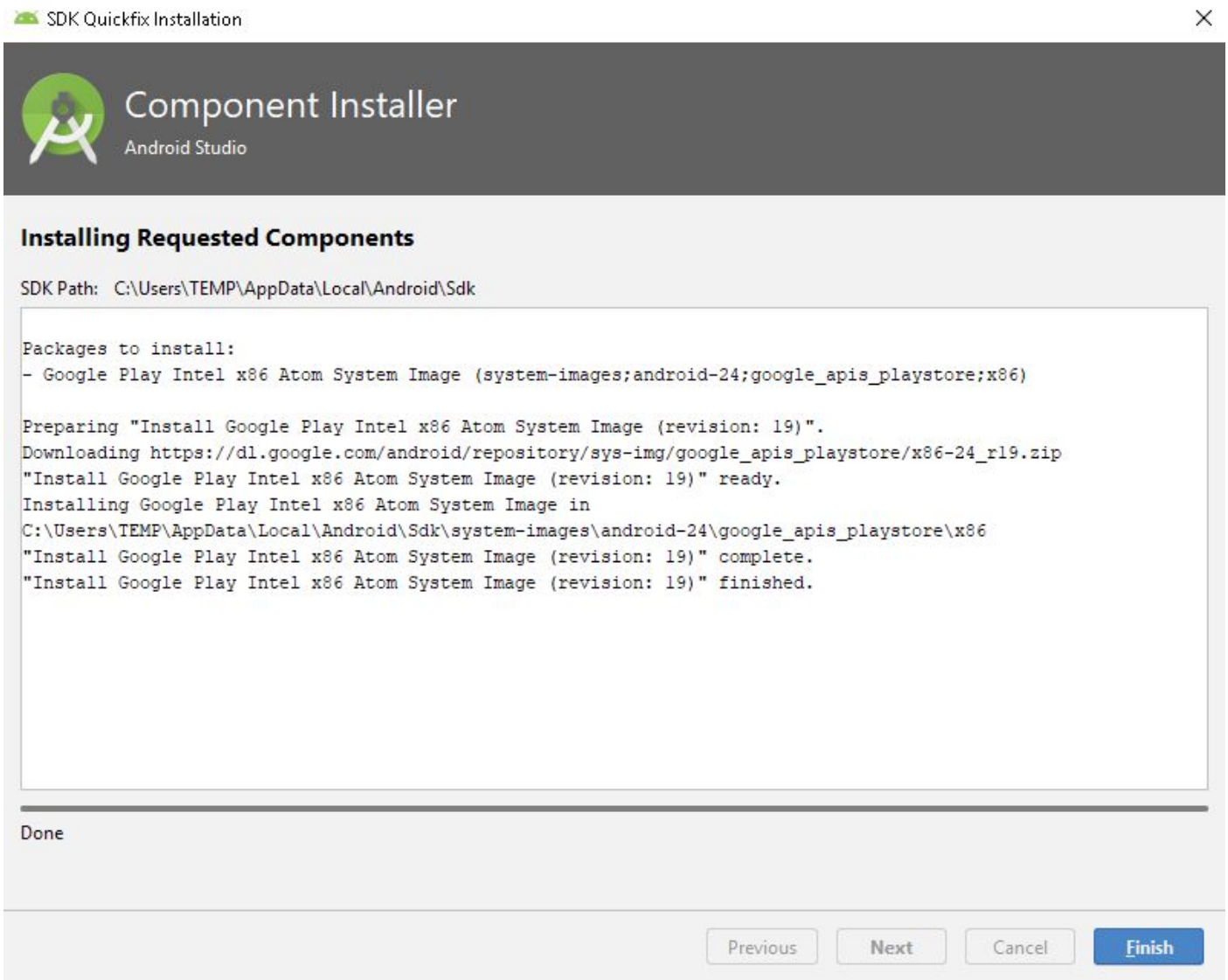


Figura 32: Tela de conclusão da instalação da versão do Sistema *Android*.

Agora a versão do emulador ficará disponível.

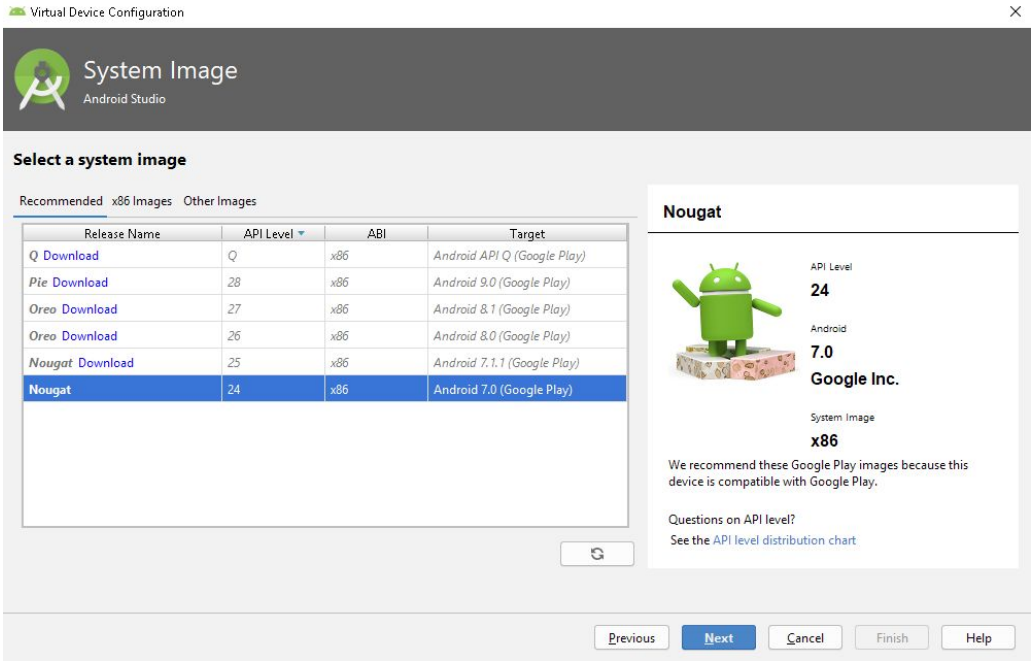


Figura 33: Tela de disponibilidade da versão do Sistema Operacional *Android*.

Após selecionar a versão e clicar em *Next*, entramos na configuração específica do emulador.

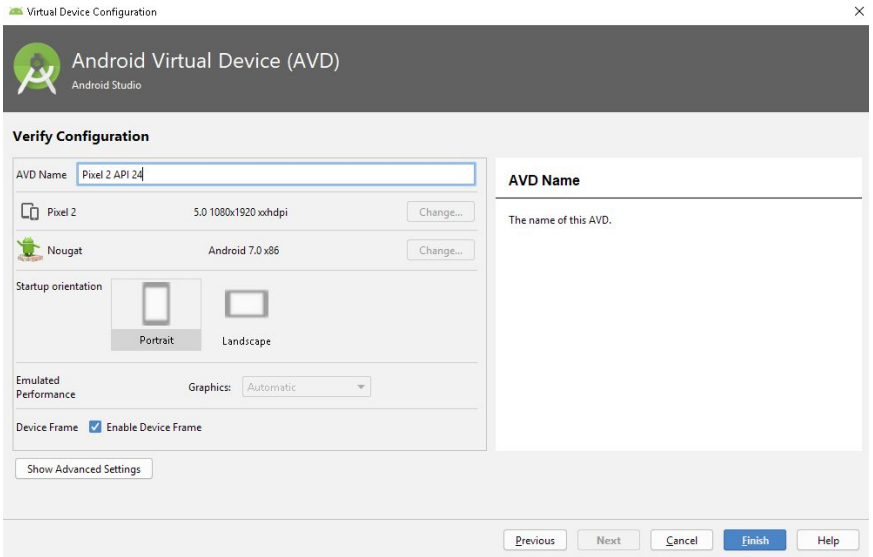


Figura 34: Tela de configurações específicas do Emulador.

Em geral, as configurações padrões atendem bem à nossa expectativa. Clique em *Finish*. Agora, note que o emulador que criamos aparecerá na listagem de dispositivos:

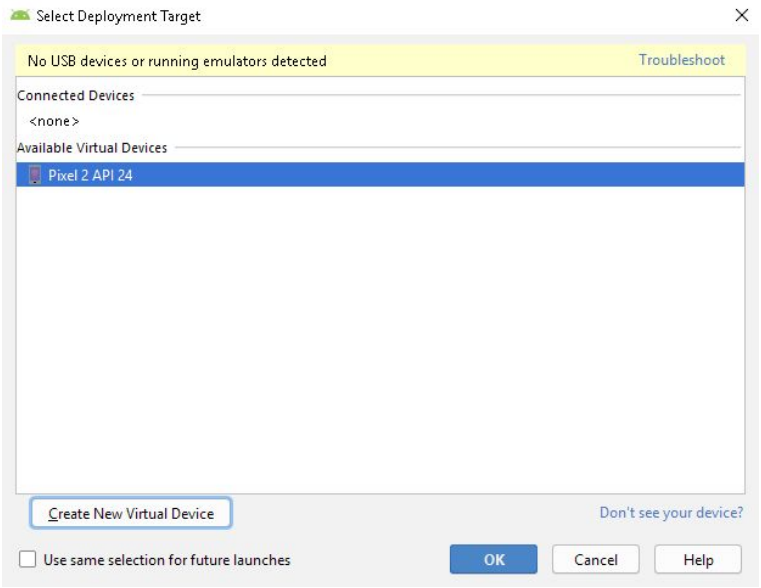


Figura 35: Tela de listagem dos emuladores disponíveis.

Será exibida uma solicitação para instalar o *Android* no emulador criado. Clique em *install and continue*.

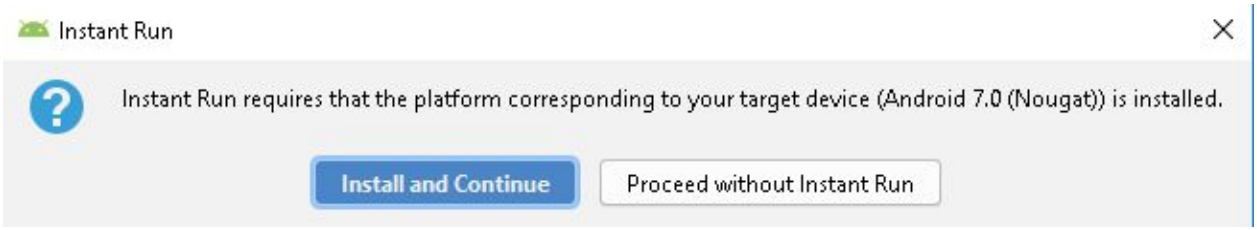


Figura 36: Instalação do Sistema Operacional *Android* no Emulador.

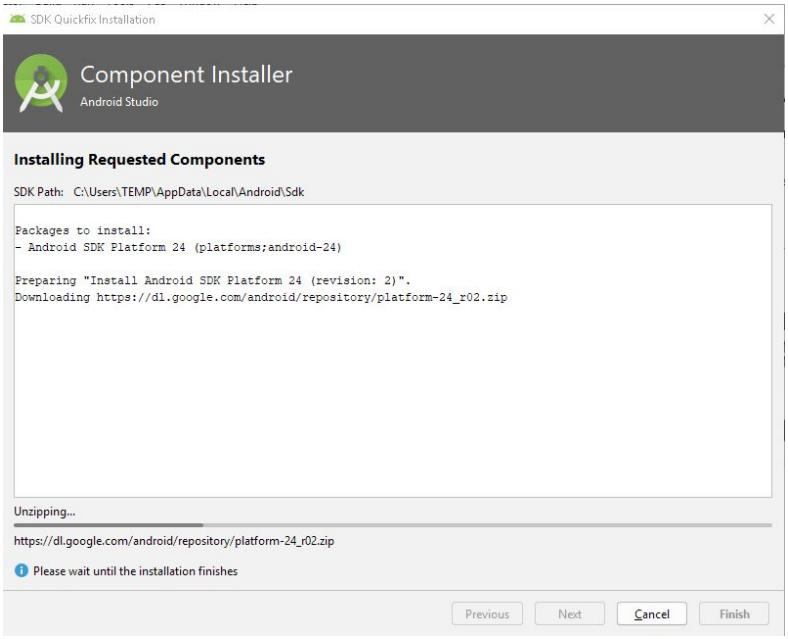


Figura 37-A: Tela de instalação do *Android* no emulador.

A instalação dos componentes do emulador foi iniciada.

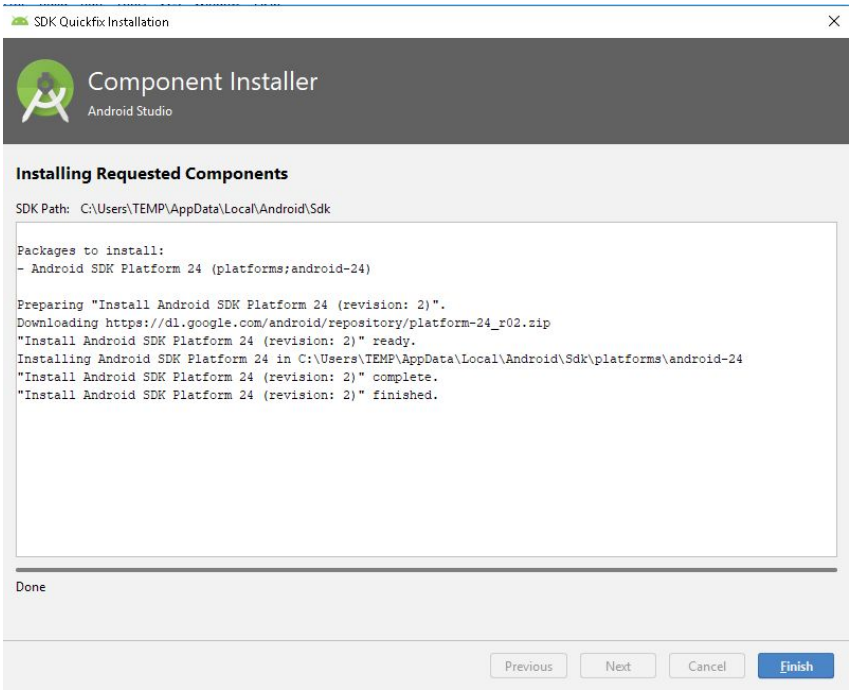


Figura 37-B: Iniciando a instalação.

Eis aí o nosso primeiro *Hello Word*!

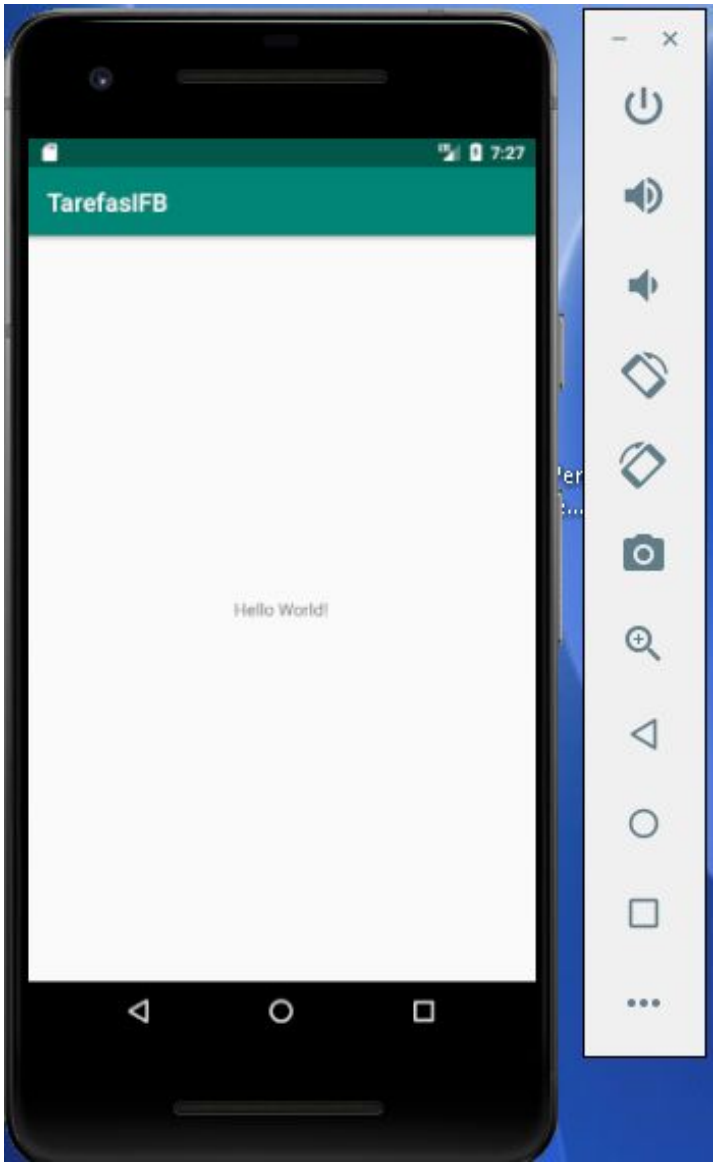


Figura 38: Tela do emulador com Hello Word.

Bem pessoal, chegamos ao final da nossa primeira unidade. Ela abordou aspectos mais históricos e introdutórios com relação à Introdução ao desenvolvimento *Mobile.*

Para sintetizarmos tudo o que estudamos nesta primeira unidade, vamos rever os principais pontos estudados?



Bora rever!

Vimos, no primeiro tópico, as quatro revoluções industriais, sendo a 1ª Revolução marcada pela mecanização da produção; a 2ª Revolução marcada pela energia elétrica e pelo uso de petróleo como combustível; a 3ª Revolução pela questão da Revolução Tecnocientífica e a 4ª Revolução, em que se registra a grande entrada da conectividade, quando temos a Internet das Coisas, grandes avanços em termos de inteligência artificial, dentre outros. Estudamos também os três tipos de consumidores e o impacto disso na concepção de criação de nossos jogos.

No segundo tópico, fizemos a configuração do ambiente, instalando o Java e o *Android Studio*.

Por fim, no terceiro tópico, fizemos nossa configuração da primeira aplicação e nosso *Hello Word*. Configuramos também o nosso emulador virtual.

Na próxima unidade, entraremos em aspectos dos códigos-fontes propriamente ditos. Entenderemos mais detalhes da arquitetura e escreveremos nossa primeira aplicação.

Sigamos em frente!

Unidade 2 PREPARANDO O PROJETO DE APLICATIVOS MÓVEIS

Nesta unidade continuaremos nossa viagem pelo mundo dos aplicativos móveis. Iniciaremos nossas primeiras funcionalidades. Nosso objetivo é entender como funcionam os *layouts*, como podemos manipular conteúdos da tela e como podemos criar métodos para manipulá-los. Antes, veremos alguns conceitos relacionados ao sistema operacional *Android*. Vamos lá!

2.1 ENTENDENDO A ARQUITETURA DA APLICAÇÃO

Neste tópico, vamos entender o que é o *Android*, como ele está estruturado e iremos também criar o nosso primeiro projeto neste sistema.

2.1.1 O que é o *Android*?

Android é o nome do sistema operacional baseado em Linux que opera em celulares *(smarthhones), netbooks, tablets*, TVs e outros dispositivos. É desenvolvido pela *Open Handset Alliance*, uma aliança entre várias empresas, dentre elas, a Google. O funcionamento do *Android* é idêntico ao de outros sistemas operacionais (como Windows, Mac, OS, Ubuntu, etc), nos quais aplicativos e do *hardware* de um computador para



A ideia inicial era ter sido criado para dispositivos com baixo processamento, embora, atualmente, os dispositivos móveis estejam com processamento maior que os computadores antigos. O sistema operacional, atualmente, pode ser encontrado em celulares, tablets, relógios (*SmartWatch*), televisões (*Android*TV) e carros (*Android* auto).

Figura 1: *Android* Auto
Fonte: https://www.tecmundo.com.br/android-auto/105073-android-auto-funcionar-qualquer-veiculo-sistema-multimidia.htm

Segundo a IDC, o Android é o sistema operacional mais utilizado no mundo, conforme pesquisa realizada com dados desde 2016. Os gráficos abaixo fazem uma comparação entre as vendas de dispositivos móveis Android, IOS e outros.

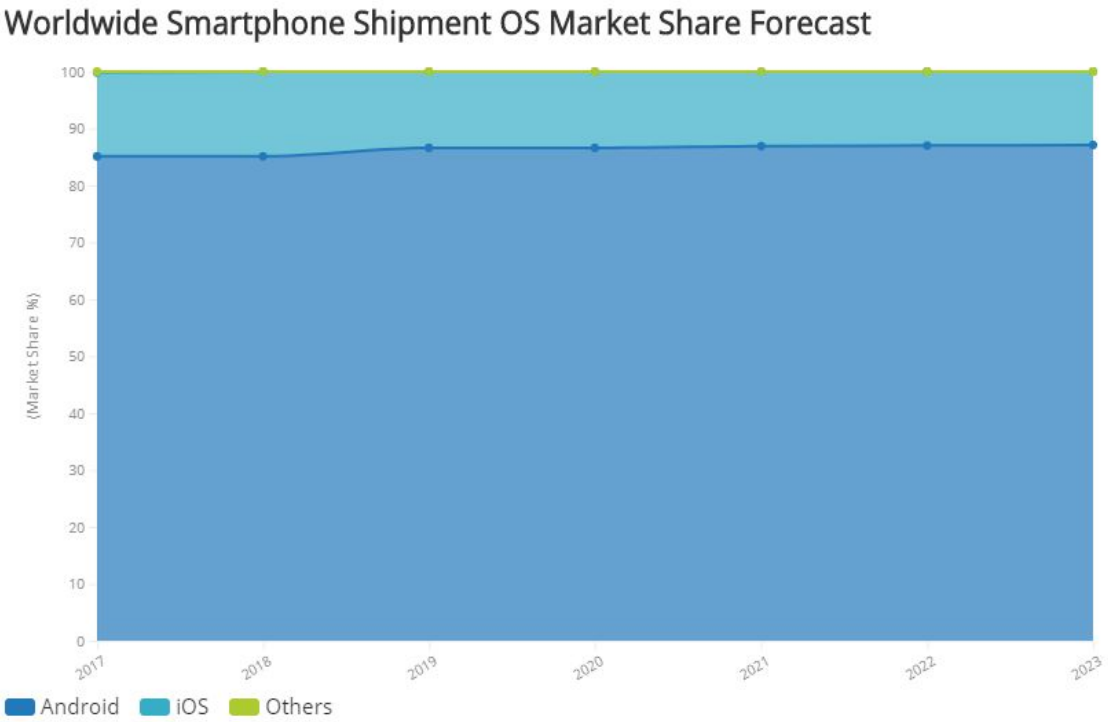


Figura 2: Gráfico comparativo das vendas de *Android* e *IOS*.
Fonte: https://www.idc.com/promo/smartphone-market-share/os

Quarter	2016Q4	2017Q1	2017Q2	2017Q3	2017Q4	2018Q1	2018Q2	2018Q3
Android	81,4%	85,0%	88,0%	87,6%	80,3%	84,3%	87,8%	86,8%
iOS	18,2%	14,7%	11,8%	12,4%	19,6%	15,7%	12,1%	13,2%
Others	0,4%	0,2%	0,2%	0,1%	0,1%	0,0%	0,1%	0,0%
TOTAL	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%

Figura 3: Percentual comparativo das vendas de *Android* e *IOS* pelo mundo.
Fonte: https://www.idc.com/promo/smartphone-market-share/os

– Nossa, professor! Não sabia dessa diferença toda!

– Pois é! O IOS ainda tem uma venda forte entre os tablets, por conta do IPAD.

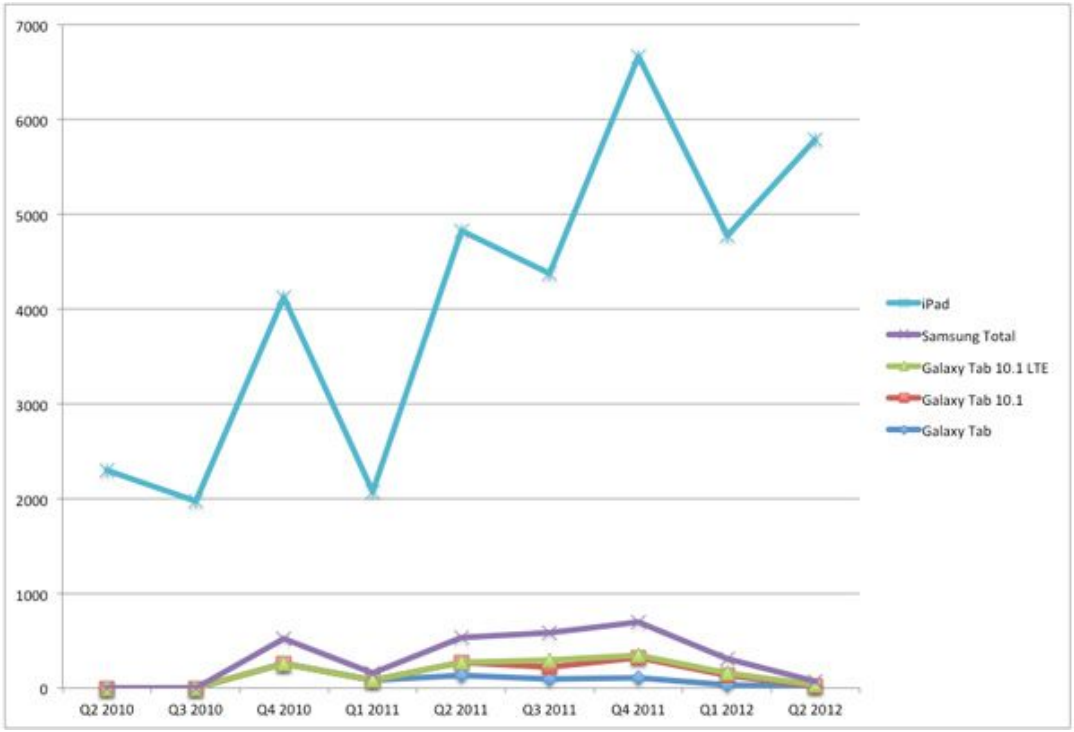


Figura 4: Gráfico de venda de Tablets.
Fonte: https://tecnoblog.net/109687/samsung-apple-processo-patentes/

– Então, pessoal! Vamos entrar em aspectos técnicos de nossos aplicativos?

– ***Vamos, professor! Queremos colocar a mão na massa!***

– Vamos em frente!!!

2.1.2 Estrutura do Projeto Android

Nessa seção, vamos estudar a estrutura básica de um projeto Android. Vejamos essa estrutura em nosso projeto *TarefasIFB*.

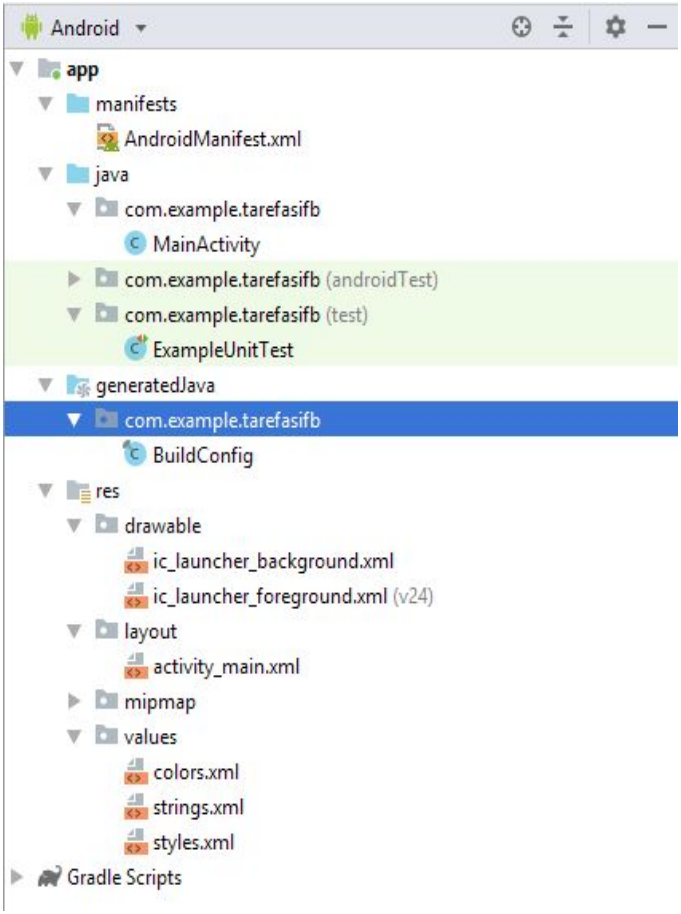


Figura 5: Estrutura do projeto TarefasIFB

– Conforme imagem ao lado, vamos analisar cada parte do projeto, OK?
– ***OK, professor!***

A primeira parte que veremos é a do diretório dos manifestos *(manifests)*, na qual temos o arquivo AndroidManifest.xml. Esse arquivo é obrigatório para uma aplicação *Android*. Trata-se de um arquivo que contém informações essenciais sobre a aplicação, como: nome de pacote do aplicativo, versão, nome, classes utilizadas. Vejamos, a seguir, um exemplo desse arquivo.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.tarefasifb">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Temos também a pasta java, onde são armazenados os códigos-fontes javas de nossa aplicação. Abaixo temos o exemplo do aquivo MainActivity.java

```
package com.example.tarefasifb;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

    public class MainActivity extends AppCompatActivity {

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

Temos também a pasta “res”, onde armazenam-se os recursos da aplicação como arquivos de *layout*, imagens, animações e xml contendo valores de *string, arrays*, dentre outros.

Abaixo, temos um exemplo do arquivo *string.xml* de nossa aplicação:

```
    <resources>
  <string name="app_name">TarefasIFB</string>
</resources>
```

Ainda temos o diretório *assets* e *libs*. O *assets* é o diretório para armazenamento de arquivos diversos utilizados na aplicação. Já a pasta *Lib* armazena as bibliotecas da aplicação.

2.1.3 Criando primeiro projeto

Para criar um novo projeto, vamos ao Menu *File > New > New Project*.

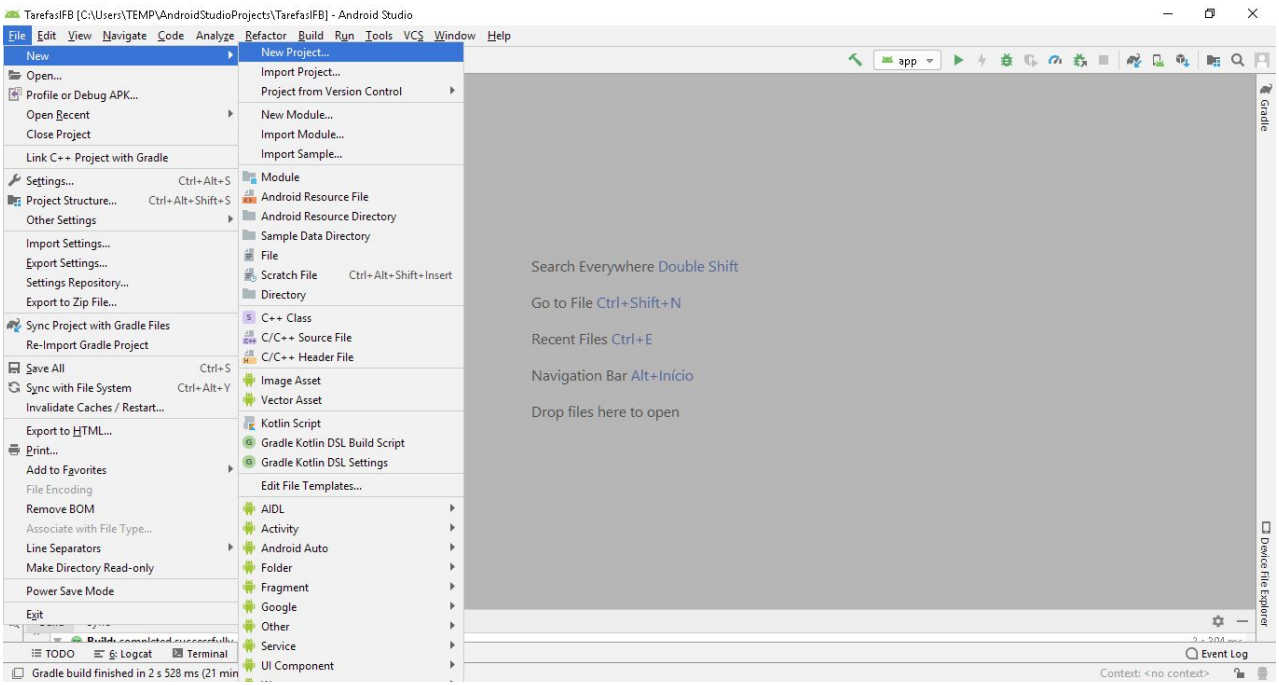


Figura 6: Criando um novo projeto no *Android Studio*

Feito isso, aparecerá a tela para escolha de dados do projeto como se o desenvolvimento fosse para celular e tablet, TV, *Android* Auto, entre outros.

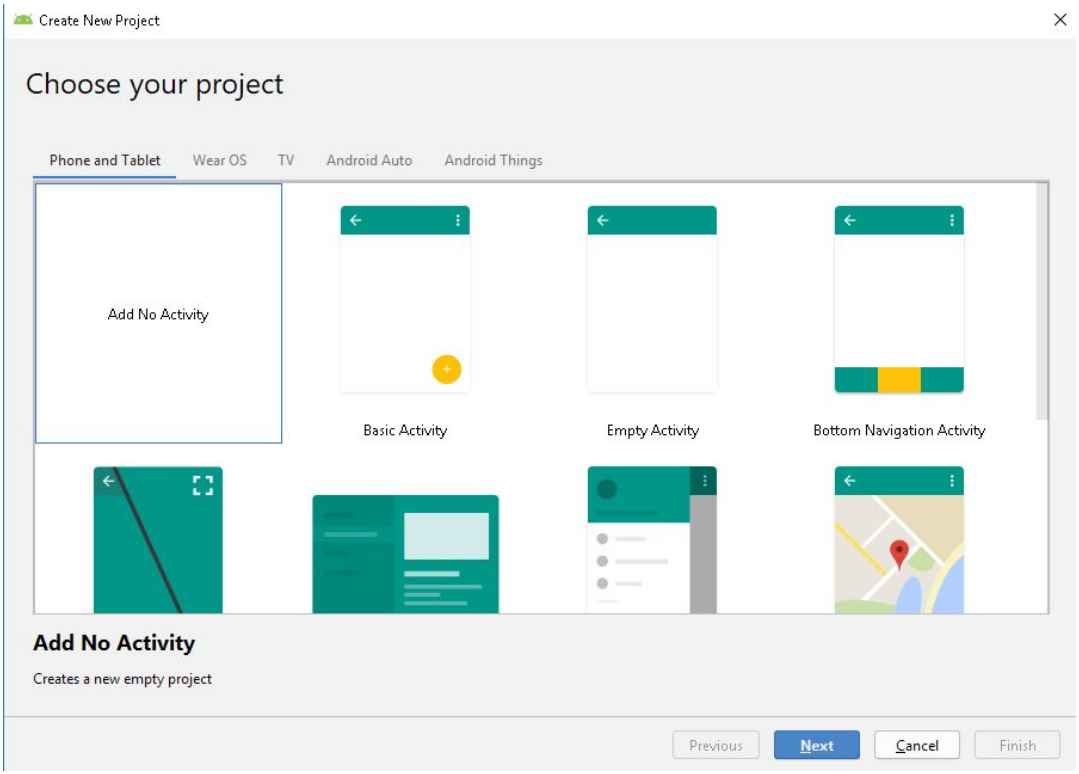


Figura 7: Selecionando o layout padrão do projeto.

Manteremos na opção de *Phone and Tablet* e escolheremos a opção *Add No Activity*, que significa que não adotaremos um *layout* padrão, mas faremos o nosso.

Após isso, iremos para uma tela de configurações de nome, pacotes e compatibilidade do projeto. Configure conforme abaixo:

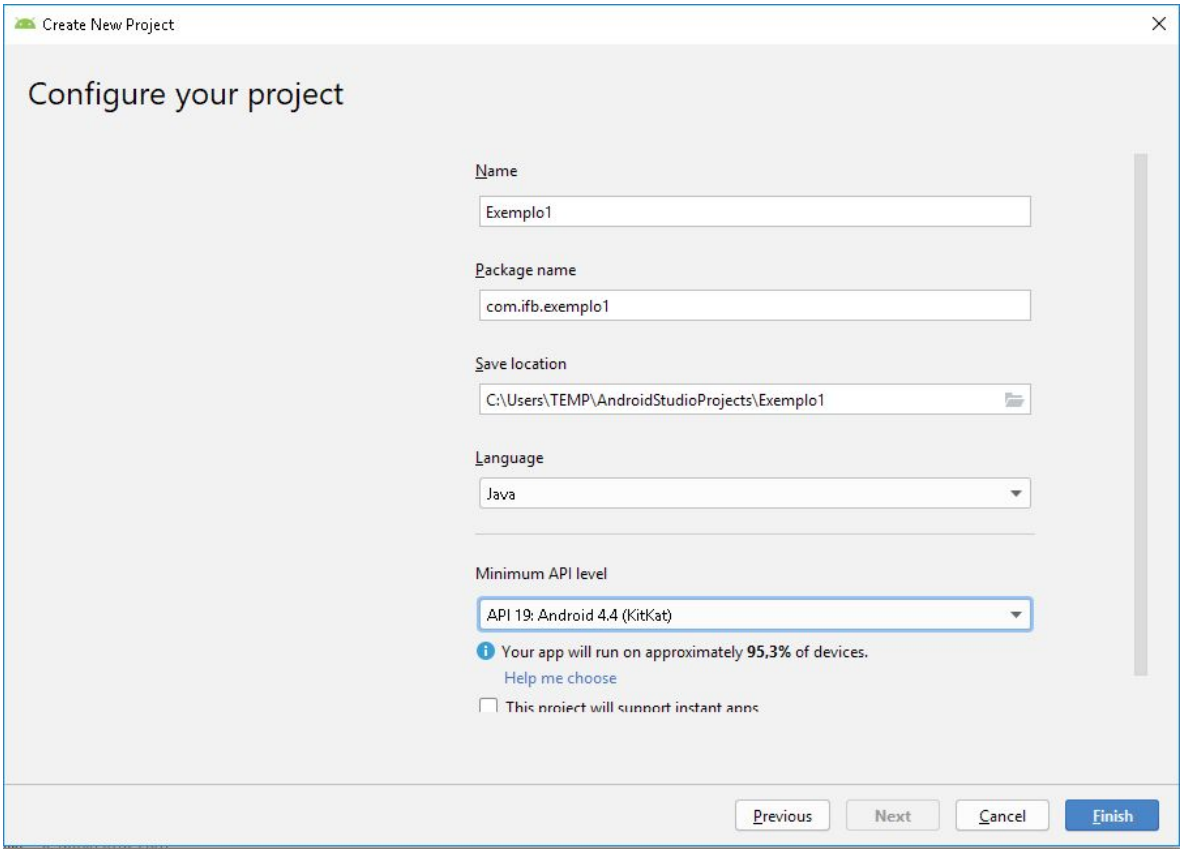


Figura 8: Configurando o projeto.

Clique em *Finish* e nosso projeto terá sido criado.

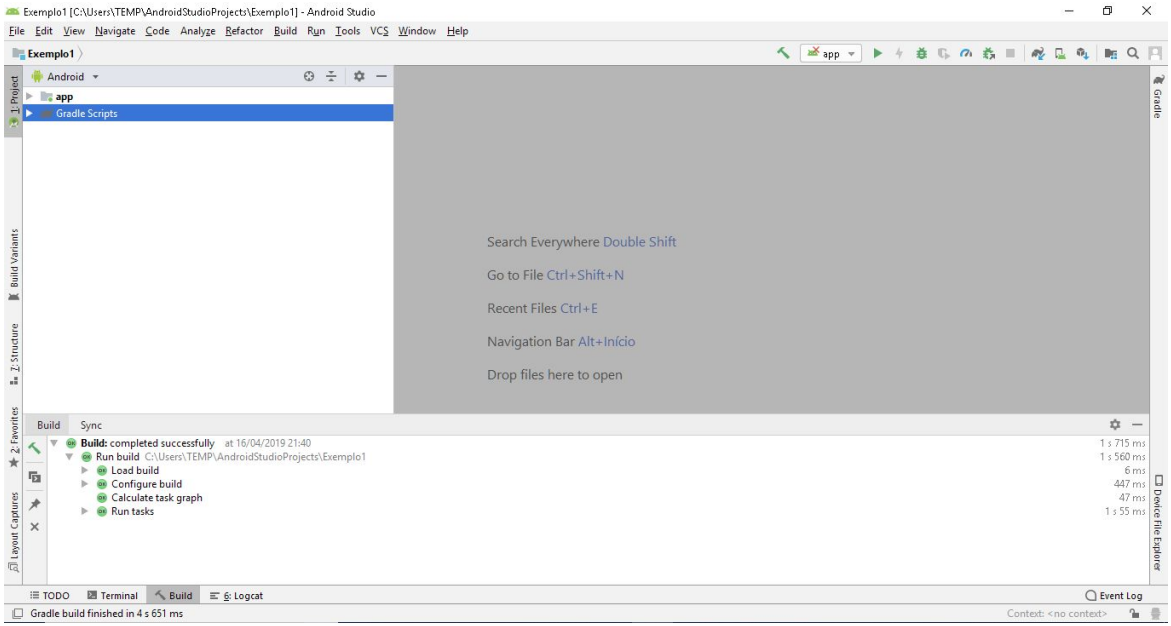


Figura 9: Estrutura do projeto criado.

E agora vejamos a estrutura de nosso projeto no *Windows Explorer*.

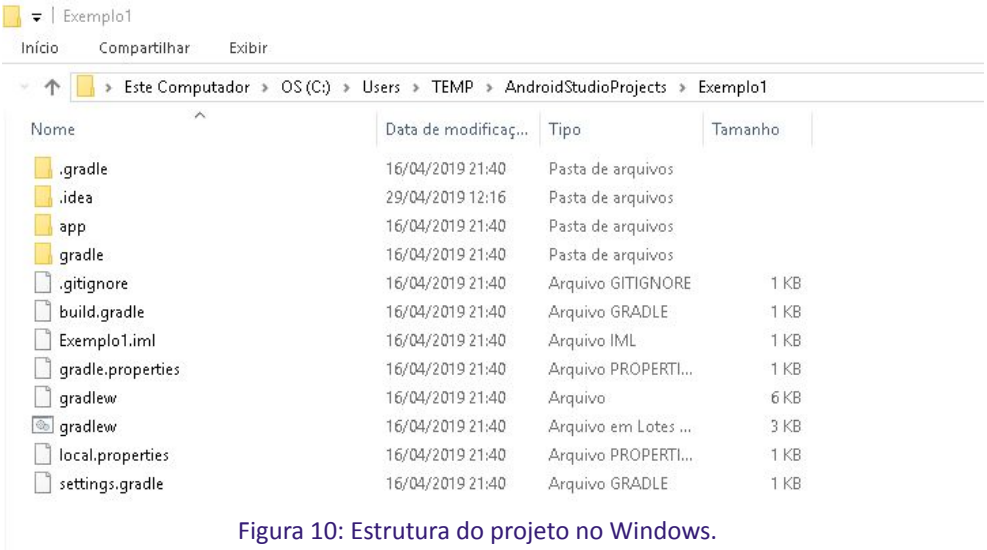


Figura 10: Estrutura do projeto no Windows.

Com isso, podemos copiar toda a estrutura do projeto e gerar novos projetos.

2.2 CRIANDO A PRIMEIRA APLICAÇÃO

Após a criação do projeto no *Android*, vamos estruturar o seu layout e inserir os objetos que irão compor o nosso aplicativo.

2.2.1 Criando a estrutura de *layout*

Vamos iniciar nosso desenvolvimento pelo *layout* de nossa aplicação. O *layout* é chamado de *activity*. Na verdade, podemos ter uma sequência de *activities*. Iniciaremos mostrando uma mensagem. Criaremos nosso arquivo na pasta: res/layout/activity_main.xml.

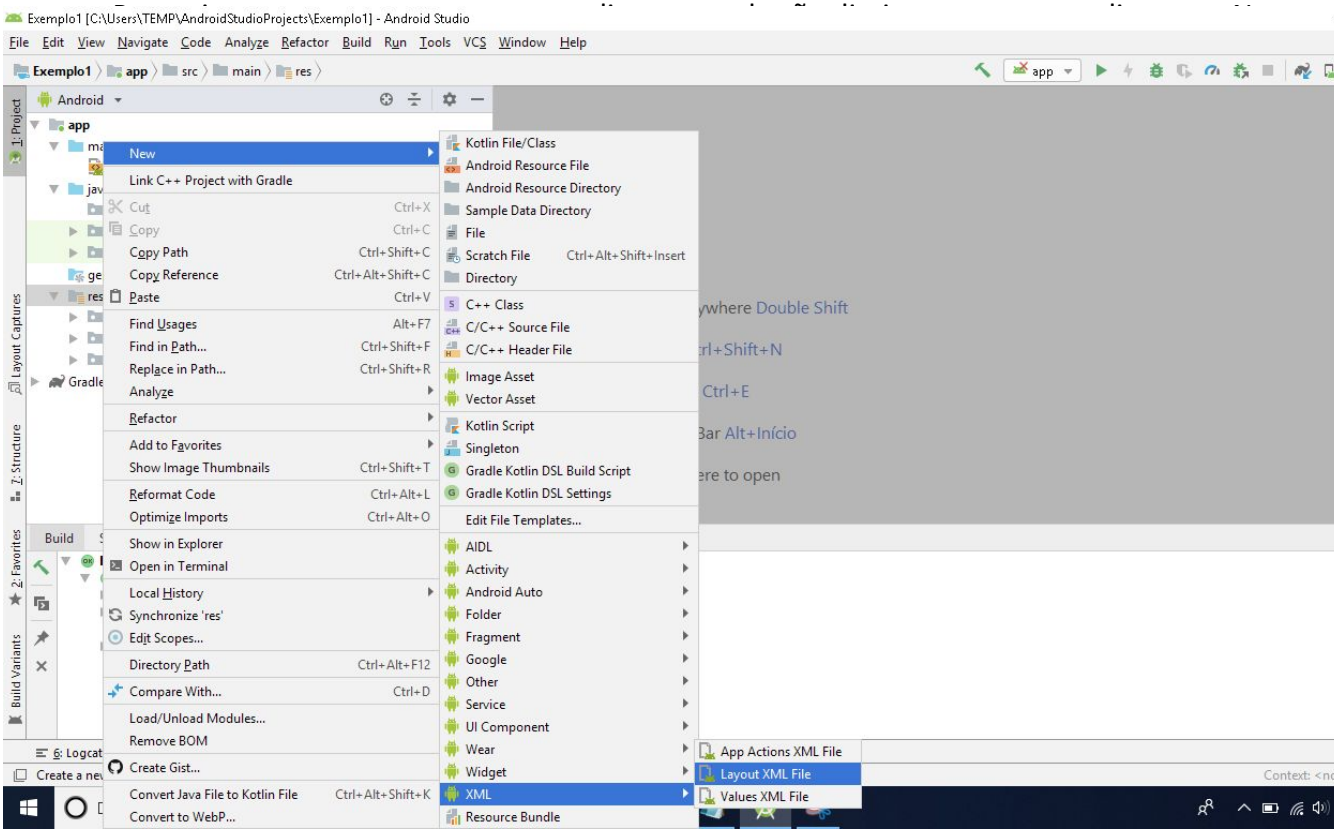


Figura 11: Criando estrutura de Layout.

O nome do arquivo deve ser: activity_main, conforme abaixo:

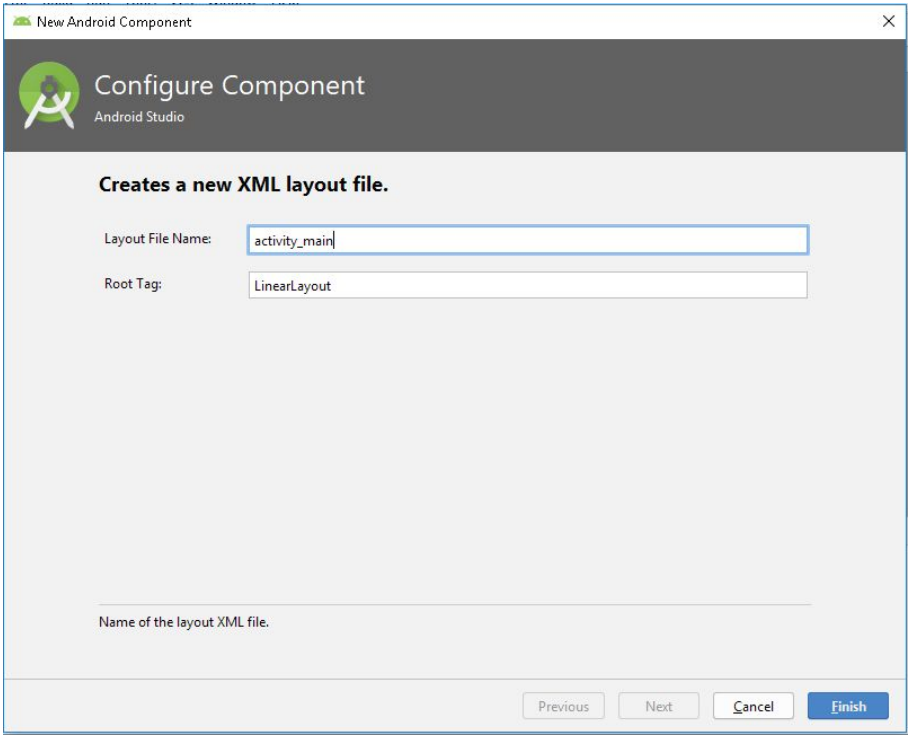


Figura 12: Criando o arquivo de layout.

Percebam que o *Android Studio* cria a estrutura de *layout* do nosso projeto.

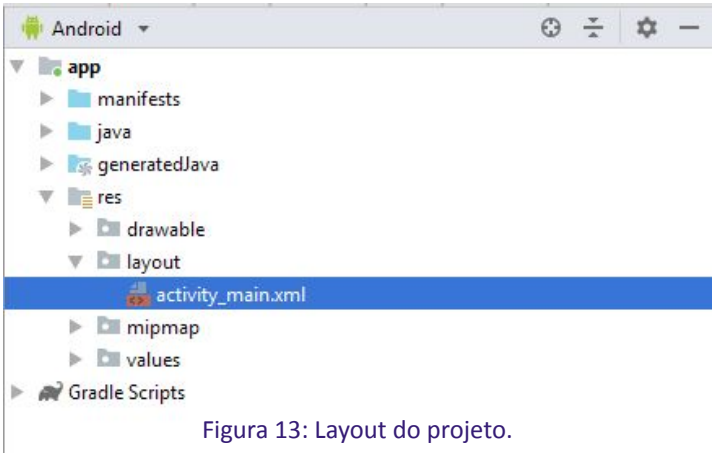


Figura 13: Layout do projeto.

Edite o arquivo activity_main.xml, conforme código abaixo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensagem" />
</LinearLayout>
```

Com o código acrescentado (text view), estamos dizendo que a largura do objeto terá a opção wrap_content, o que significa que o objeto terá apenas o tamanho necessário para sua exibição. E um objeto do tipo *string* com o nome mensagem. Observe que o IDE está reclamando que não encontrou o objeto mensagem.

Então, vamos criar o objeto conforme solicitado. A criação será no *values >> string.xml*, acrescentando o seguinte código: *<string name*="mensagem">Minha primeira mensagem!!*</string>*. Então, nosso arquivo ficará da seguinte forma:

```
<resources>
    <string name="app_name">Exemplo1</string>
    <string name="mensagem">Minha primeira
mensagem!!</string>
</resources>
```

Dessa forma, especificamos o *layout* e dissemos que ele terá apenas uma mensagem: “Minha primeira mensagem!!”. Agora precisamos criar dois arquivos obrigatórios para rodar o aplicativo. São eles: o *MainActivity.java* (tela inicial a ser executada) e o *AndroidManifest.xml* (especificar qual *activity* vai carregar).

Para criar o MainActivity.java, clicamos no botão direito, no pacote *com.ifb.exemplo1* dentro da pasta java, vá ao menu *new >> Java Class*.

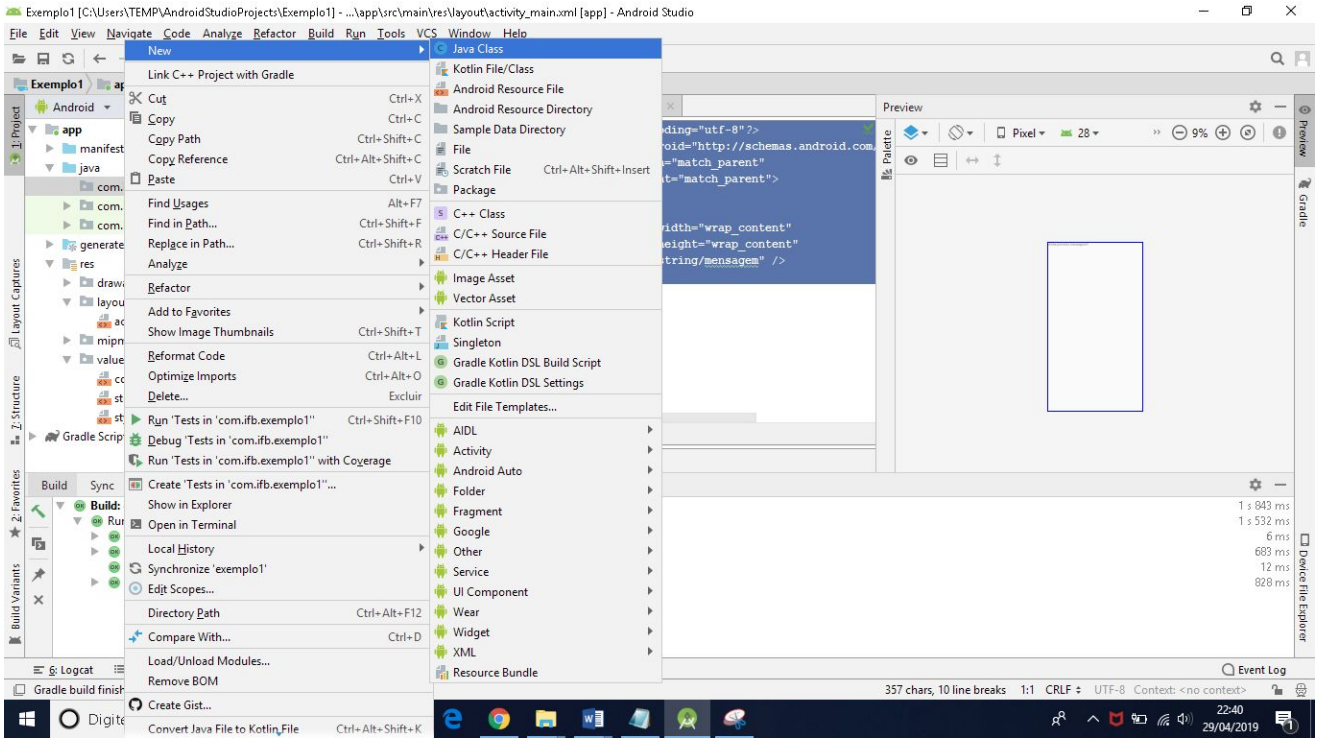


Figura 14: Criando nova classe java.

O nome do arquivo será *MainActivity* e o restante pode ser deixado *default*, conforme imagem abaixo.

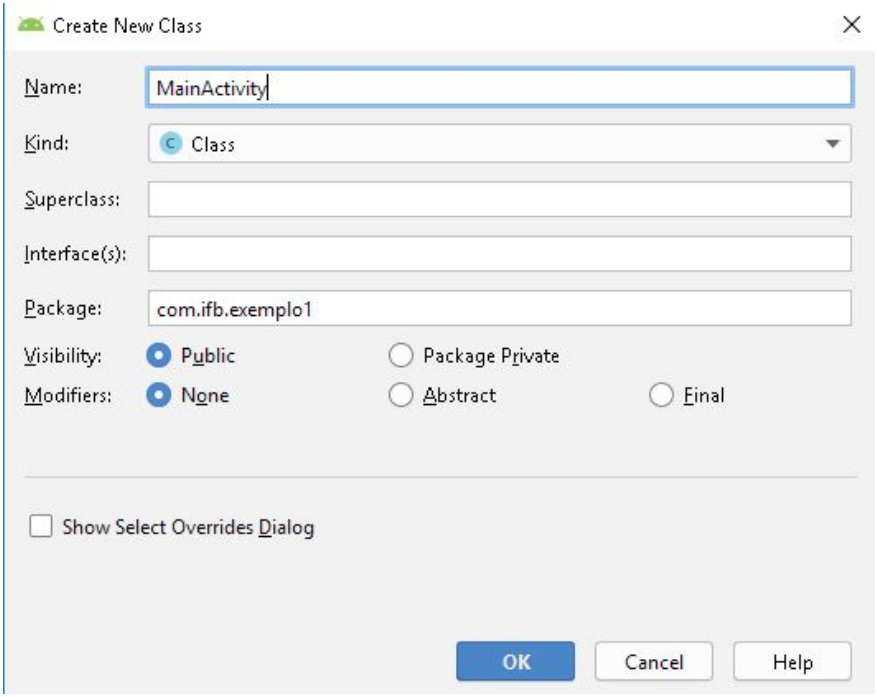


Figura 15: Configurando o MainActivity.java.

Nossa classe *MainActivity* precisa extender de *Activity*. Para isso, basta escrever *extends Activity* e utilizar o *Alt + enter* para atualizar os *imports* da classe (importando a classe do pacote *android.app*). Então, precisaremos dizer qual o *layout (activity)* será chamado no xml. Para isso, utilizaremos o seguinte código:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout. activity_main);
}
```

O código final do *MainActivity.java* ficará da seguinte forma:

```
package com.ifb.exemplo1;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. activity_main);
    }
}
```

Vamos agora editar o *AndroidManifest.xml.* Ele deve ficar da seguinte forma:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ifb.exemplo1">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >

        <activity android:name=".MainActivity" >
            <intent-filter> <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

    </application>

</manifest>
```

Então, ao rodar nossa aplicação com *Shift* + F10, temos o seguinte resultado:

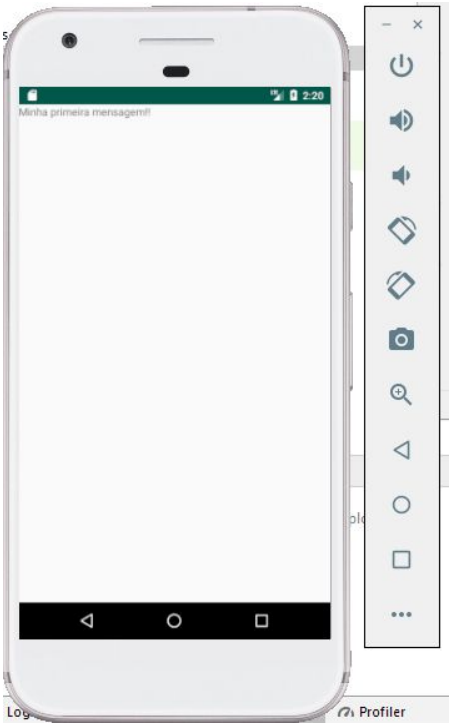


Figura 16: Emulador com a primeira mensagem.

Agora, com o emulador iniciado, basta dar um *play* na aplicação e ela será novamente carregada no emulador.

2.2.2 Criando nova mensagem

Vamos agora criar uma segunda mensagem na tela. Para isso, precisaremos alterar o arquivo *activity_main.xml* com o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensagem" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensagem2" />

</LinearLayout>
```

Agora, precisaremos criar o objeto mensagem2 no *strings.xml*. Dessa forma, o *arquivo strings.xml* ficará com o seguinte texto:

```
<resources>
    <string name="app_name">Exemplo1</string>
    <string name="mensagem">Minha primeira mensagem!!</ string>
    <string name="mensagem2">Minha segunda mensagem!!</ string>
</resources>
```

A linha *android:orientation*="vertical", vai fazer com que os objetos sejam dispostos na vertical. O *match_parent* fará com que o conteúdo se espalhe por toda a tela do dispositivo. Vamos ver nossa tela?

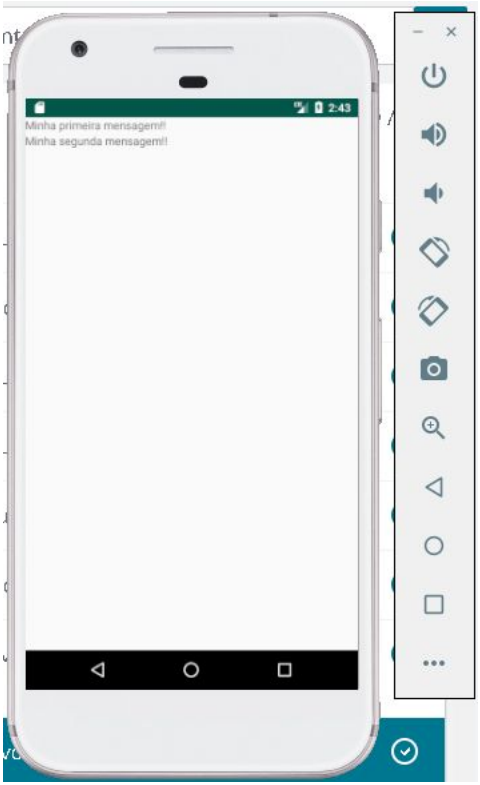


Figura 17: emulador com a segunda mensagem.

2.2.3 Criando uma caixa de texto

Vamos agora criar objetos do tipo texto e botão em nossa aplicação. Para criar esses objetos, vamos inseri-los no arquivo *activity_main.xml*. O código deste arquivo ficará da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensagem" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensagem2" />

    <EditText
        android:id="@+id/nomeText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPersonName" >

        <requestFocus />
    </EditText>


    <Button
        android:id="@+id/OKButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/MensagemOK" />

</LinearLayout>
```

A novidade do código acima está no *ID*, que será utilizado para referenciar cada elemento. Não esqueça que, no Java, há a diferenciação de termos maiúsculos e minúsculos.

Para o objeto de texto *"EditText"*, utilizaremos *textPersonName* que é utilizado para letras e o *requestFocus* é para que o cursor fique no campo automaticamente.

Para o objeto Button, a propriedade *layout_gravity* define o que será centralizado e setamos *a string* que será buscada no *strings.xml*.

Então, agora precisaremos criar esse objeto MensagemOK no *Strings.xml.* Assim, ele ficará da seguinte forma:

```
<resources>
<string name="app_name">Exemplo1</string>
<string name="mensagem">Minha primeira
mensagem!!</string>
<string name="mensagem2">Minha segunda
mensagem!!</string>
<string name="MensagemOK">Pressione aqui.</string>
</resources>
```

Então, vamos ver como ficou nosso aplicativo até agora...

– Legal, professor! Mas como fazemos para colocar alguma funcionalidade nesse botão?

– Nós vamos ver isso pessoal! Mas e se, primeiro, melhorássemos o *layout* da aplicação. Vocês acham isso importante?

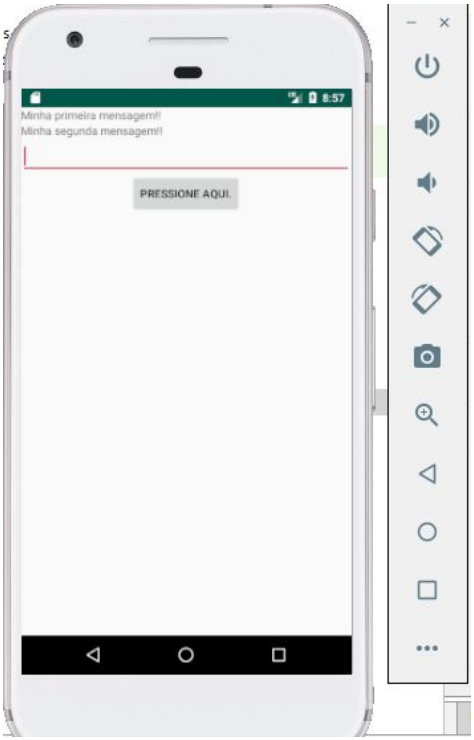


Figura 18: Aplicativo com o campo texto e botão.



Figura 19: Meme sobre design de aplicações.
Fonte: https://www.appivo.com/bad-design-meme/

– Brincadeiras à parte, pessoal! Mas, de fato, uma boa apresentação para o usuário, aliada à facilidade de uso é essencial para o sucesso de um aplicativo. Vamos melhorar um pouco essa apresentação?
– ***Vamos em frente, professor!***

2.2.4 Trabalhando o layout da aplicação

Vamos agora melhorar um pouco o *layout* de nossa aplicação. Nosso curso não focará nos aspectos de *design* da aplicação, mas daremos algumas orientações sobre isso. Comecemos alterando as mensagens do aplicativo, conforme abaixo:

```
<resources>
<string name="app_name">Exemplo1</string>
<string name="mensagem">Meu primeiro
aplicativo</string>
<string name="mensagem2">Digite seu nome</string>
<string name="MensagemOK">Ok</string>
</resources>
```

Lembrando que no arquivo *activity_main.xml*, eu poderia informar o texto direto entre aspas ou fazer a menção ao arquivo *strings.xml* como fizemos.

Para continuar editando o aplicativo, podemos trabalhar no modo texto ou *Design*. Clique no *Design*.

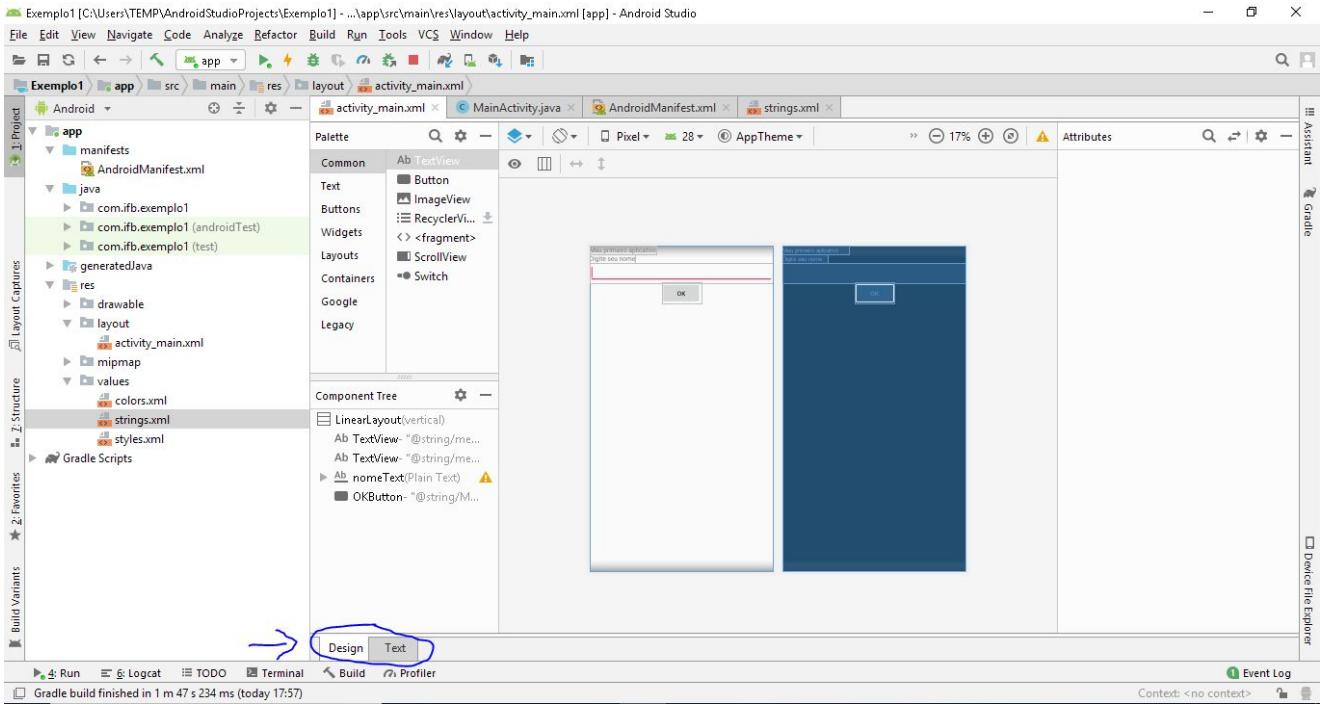


Figura 20: Selecionando modo Design

Ao clicar em *design*, aparecerá uma barra de atributos, que mostrará os atributos de cada um dos objetos do aplicativo ao serem clicados. Vamos clicar na mensagem “meu primeiro aplicativo”. Nas propriedades, selecione *layout_gravity* e clique em *enter*[A-CRF1] [JMdA2] . Vamos também alterar a cor da fonte com a propriedade *textColor* e clicar nos “...”. Ao fazer isso, aparecerá uma tela semelhante à que é mostrada abaixo:

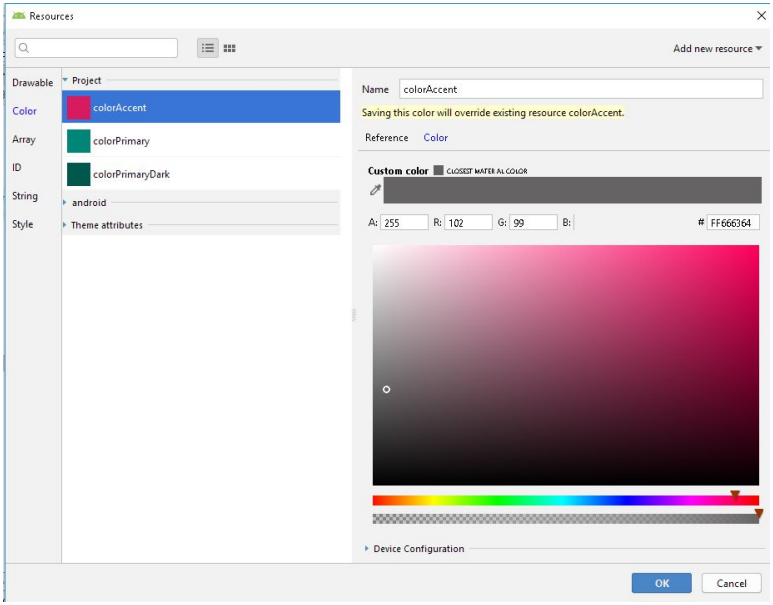


Figura 21: Paleta de cores.

Selecione a cor de sua preferência. Colocarei um cinza mais escuro. Agora, vá até a propriedade *textSize* e informe o valor 20, para aumentar a fonte.

Agora, selecione o texto "Digite seu nome", coloque o tamanho da fonte como 15, a cor vermelha, negrito (propriedade *textStyle*) e centralizado.

Agora, voltemos ao modo *"Text"* e vejamos as alterações que o IDE fez no código. Como falei anteriormente, não entraremos em aspectos mais detalhados do *design* de aplicações, mas aqui temos uma visão geral da gama de recursos que podem ser utilizados.



Figura 22: Vizualização prévia do *Android Studio*.

2.2.5 Inserindo funcionalidades

O que faremos agora será iniciar o uso de funcionalidades, uma das partes mais importantes dos aplicativos. Iremos fazer com que o texto digitado, ao clicar no botão "OK" apareça em outro local, do tipo *text*.

Primeiro, vamos definir o método que será chamado ao clicar no botão, assim utilizaremos o seguinte comando: *android:onClick*="exibeMensagem". Nosso botão então ficará da seguinte forma:

```
<Button
    android:id="@+id/OKButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="@string/MensagemOK"
    android:onClick="exibeMensagem"/>
```

Já criamos nosso método. Vamos então criar um novo *textView* abaixo do botão. Usaremos o seguinte código:

```
<TextView android:id="@+id/mensagemText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Agora, precisaremos criar duas variáveis: uma para receber o nome e outra para exibir o nome. Criaremos esses objetos na classe *MainActivity.java*. Então, criaremos as seguintes variáveis:

```
private EditText Nome;
private TextView Mensagem;
```

Ao criá-las, você deverá utilizar os atalhos *Alt* + *Enter* (com o cursor em cima dos tipos *EditText* e *TextView*), para fazer os *imports* respectivos, que são *import android.widget.EditText* e *import android.widget.TextView*.

Agora, precisaremos informar à classe quais objetos na tela essas variáveis representam. Assim, utilizaremos os seguintes códigos:

```
this.Nome = (EditText) findViewById(R.id. nomeText);
this.Mensagem = (TextView) findViewById(R.id. mensagemText);
```

Faremos isso no método *onCreate*, que ficará da seguinte forma:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout. activity_main);
    this.Nome = (EditText) findViewById(R.id. nomeText);
    this.Mensagem = (TextView) findViewById(R.id. mensagemText);
}
```

Agora, nos resta criar o método que será chamado ao clicar no botão. Esse método terá o seguinte código:

```
public void exibeMensagem(View v) {
    Editable texto =
this.Nome.getText();
    this.Mensagem.setText(texto);
}
```

Não se esqueça de fazer os devidos *imports* que são: *import android.text.Editable* e *import android.view.View*.

– Alguém entendeu o que está fazendo no método exibeMensagem?
– Acho que sim, professor. Mas você pode explicar?
– Claro!

O método pega o conteúdo do Nome {que setamos no *onCreate* recebendo *findViewById*(R.id.nomeText)} e carrega no objeto Mensagem {que aponta para o (*TextView*) *findViewById*(R.id.mensagem*Text*)}. Em outras palavras, uma variável que aponta para o campo texto, que recebe o nome na tela, passa o conteúdo para outra variável que aponta para o texto abaixo.

Vamos então ver como ficou nossa tela?

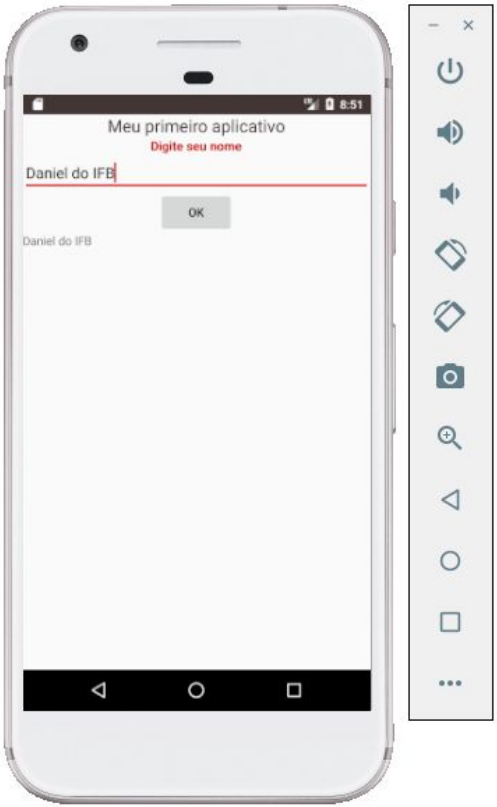


Figura 23: Vizualização prévia do *Android Studio*. https://res.cloudinary.com/practicaldev/image/fetch/s--fPP5x7TC--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https://1.bp.blogspot.com/-3D_vOqmKSTc/XIc8KIeix8I/AAAAAAAAChQ/EZqpHeDxb7E10wkZF-euY9XqomvcCvWqQCLcBGAs/s640/Untitled.png

2.2.6 Trabalhando a mensagem de apresentação

Vamos agora fazer algo simples mas muito útil, inclusive serve de revisão de trabalho com *Strings*. Vamos concatenar uma mensagem ao conteúdo do nome e imprimir uma mensagem de saudação.

Para isso, vamos editar o método exibeMensagem da seguinte forma:

```
public void exibeMensagem(View v) {
    Editable texto = this.Nome.getText();
    String msg = "Olá, " + texto.toString() + " ,seja bem-vindo(a)!!!";
    this.Mensagem.setText(msg);
}
```

Percebam que escrevemos a mensagem "Olá" entre aspas, concatenando com o *toString* do texto. E, finalmente, concatenando com uma mensagem de seja bem-vindo.

Vamos ver como ficou nossa tela...

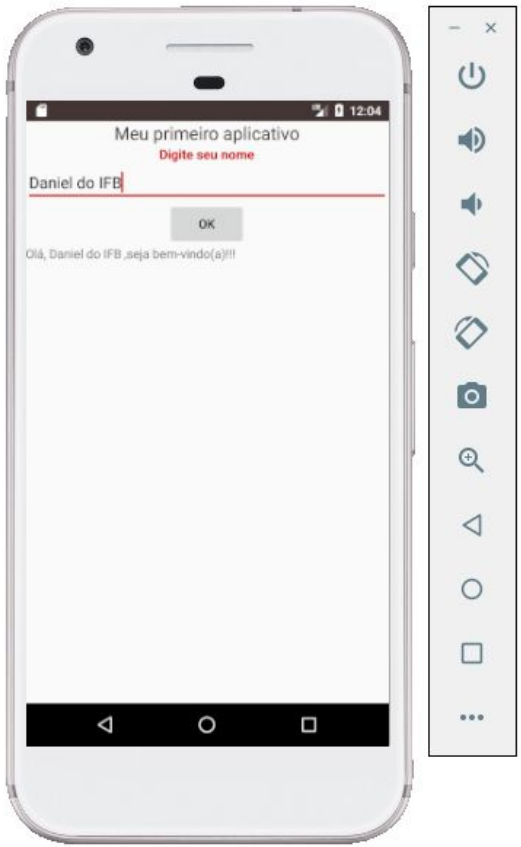


Figura 24: Emulador com mensagem de boas-vindas.

Chegamos ao final da nossa segunda unidade. Ela abordou aspectos mais específicos sobre o sistema operacional *Android* e iniciamos o desenvolvimento de nosso primeiro aplicativo, aprendendo a manipular *layout*, a colocar mensagens em tela, componentes de textos, botões e, por fim, a estrutura de funcionalidades.

Para sintetizarmos tudo o que abordamos nesta primeira unidade, vamos rever os principais pontos estudados?



Bora rever!

Vimos, no primeiro tópico, aspectos do sistema operacional *Android*. Vimos como criar nosso projeto e como fica a estrutura de arquivos e a pasta dele.

No segundo tópico, fizemos nossa primeira aplicação com funcionalidades. Vimos como manipular o *layout* da aplicação, como inserir componentes de texto e botões e aprendemos a criar funcionalidades.

Na próxima unidade entraremos em mais aspectos dos códigos fontes, aprendendo mais detalhes sobre os activitys e os intents. E veremos mais possibilidades para nossas aplicações.

Sigamos em frente!

Unidade 3 DESENVOLVENDO UM APLICATIVO COMPLETO

Nesta unidade, construiremos um aplicativo completo. Trabalharemos aspectos de layout e diversas funcionalidades. Iniciaremos com alguns conceitos chaves. Vamos em frente!

3.1 Construindo o projeto

Antes de criar nosso novo projeto, vamos falar de alguns conceitos importantíssimos para os aplicativos *Android*. Vamos juntos!

3.1.1 O que é o Activity?

Você sabe o que é uma *Activity*?

- Ué professor, já trabalhamos com isso! Não seria o layout da aplicação?
- Isso mesmo! Mas vamos entrar em mais detalhes?! Venha comigo!

Activity atua como gerenciador de interface com o usuário. Todas as aplicações *Android* devem conter uma ou mais *Activities*, executando uma *Activity* por vez. Cada tela tem um *Activity* própria. Numa *Activity* temos todas as descrições de elementos da tela, como botões, textos, imagens, vídeos, cores, etc.

Embora seja possível configurar os elementos da tela através de código java, é recomendado que isso seja feito através de arquivo XML.

3.1.2 O que é o Intent?

E uma *Intent*? Sabe o que é?

- Não sei professor!

- *Intent* significa intenção, dando-nos uma visão de que se trata de operações.

Uma *Intent* é uma descrição abstrata de uma operação que será executada. Ela pode ser utilizada para iniciar uma *Activity*, "ativar" um *broadcast*, enviar uma mensagem para uma outra aplicação, dentre outras ações.

3.1.3 Trabalhando uma nova aplicação

Pessoal, após entendermos os conceitos iniciais para trabalharmos nossos aplicativos, vamos agora trabalhar em nosso novo projeto. Será uma tela de login em que aplicaremos algumas funcionalidades.

Teremos inicialmente dois *Activities*, sendo um com usuário e senha e outro que informe a mensagem se está ou não logado.

Primeiramente feche o projeto anterior, utilizando o menu ***File >> Close Project***.

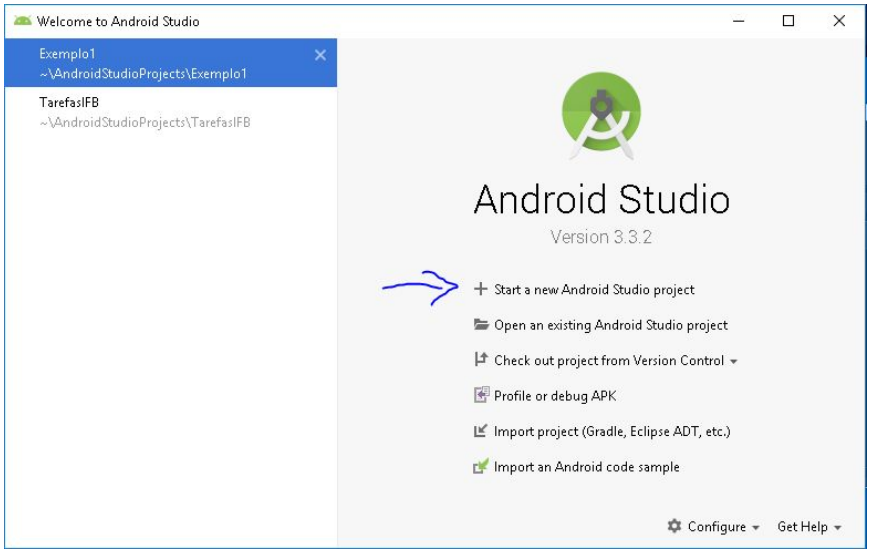


Figura 1: Tela de projetos recentes e opções para novo projeto.

Na tela seguinte serão mostrados os projetos recentes e haverá uma opção para criar um novo projeto. Clique em *Start a new Adroid Studio Project*.

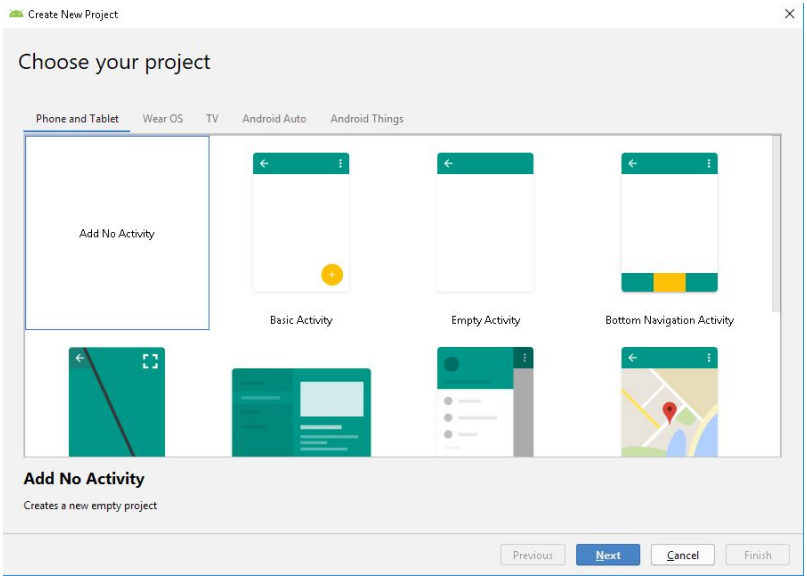


Figura 2: Configurando o Activity do novo projeto.

Na configuração do *Activity*, selecione a opção sem o *Add No Activity* para que possamos criar o nosso projeto.

Depois disso configure o projeto com o nome “Exemplo2”. Os parâmetros adicionais podem ser idênticos ao descrito ao lado.

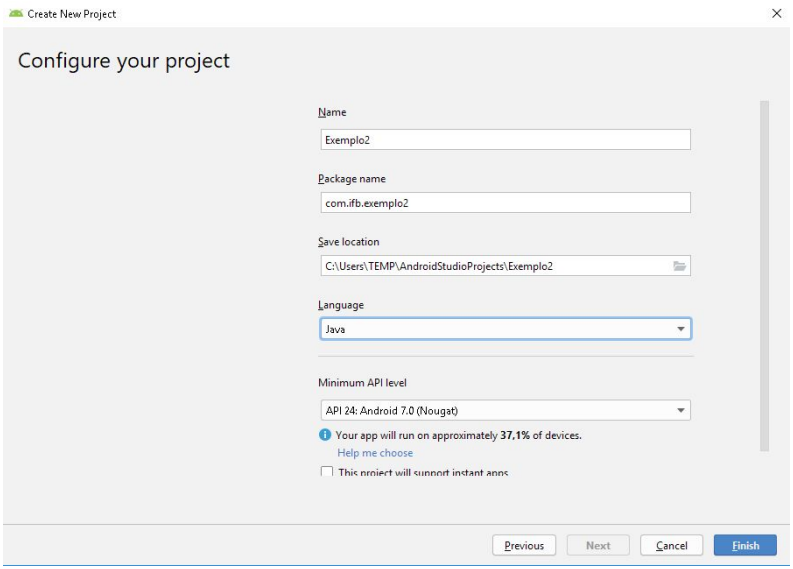


Figura 3: Configuração do novo projeto.

Clique em *finish* e nosso projeto foi criado.

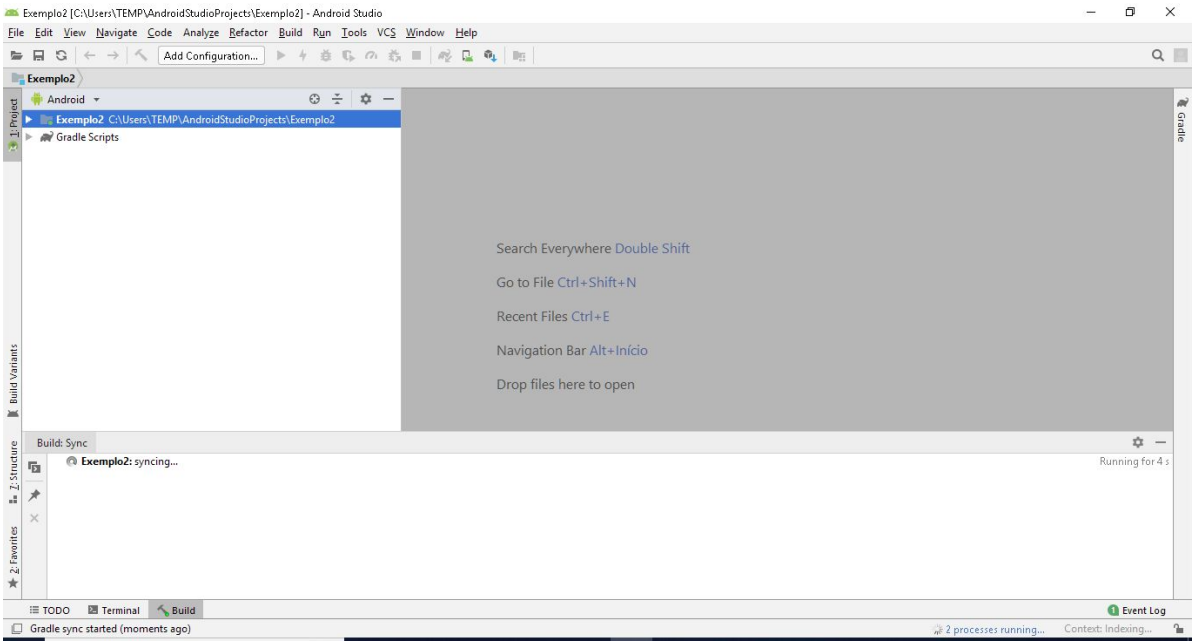


Figura 4: Tela do novo projeto criado.

Pronto! Temos nosso novo projeto criado. Vamos agora trabalhar com o *layout* ou as *Activities* de nossa aplicação. Faremos isso a seguir.

3.2 Trabalhando o layout

Neste tópico, vamos criar e configurar as *activities* do nosso aplicativo.

3.2.1 Criando a primeira *Activity*

Pessoal, vamos então trabalhar nossa tela de *login*. A primeira coisa a fazer é criar o *layout* conforme já fizemos anteriormente. Clique com o botão direito na pasta "res", depois ***New >> XML >> Layout XML File***, como mostrado na figura abaixo.

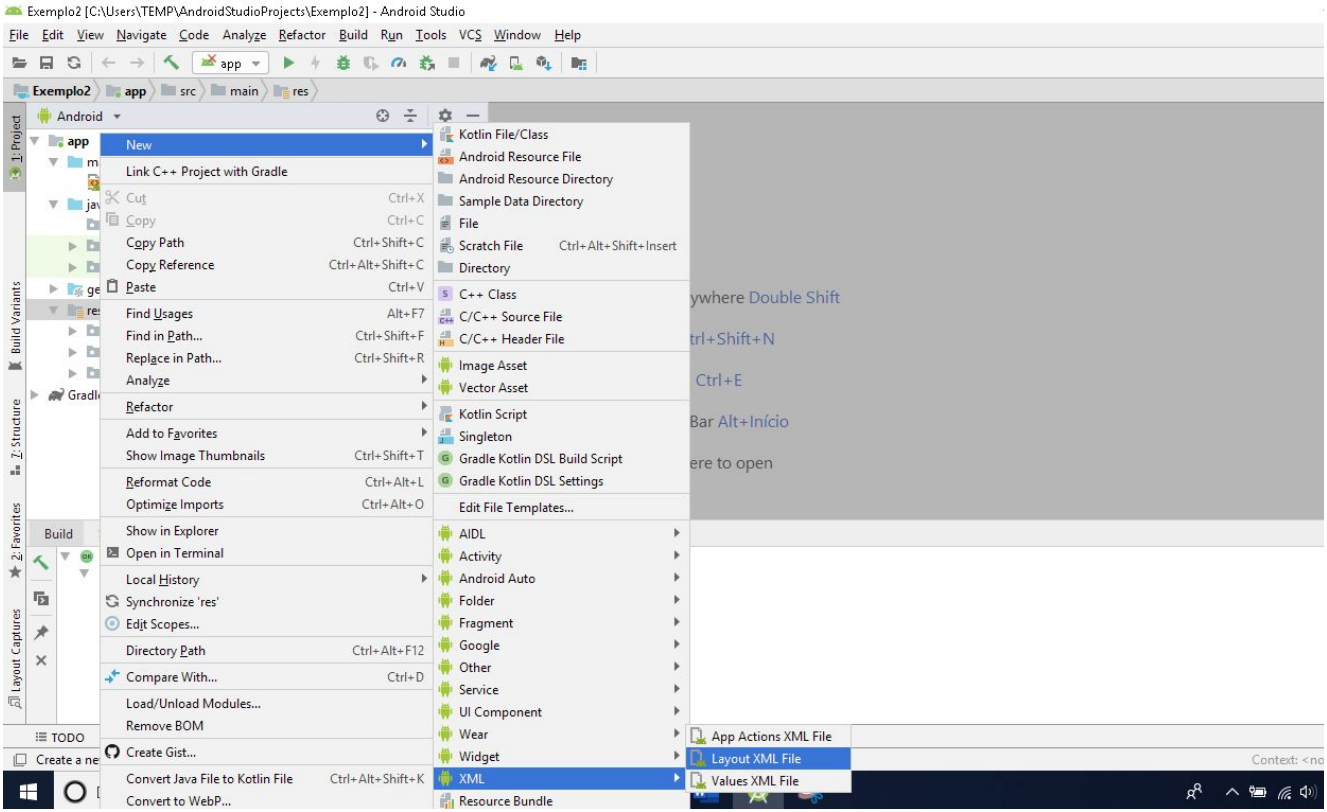


Figura 5: Tela de criação do *Activity*.

Chamaremos nossa primeira tela de "*login*". Por ser a tela em que logaremos, a qual será chamada inicialmente. Deixaremos a opção padrão "*LinearLayout*".

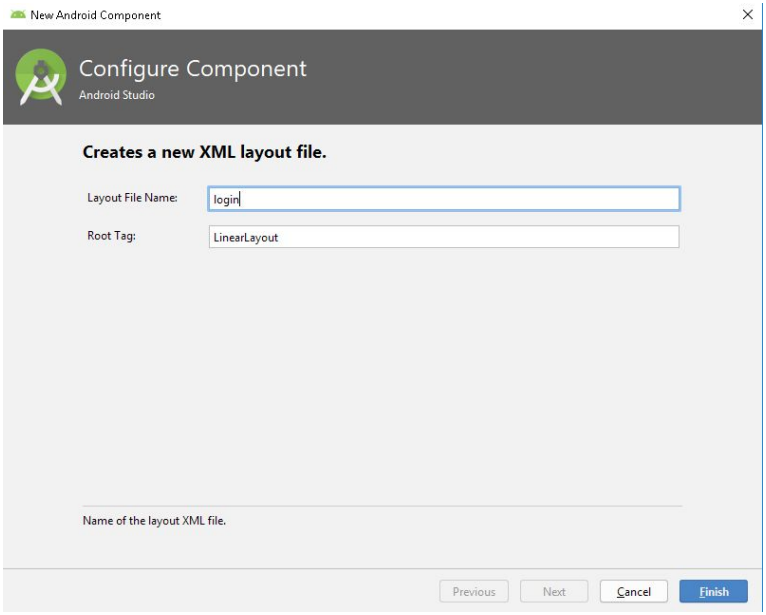


Figura 6: Configuração do Activity de Login.

Clique em *Finish*.

Perceba que o *Android Studio* criou a pasta *layout* e o *Activity login* dentro dele, conforme fizemos anteriormente. Vamos então para o modo *Design* para o *Activity*.

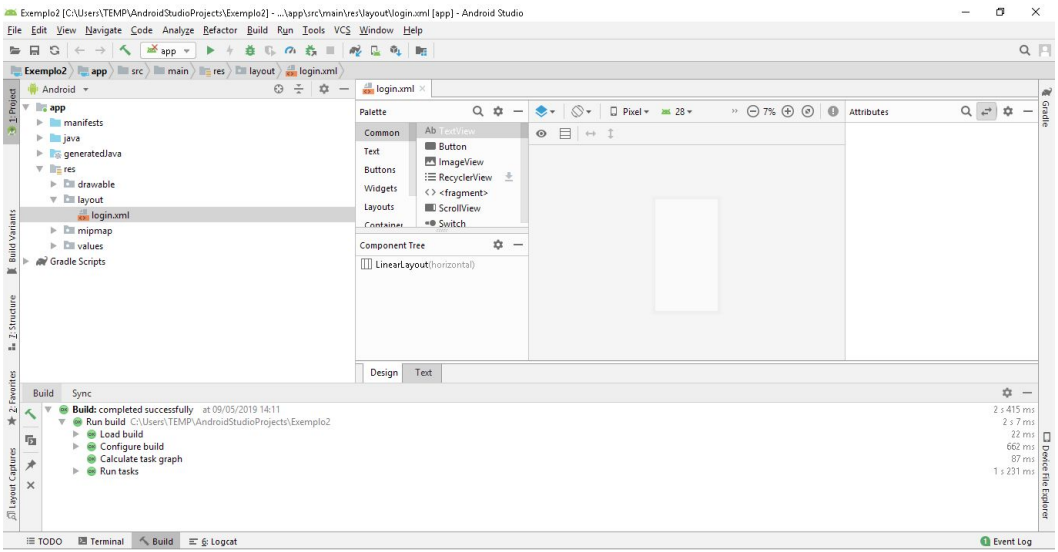


Figura 7: Tela de layout com Design.

3.2.2 Configurando o primeiro *Activity*

Vamos iniciar a tela com a orientação do *Layout*. Vamos setar como vertical. Para isso, clique no *LinearLayout* dentro da aba *Component Tree*.

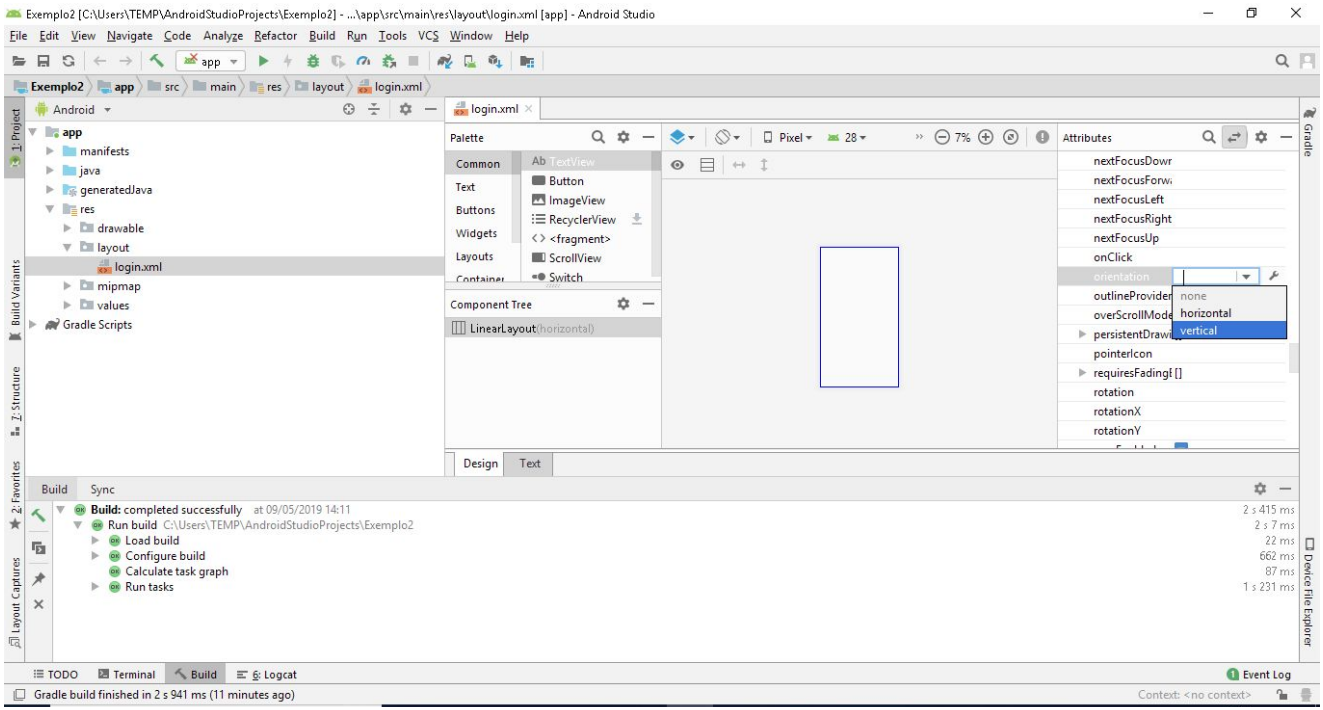


Figura 8: Setando a orientação do layout.

Agora vamos importar as imagens do projeto. Clique com o direito na pasta *res >> New >> Image Asset*.

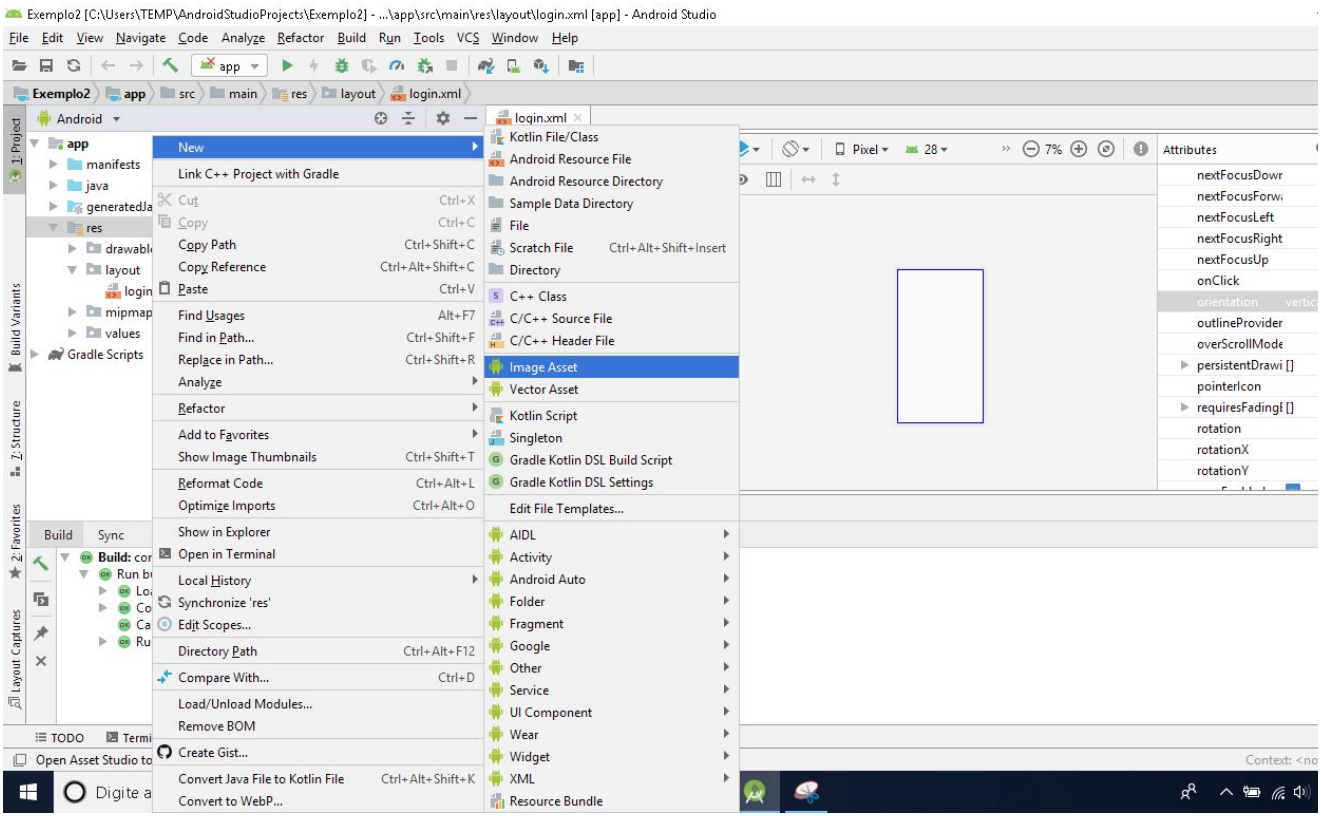


Figura 9: Tela de criação de uma Image Asset.

A tela seguinte nos dá uma gama de opções para configuração dessas imagens. Entre as opções de imagens temos: *Launcher Icons* (imagens padrão novos e legados), *Actions Bars e Tab Icons* (ícones de barra de tarefas) e *Notification Icons* (ícones de notificações).

Vamos usar o *Laucher Icons (Adaptative e Legacy)* e utilizaremos as imagens disponibilizadas no *git* para a pasta de nosso projeto (Exemplo 2). Esses arquivos podem ser encontrados em https://github.com/danielcelestino/tutorial_ifb_android2.

Utilizaremos a imagem *logon.png*. Essa seleção é feita através dos campos *Path* na área *Source Asset*. Configure o restante conforme a imagem a seguir.

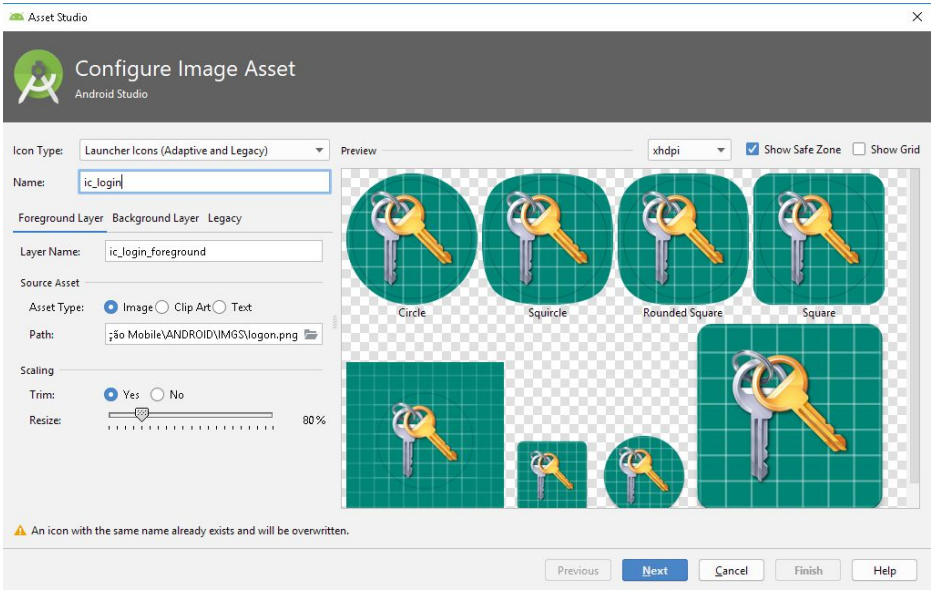


Figura 10: Configurando o Asset de login.

Entre as opções, temos o ajuste de escala e tipos de imagem. Clique em *Next*. Aparecerá então uma tela de confirmação das imagens.

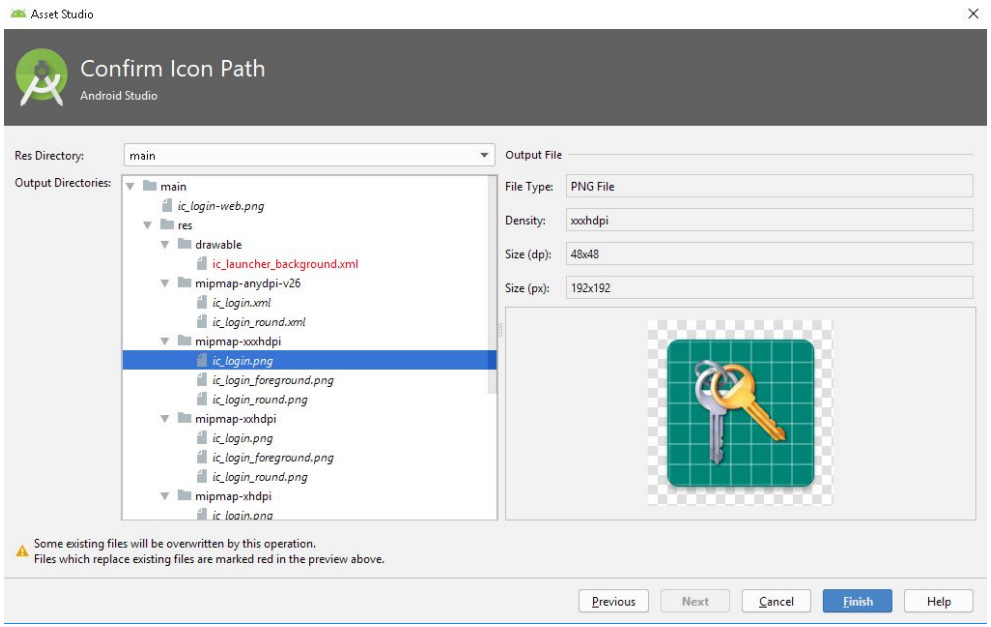


Figura 11: Tela de confirmação dos ícones.

Ao clicar no *Finish*, será criado um novo diretório na pasta *res*. Vejamos...

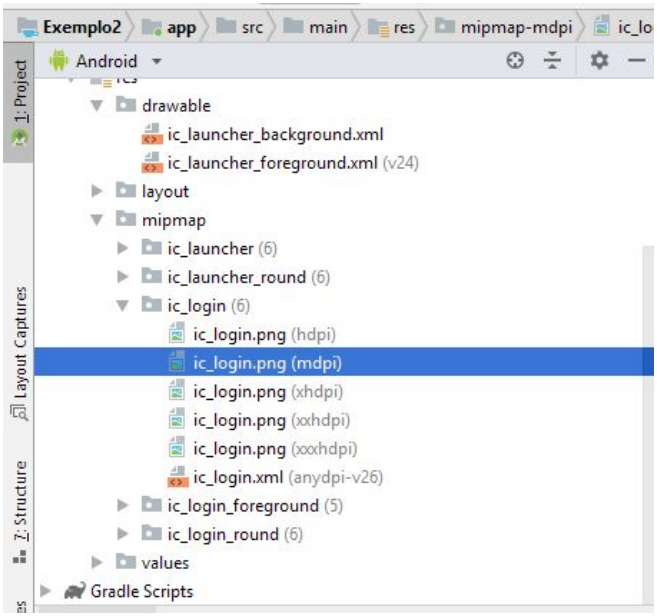


Figura 12: Pasta do ic_login.

Agora vamos colocar essa imagem na tela do aplicativo. Para isso, utilizaremos o componente *ImageView*.

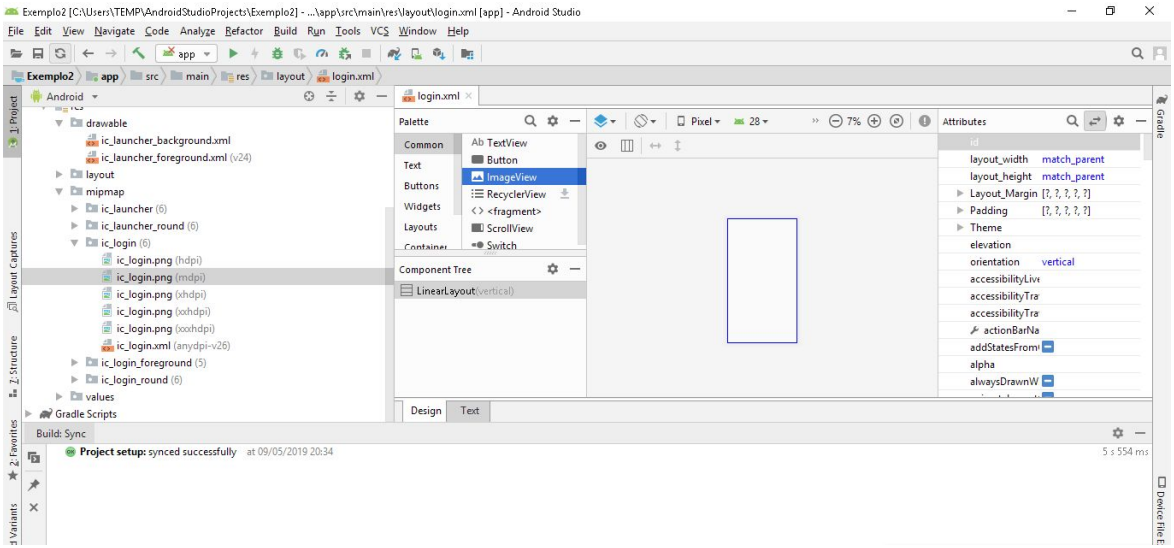


Figura 13: Componente image view.

Basta clicar no componente e arrastar para a tela. Então aparecerá uma tela para configuração dessa imagem, conforme figura ao lado:

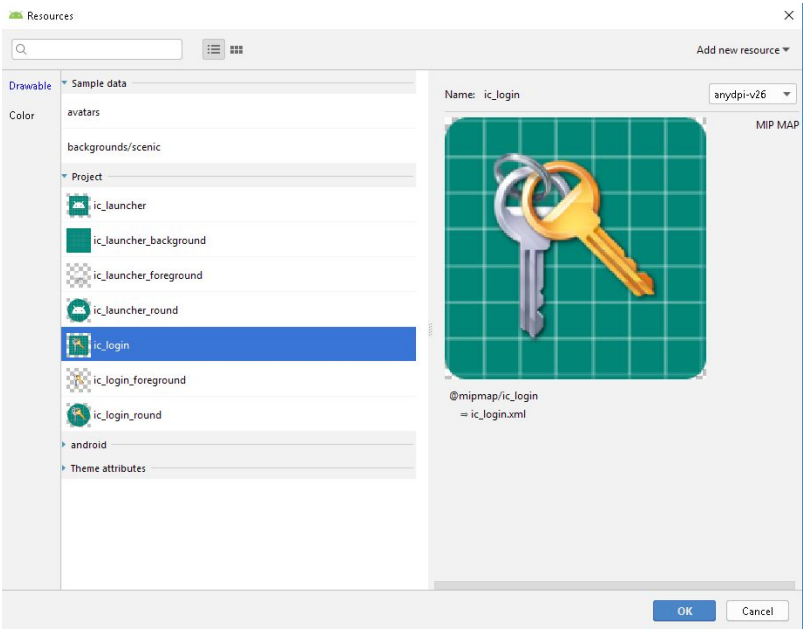


Figura 14: Configuração do Imageview.

Selecione o projeto e o objeto *ic_login*. Após isso, clique em Ok. Veremos que a imagem já ocupa um local na tela. Aumente um pouco a imagem para melhor visualização. Altere também o *ID* da imagem para facilitar a localização futuramente.

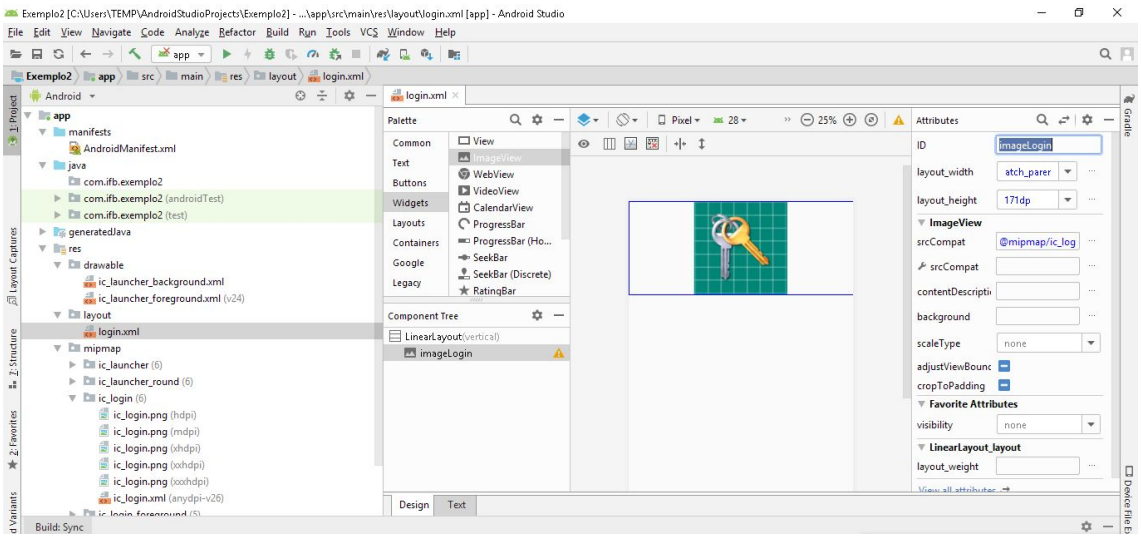


Figura 15: Configuração da tela com ic_login.

Criaremos agora um *text view* com o nome do usuário. Utilizaremos o *TextView*. Basta clicar e arrastar para abaixo da imagem.

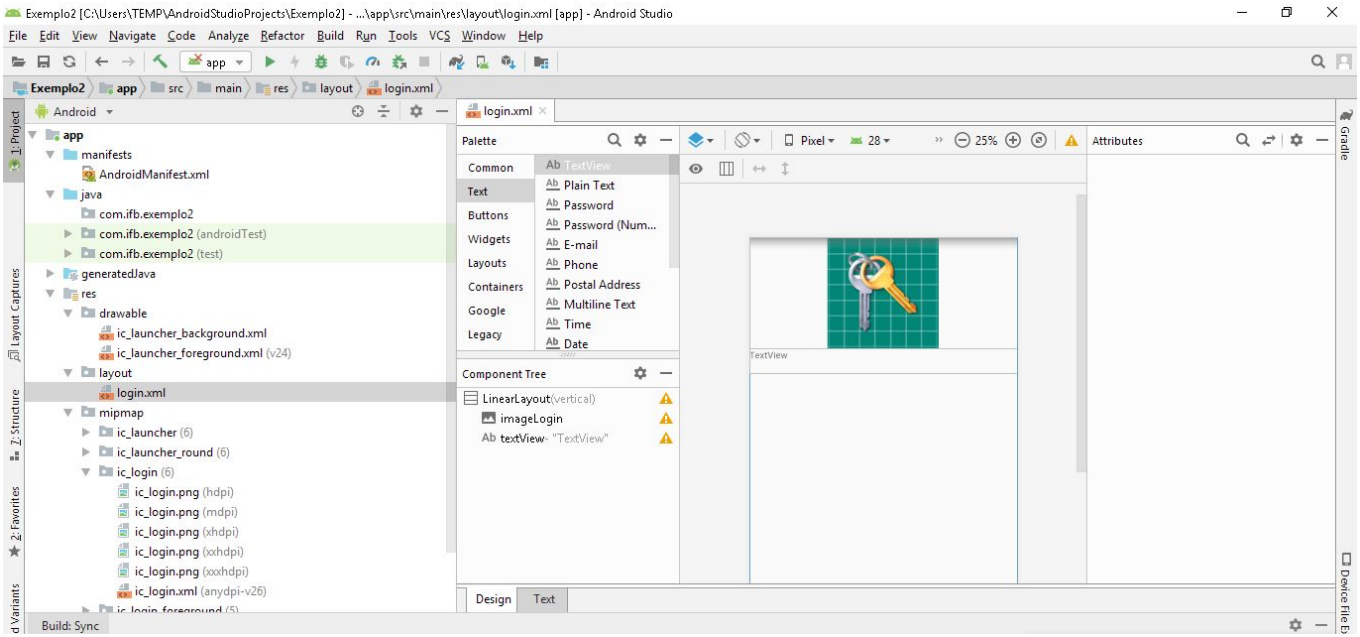


Figura 16: Inserindo TextView de login.

Clique em cima do *TextView* e altere a propriedade *id* para *textLogin*. Altere também a propriedade *text* para “Digite o nome do usuário:”.

Clique em *View All atributes* para ver todas as propriedades. Altere a propriedade *textColor* para *holo_red_dark*. A propriedade *textAlignment* para *center*. A *textSize* para 18sp. E, por fim, o *textStye* para *bold*.

Agora vamos adicionar a caixa de texto para digitação. Utilizaremos o *Plain Text*. Altere a propriedade *layout_gravity* para *center* e o *TextAlignment* para *center*. Deixe o *text* em branco. Ajuste o tamanho do componente para melhor visualização. Importante alterar o nome do *ID* para *“editLogin”* para poder ser referenciado depois.

Agora que os elementos de *login* foram criados, copie dos dois e cole abaixo para fazer a senha. Altere o *id* do texto para *textSenha* e o *text* para “Senha:”.

Altere também o campo *text* para *editSenha* e o *inputType* para *textPassword*.

Agora vamos adicionar um botão. Selecione o *Button* e arrraste logo abaixo. Diminua um pouco o botão e altere o *text* para “Ok”, o *layout_gravity* para *“center”* e o *id* para *“buttonOk”*.

Nossa tela até agora está assim:

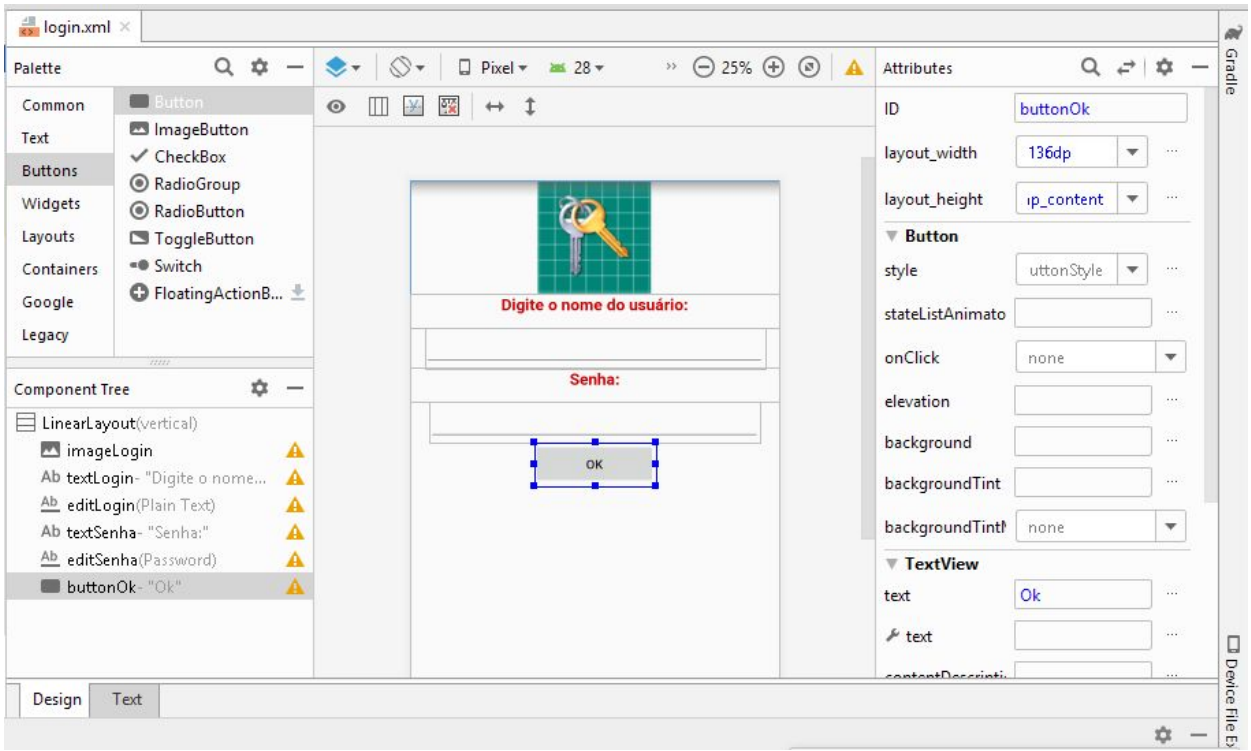


Figura 17: Tela inicial de login.

3.2.3 Configurando o segundo *Activity*

Vamos agora criar a tela principal, que será mostrada após o sucesso no login. Já aprendemos como fazer isso: clique com o botão direito na pasta *res >> New >> XML >> Layout XML*. O *layout file name* será *"principal"*.

Nessa segunda tela, não esqueça de setar a *orientation* para vertical. Vamos agora criar uma imagem para essa tela principal. Utilizaremos a de nome: *user.png.* Vejamos essa tela de configuração novamente:

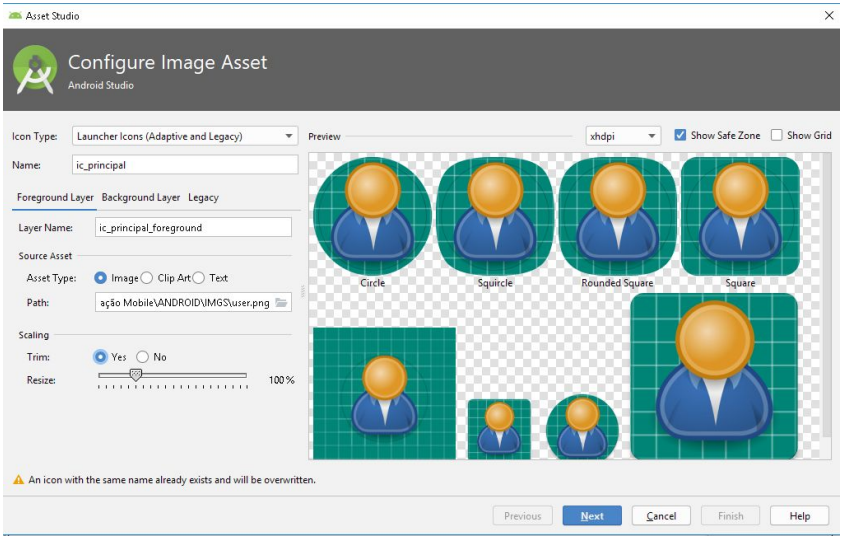


Figura 18: Tela de configuração da tela principal.

Perceba que usamos o nome ic_principal e ic_principal_foreground. Vamos agora colocar a imagem na tela. Já vimos que usaremos o *ImageView*. Arraste-o para a tela e selecione o ic_principal.

Vamos agora criar um texto embaixo dessa imagem com o comentário: "Você está logado!".

- Já sabemos como fazer isso não é, pessoal!?
- Sim, professor!

3.2.4 Configurando o *Activity* que será chamado primeiro

Vamos agora setar o *Activity* principal da aplicação. Para isso, vamos abrir o *AndroidManifest*.xml. Para isso alteraremos o código do Alert. No final, nosso código do *AndroidManifest*.xml ficará da seguinte forma:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ifb.exemplo2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >


        <activity android:name=".login" >
            <intent-filter> <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

    </application>
</manifest>
```

Observem que estamos chamando o *login*. Em nosso código, estamos setando o *login* como principal (*android.intent.action.MAIN*) e executando-o assim que carregar a página (*android.intent.category.LAUNCHER*).

Agora só nos falta uma classe java para ter o código desse *activity*. Vamos criar uma nova classe que terá o nome *Login*. Clique com o botão direito na pasta do projeto, que fica dentro da pasta java. Após isso, selecione *New >> Java Class*. Observe que o nome das activities interfere em todo o projeto, assim nossa classe se chamará *"login"*.

Nosso código da classe ficará da seguinte forma:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.os.Bundle;

public class login extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout. login);
    }
}
```

A classe R é uma das classes principais do *Android*, em que eu tenho as informações do *Activity*. Observe que estamos indicando o *login* como parâmetro do *ContentView*.

Vamos agora testar nosso aplicativo!?

Lembrando que, para isso, precisaremos ter um dispositivo virtual criado. Caso não exista ainda, já vimos como criar.

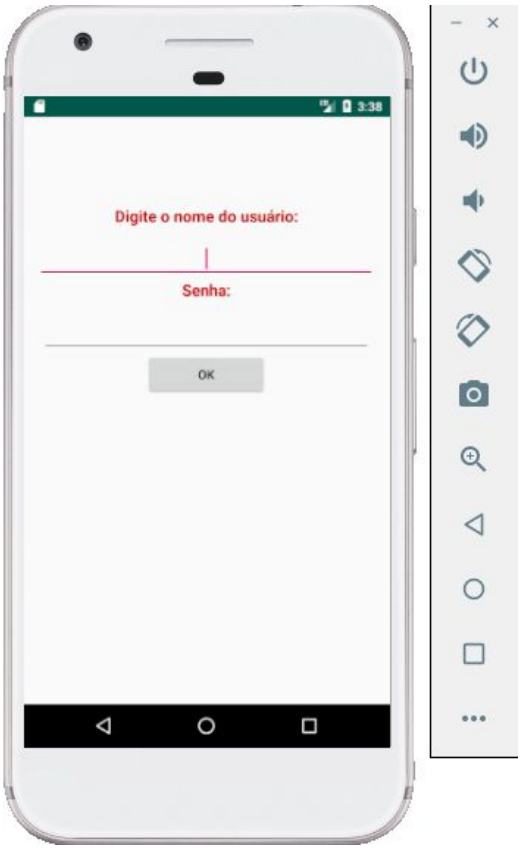


Figura 19: Tela de login versão 1
https://medium.com/@lcmuniz/parte-4-executando-a-aplica%C3%A7%C3%A3o-no-emulador-do-android-e398b92555f9

- Como assim, professor! Não apareceu a imagem no emulador. Esse negócio não funciona!



Figura 20: Meme tentando me enganar.
Fonte: http://geradormemes.com

– É verdade, pessoal! Verifiquei, em algumas pesquisas na Internet, que as versões mais recentes no *Android Studio* não estão com o uso de imagem funcionando ao realizar a inserção pelo modo *Design*. Vou explicar melhor.

Ao fazer, no modo *design*, o código inserindo pelo IDE é *app:srcCompat = "@mipmap/ic_login"*. E isso não está funcionando bem em algumas versões. Para corrigir o problema, temos que alterar esse código (no modo *Text* ao em vez de *Design*) para *android:src="@mipmap/ic_login"*.

Nosso código então ficou da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/an
droid"

xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/imageView3"
        android:layout_width="match_parent"
        android:layout_height="154dp"
        android:src="@mipmap/ic_login" />

    <TextView
        android:id="@+id/textLogin"
        android:layout_width="match_parent"
        android:layout_height="38dp"
        android:text="Digite o nome do usuário:"
        android:textAlignment="center"

android:textColor="@android:color/holo_red_dark"
        android:textSize="18sp"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/editLogin"
        android:layout_width="379dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:ems="10"
        android:inputType="textPersonName"
        android:textAlignment="center" />
```

```
<TextView
        android:id="@+id/textSenha"
        android:layout_width="match_parent"
        android:layout_height="38dp"
        android:text="Senha:"
        android:textAlignment="center"
        android:textColor="@android:color/holo_red_dark"
        android:textSize="18sp"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/editSenha"
        android:layout_width="369dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:ems="10"
        android:inputType="textPassword"
        android:textAlignment="center" />

    <Button
        android:id="@+id/buttonOk"
        android:layout_width="136dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Ok" />

</LinearLayout>
```

Vejamos se a nossa tela funcionou!

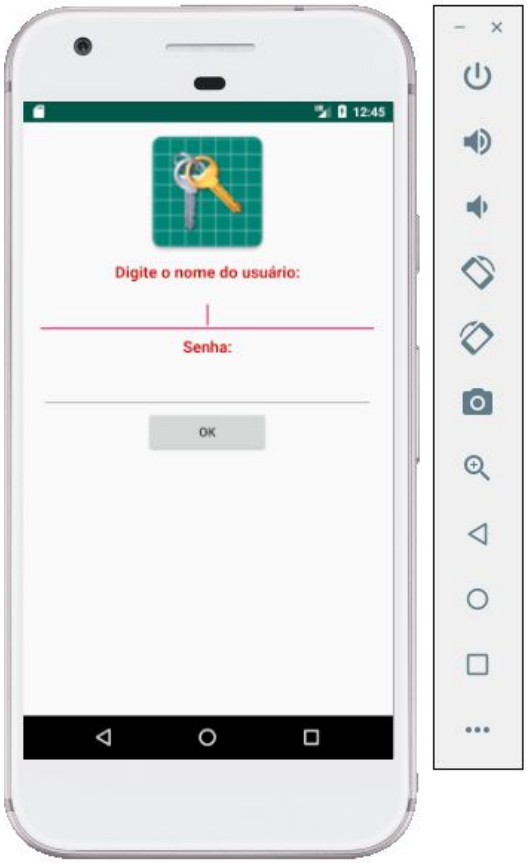


Figura 21: Tela com a exibição da imagem.

- Agora sim, professor!

- Ok pessoal! Vamos então colocar esse botão ok para funcionar?

- Vamos em frente, professor!

- No próximo tópico iremos trabalhar as funcionalidades.

3.3 Criando as funcionalidades

Olá pessoal mais uma vez! Nesse tópico vamos trabalhar algumas funcionalidades! Começaremos fazendo o botão “Ok”, validar a senha e enviar para a tela principal de nosso aplicativo.

3.3.1 Abrindo um novo Activity

Vamos começar a evoluir nossa aplicação. A primeira coisa será receber um *email* ao invés do nome do usuário. Temos então que alterar o campo para o tipo *email* e o texto para “Digite o usuário:”.

Escolha novamente o modo *Design*, altere o texto “Digite o nome do usuário:” para “Digite o usuário:”.

Agora selecione a caixa de texto e vá na propriedade *inputType* e selecione a opção *textEmailAdress*.

Agora voltemos ao modo *text* para setar as ações a serem realizadas ao clicar no botão “Ok”. No modo *text* precisamos adicionar a linha *{android:onClick="verificaLogin"}* no botão Ok. Dessa forma, o código do botão ficará como segue.

```
<Button
    android:id="@+id/buttonOk"
    android:layout_width="136dp"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Ok"
    android:onClick="verificaLogin"/>
```

Perceba que fazemos referência ao método *verificaLogin*, que ainda não existe. Mas vamos, antes, criar a classe referente à tela do *Activity* principal que criamos anteriormente.

Já sabemos que devemos clicar com o botão direito no pacote com.ifb.exemplo2 dentro da pasta Java, *New >> Java Class*. Essa classe terá o seguinte código:

```
package com.ifb.exemplo2;

import android.app.Activity;

public class principal extends Activity{
}
```

Importante destacar que o nome da classe deve ser o mesmo do *Activity*, ou seja, principal.xml, isto é, deve ter o nome principal. Vamos agora colocar o *Activity* que será chamado ao executar essa classe, ou seja, o método *onCreate*. Esse método terá o seguinte

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

Com os *imports* necessários, a classe ficará da seguinte forma:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.os.Bundle;

public class principal extends Activity{

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

Agora vamos voltar à classe *Login.java* para criar o método que será chamado ao clicar no botão "Ok". Vamos começar criando as duas variáveis que receberão as caixas de textos da tela de *login*. Teremos então que criar:

```
private EditText Usuario;
private EditText Senha;
```

Não esqueça do *import* respectivo ao *EditText* (*android.widget.EditText*). Essas variáveis receberão os valores digitados nos campos de nossa tela que são respectivamente *editLogin* e editSenha.

Temos que alterar o método *onCreate* acrescentando o seguinte:

```
this.Usuario = (EditText)
findViewById(R.id.editLogin);
this.Senha = (EditText)
findViewById(R.id.editSenha);
```

Dessa forma, o método ficará assim:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.login);
    this.Usuario = (EditText) findViewById(R.id.editLogin);
    this.Senha = (EditText) findViewById(R.id.editSenha);
}
```

Vamos, finalmente, criar o método *verificaLogin,* da seguinte forma:

```
public void verificaLogin(View v) {
    String txtUsuario = Usuario.getText().toString();
    String txtSenha = Senha.getText().toString();

    if (txtUsuario.equals("teste@ifb.com") && txtSenha.equals("123")) {
        Intent intent = new Intent (this, principal.class);
        startActivity(intent);
    }
    else {


    }
}
```

Estamos recebendo o valor de usuário e senha como *String*, comparando o *email* e senha que são teste@ifb.com e senha 123. Então, criamos uma nova intente que chama a classe Principal.java. Caso a senha seja incorreta, por enquanto não fazemos nada (*else* em branco).

Para finalizar, precisaremos setar todos os *Activities* no *AndroidManifest.xml*. Basta então inserir o seguinte código: *<activity android:name=".principal" />*. Dessa forma, nosso *AndroidManifest.xml* ficará da seguinte forma:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ifb.exemplo2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >


        <activity android:name=".login" >
            <intent-filter> <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

        <activity android:name=".principal" />

    </application>
</manifest>
```

Eis então nossa tela de sucesso no *login*:

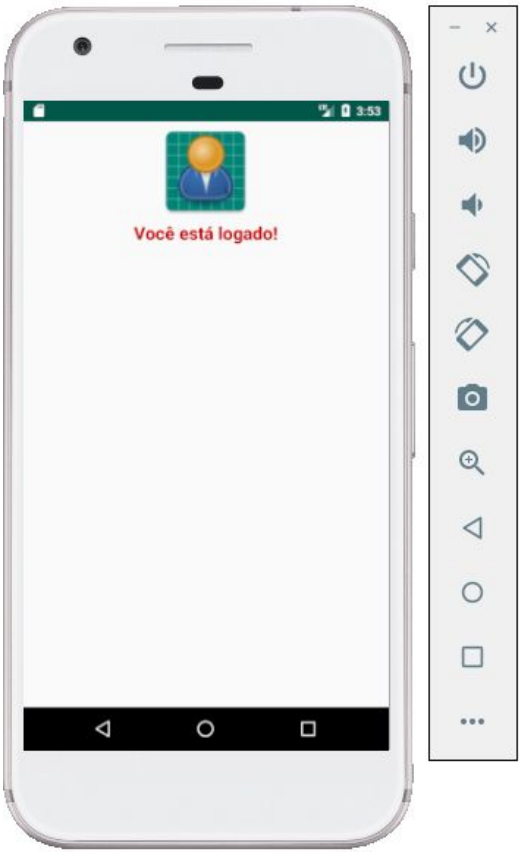


Figura 22: Tela de login com sucesso.

3.3.2 Utilizando *Dialog*

Vamos agora adicionar um novo recurso em nossa aplicação. Será mostrar uma mensagem em forma de *Dialog* para quando o usuário ou a senha forem inválidos. Para fazer isso, utilizaremos o seguinte código no *else*:

```
AlertDialog alertDialog =  new AlertDialog.Builder( this).create();
alertDialog.setTitle( "Erro!");
alertDialog.setMessage( "Nome de usuário e/ou senha incorretos!!");
alertDialog.show();
```

Estamos então criando um *alertDialog* e setando o título, mensagem e exibindo a mensagem. Dessa forma, nosso código ficará assim:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class login extends Activity {

    private EditText Usuario;
    private EditText Senha;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. login);
        this.Usuario = (EditText) findViewById(R.id. editLogin);
        this.Senha = (EditText) findViewById(R.id. editSenha);
    }

    public void verificaLogin(View v) {
        String txtUsuario =  Usuario.getText().toString();
        String txtSenha =  Senha.getText().toString();

        if (txtUsuario.equals( "teste@ifb.com") && txtSenha.equals( "123")) {
            Intent intent =  new Intent (this, principal.class);
            startActivity(intent);
        }
```

```
 else {
            AlertDialog alertDialog =  new
AlertDialog.Builder( this).create();
            alertDialog.setTitle( "Erro!");
            alertDialog.setMessage( "Nome de usuário e/ou senha
incorretos!!");
            alertDialog.show();
        }
    }

}
```

Vamos testar?

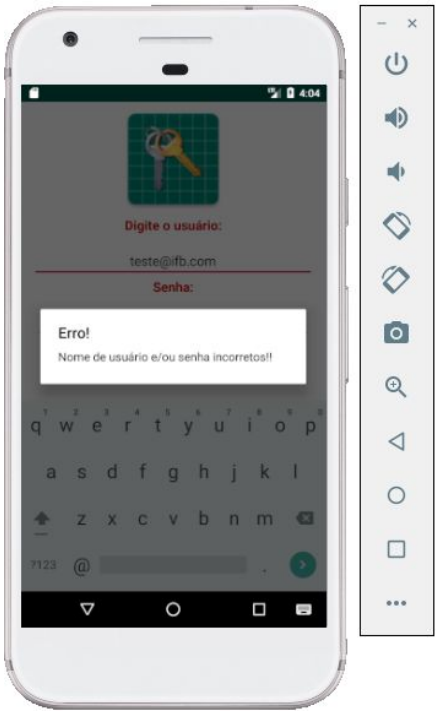


Figura 23: Alert de erro de usuário ou senha.

3.3.3 Passagem de parâmetros

Como fazemos para passar os dados entre *Activities*? Suponha que queiramos passar o *email* do usuário para a próxima tela. Vamos fazer isso?

A primeira coisa a fazer é inserir o elemento na chamada da Intent com o Método *putExtra* que ficará da seguinte forma: *intent.putExtra("EMAIL", txtUsuario)*.

Agora, na classe principal (a *Activity* que receberá o texto), vamos criar uma variável para referenciar o campo da tela que receberá o *email* informado na tela anterior. Faremos isso com o seguinte código *private TextView usuarioInformado*.

Agora precisamos setar o valor que recebemos do extra com a chave *EMAIL* e carregar na variável correspondente ao campo. O código será o seguinte:

```
this.usuarioInformado = (TextView) findViewById(R.id. txtEmailRecebido);
Bundle bundle = getIntent().getExtras();
String recebido = bundle.getString( "EMAIL");
this.usuarioInformado.setText(recebido);
```

- Lembram dos *Maps* em java? O extra funciona como um Mapa.

Falta agora apenas criar um local na tela que receberá esse campo informado e ao qual daremos o nome de *txtEmailRecebido*. Vá ao modo *Design*, na tela principal, e crie um *textView* vazio para receber essa informação ao carregar a tela.

Então, ficou assim a nossa classe *Login.java*:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class login extends Activity {

    private EditText Usuario;
    private EditText Senha;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. login);
        this.Usuario = (EditText) findViewById(R.id. editLogin);
        this.Senha = (EditText) findViewById(R.id. editSenha);
    }

    public void verificaLogin(View v) {
        String txtUsuario = Usuario.getText().toString();
        String txtSenha = Senha.getText().toString();

        if (txtUsuario.equals( "teste@ifb.com") && txtSenha.equals( "123")) {
            Intent intent = new Intent (this, principal.class);
            intent.putExtra( "EMAIL", txtUsuario);
            startActivity(intent);
        }
        else {
            AlertDialog alertDialog = new
AlertDialog.Builder( this).create();
            alertDialog.setTitle( "Erro!");
            alertDialog.setMessage( "Nome de usuário e/ou senha
incorretos!!");
            alertDialog.show();
        }
    }
```

Nossa classe Principal.java ficou da seguinte forma:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class principal extends Activity{

    private TextView usuarioInformado;

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

        this.usuarioInformado = (TextView) findViewById(R.id. txtEmailRecebido);
        Bundle bundle = getIntent().getExtras();
        String recebido = bundle.getString( "EMAIL");
        this.usuarioInformado.setText(recebido);
    }
}
```

E o código de nossas *Activity principal.xml* ficou o seguinte:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="112dp"
        android:src="@mipmap/ic_principal" />

    <TextView
        android:id="@+id/textLogado"
        android:layout_width="match_parent"
        android:layout_height="38dp"
        android:text="Você está logado!"
        android:textAlignment="center"
        android:textColor="@android:color/holo_red_dark"
        android:textSize="20sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/txtEmailRecebido"
        android:layout_width="match_parent"
        android:layout_height="38dp"
        android:textAlignment="center"
        android:textColor="@android:color/holo_red_dark"
        android:textSize="20sp"
        android:textStyle="bold" />
</LinearLayout>
```

Vamos testar nossa aplicação?

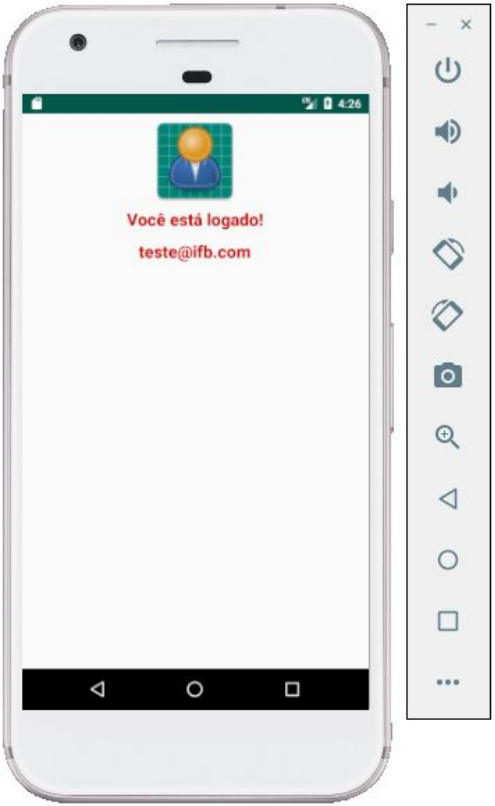


Figura 24: Tela com passagem de email entre Activities.

Pessoal, chegamos ao final da nossa terceira unidade. Aqui criamos uma tela de *login*, trabalhamos chamadas a outras telas, vimos como acionar as caixas de textos *(Dialogs)* e verificamos, ainda, como passar parâmetros entre as *Activities*.

Para sintetizarmos tudo o que estudamos nesta unidade, vamos rever os principais pontos?



Bora rever!

Vimos, no primeiro tópico, aspectos introdutórios dos conceitos de *Activities* e *Intents*. Estudamos que os *Activities* são, de fato, as nossas telas e que são acionadas pelas *intents*, que são nossas ações.

No segundo tópico, trabalhamos mais aspectos dos *layouts*, inclusive, com o uso de imagens.

Por fim, no terceiro tópico, verificamos como fazer chamada de outras telas e passar parâmetros, além da criação de um *Dialog*.

Na próxima unidade, veremos novas funcionalidades que completarão nosso aplicativo de testes, funcionalidades muito úteis entre os aplicativos atuais. Além disso, finalizaremos gerando o nosso *APK* e publicando-o na loja da *Google Play*.

Unidade 4 PUBLICANDO O APLICATIVO

Nesta unidade, geraremos o *apk* de nosso aplicativo e publicaremos na loja. Antes disso, veremos mais algumas funcionalidades importantes que usaremos em nossos aplicativos. Vamos em frente!

4.1 Últimos ajustes da aplicação

Agora, vamos continuar a inserir funcionalidades importantes em nosso aplicativo.

4.1.1 Capturando imagens

A câmera é uma funcionalidade indispensável na maioria dos aplicativos, por isso, iremos ver como utilizá-la em nossa aplicação.

Vamos começar inserindo a imagem em nossa aplicação, na tela de usuário logado, ou seja, nossa tela principal. Utilizaremos a imagem camera_btn, que está localizada em nossa pasta.



Figura 1: Imagem que utilizaremos como menu para câmera em nosso aplicativo

Para isso, faça um Crtl + C na imagem e, na pasta *drawable* do projeto (no *Android Studio*), faça um Crtl + V.

Clique em *OK* na tela de seleção do destino.

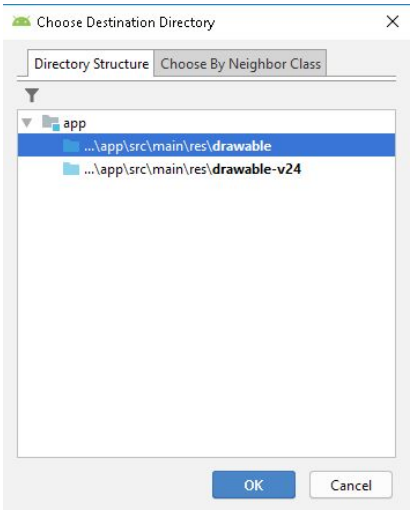


Figura 2: Tela de destino da imagem copiada

Clique em *OK* na tela de diretório da cópia.

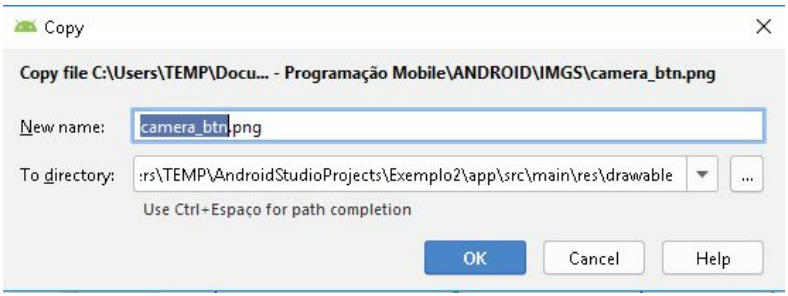


Figura 3: Seleção de diretório da cópia

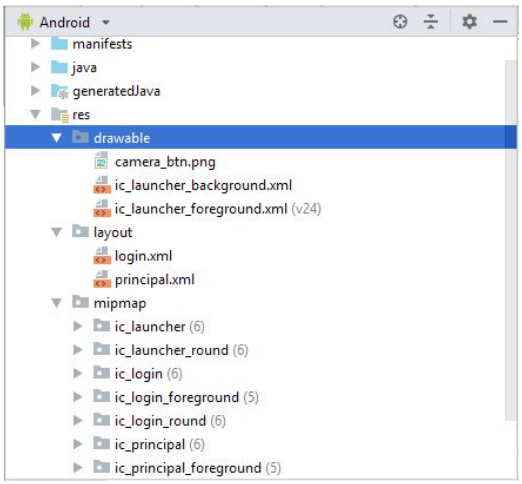


Figura 4: Tela com a imagem copiada para o projeto

Agora, vamos criar um botão com imagem. Utilizaremos o componente *ImagemButton*.No modo *Design* da tela Principal, clique em *ImageButton* e arraste para embaixo da tela.

Selecione a imagem câmera_btn na tela de seleção seguinte, que estará dentro de *Project*:

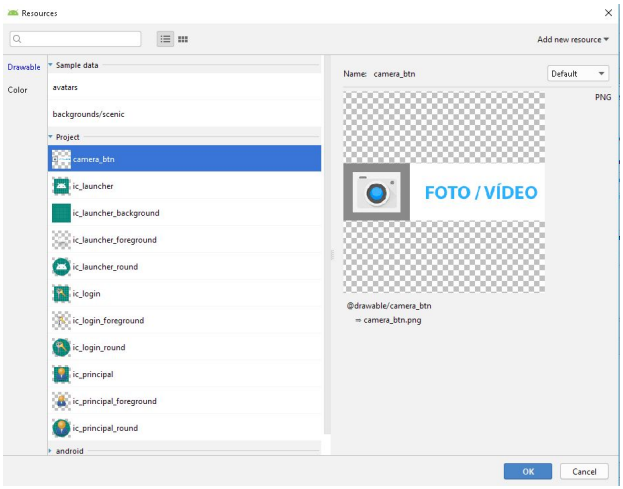


Figura 5: Selecionando a imagem do *menu*

E assim ficará em nosso protótipo:



Figura 6: Protótipo da tela principal

Mude o *ID* para *btnCamera*, na aba de atributos (com o botão da câmera selecionado):

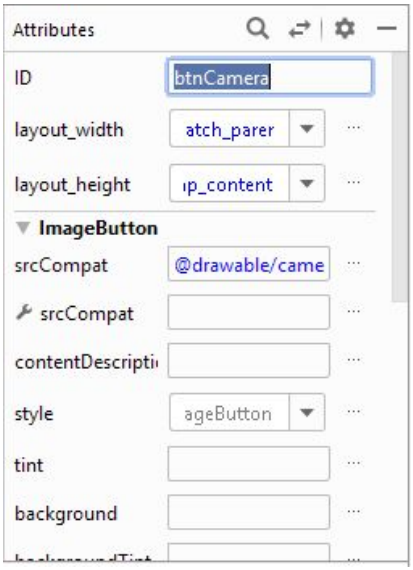


Figura 7: Atributos do botão de Câmera

Precisaremos agora criar um novo *layout* que será o de uso da câmera. Ele será chamado ao clicar em câmera.

- Já sabemos como fazer para criar um novo *layout*, não é mesmo?

- Sim, professor! Clicar com o direito na pasta layout, depois New >> XML >> Layout XML File.

- Isso mesmo, pessoal! Chamaremos o *layout* de “câmera”.

- OK, professor!

Ao criar, abra o *camera.xml* no modo *Design*. Defina a orientação como vertical e centralize a tela com a propriedade *gravity*, atribuindo o valor *center*.

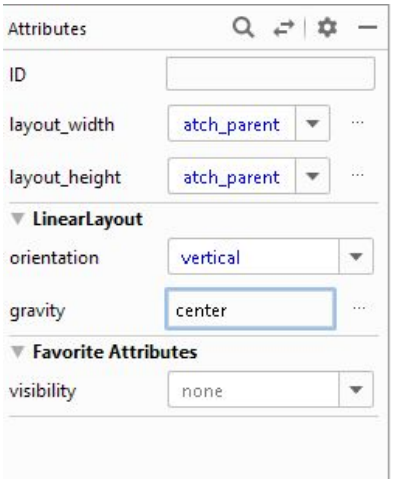


Figura 8: Orientação da tela como vertical e *gravity* como *center*

Vamos agora importar a imagem “foto_btn.png”. Façamos da mesma forma que fizemos com a imagem anterior. Crtl + C na imagem foto_btn.png e Crtl + V na pasta *drawable*.

Agora, na Activity camera.xml, crie uma *ImageButton*, selecionando a imagem foto_btn da pasta *Project*.



Figura 9: Tela de foto

Agora, precisaremos criar nossa classe Java para a tela de câmera.

- Sabemos como fazer isso não é mesmo?
- Sim, professor. Na pasta java, New >> JavaClass, com o nome câmera.
- Perfeito!

A nova classe câmera deve estender *Activity* e setar qual *activity* será chamado. O código ficará da seguinte forma:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.os.Bundle;

public class camera extends Activity {

    @Override
    public void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.camera);

    }
}
```

Não se esqueçam de resolver os *imports* da classe com Alt + enter.

- Pronto, pessoal?
- Não, professor. Falta setar a activity no AndroidManifest.xml!
- Excelente, pessoal! Vocês sabem tudo!

E nosso *manifest* ficará da seguinte forma:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ifb.exemplo2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >


        <activity android:name=".login" >
            <intent-filter> <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

        <activity android:name=".principal" />
        <activity android:name=".camera" />

    </application>
</manifest>
```

Certo, pessoal. Vamos agora fazer com que, ao clicar no botão Foto/Vídeo, seja chamada a *activity* camera.xml, que será a tela na qual capturaremos nossas fotos.

Já fizemos isso. Faça o seguinte agora: crie um método executaFoto que chame a tela de foto. Pare aqui a leitura e tente fazer sozinho. Depois de tentar, volte aqui para conferir como fizemos. *OK*?

(... Voltando após a pausa para criação)

- Conseguiram? Tenho certeza que sim! Vamos em frente!

Crie um método executaFoto na classe principal.java. Ele terá o seguinte código:

```
public void executaFoto(View v) {
    Intent intent = new Intent(this,
camera.class);
    startActivity(intent);
}
```

Agora precisamos chamar esse método ao clicar no botão. Tenho certeza de que vocês fizeram isso também. Lá no principal.xml no *imageButton btnCamera*, adicione a linha android:onClick="executaFoto". O código da tela principal.xml, então, ficará da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="112dp"
        android:src="@mipmap/ic_principal" />

    <TextView
        android:id="@+id/textLogado"
        android:layout_width="match_parent"
        android:layout_height="38dp"
        android:text="Você está logado!"
        android:textAlignment="center"
        android:textColor="@android:color/holo_red_dark"
        android:textSize="20sp"
        android:textStyle="bold" />
```

```
android:layout_width="match_parent"
        android:layout_height="38dp"
        android:textAlignment="center"

android:textColor="@android:color/holo_red_dark
"
        android:textSize="20sp"
        android:textStyle="bold" />

    <ImageButton
        android:id="@+id/btnCamera"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:srcCompat="@drawable/camera_btn"
        android:onClick="executaFoto"/>
</LinearLayout>
```

Vamos agora parar um pouco e testar nosso aplicativo!

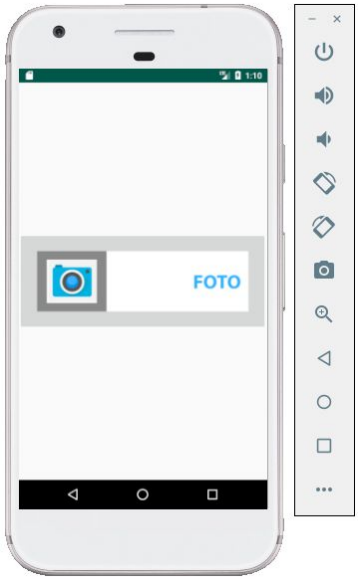


Figura 10: Tela da *câmera.xml*

Funcionou, pessoal!

Agora vamos fazer com que ao clicar em Foto, seja acionado o *Hardware* do dispositivo para tirar uma foto.

Para isso, começaremos criando duas variáveis na classe camera.java. Uma para controle, caso seja do tipo de captura, isto é, se é for foto ou vídeo. Esta classe será denominada CAPTURAR_IMAGEM. E outra que terá o caminho de armazenamento da imagem no cartão SD, que chamaremos de uri. Isso será feito da seguinte forma:

```
      private static final int CAPTURAR_IMAGEM =
1;
      private Uri uri;
```

Teremos agora que criar um método para capturar imagem. Ao clicar no botão foto, chamaremos o método capturarImagem. Ele terá o seguinte código:

```
public void capturarImagem(View v){
    boolean temCamera =
getPackageManager().hasSystemFeature(PackageManagerEEATURE_CAMERA);

    if(temCamera){
        File diretorio =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);

        String nomeImagem = diretorio.getPath() +"/" + System.currentTimeMillis()
+ ".jpg";

        uri = Uri.fromFile(new File(nomeImagem));
        Intent intent =new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

        startActivityForResult(intent,CAPTURAR_IMAGEM);
    }
}
```

Vamos entender esse método. Primeiramente, iremos verificar se o dispositivo tem câmera com o método *hasSystemFeature* passando a Feature de câmera como parâmetro e o resultado será armazenado na variável *temCamera*.

Depois, caso tenha câmera, armazenaremos o diretório padrão de imagem na variável diretório.

Depois criaremos um nome para nossa imagem. Mas para que não seja toda hora sobreposta à imagem, utilizaremos o horário corrente com milissegundos, o que torna quase impossível a sobreposição de imagens.

Isso será armazenado em uma Uri com o nome do arquivo.

Então, será chamada uma *Intent* do tipo *MediaStore.ACTION_IMAGE_CAPTURE* que acionará o hardware da câmera, enviando também o nome da imagem, pois queremos que seja salvo com o nome criado. Esse nome já está setado na variável da classe.

E, por fim, será iniciada uma nova *activity*, chamando uma *activityForResult*, isto é, uma *activity* por resultado passando uma foto e não um vídeo. Em outra palavras, passaremos a variável de controle que criamos anteriormente, se ela tiver o valor CAPTURAR_IMAGEM (que no nosso caso é 1), saberemos que o método por resultado chamará a câmera.

Vamos criar o método *onActivityResult* e vocês vão entender melhor. Esse método, também na classe *camera.java* terá o seguinte código:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == CAPTURAR_IMAGEM) {
        if (resultCode == RESULT_OK) {
            mostrarMensagem('Imagem capturada!");
            adicionarNaGaleria();
        } else {
            mostrarMensagem('Imagem não capturada!");
        }
    }
}
```

Percebam que estamos testando a captura da imagem e a efetividade do código de resultado. Então, chamaremos dois métodos que são o mostrar mensagem e o adicionar na galeria. Este último, caso tenhamos tido sucesso na captura da imagem. Vamos logo criar também esses métodos:

```
private void mostrarMensagem(String msg){
    Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
}

private void adicionarNaGaleria() {
    Intent intent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    intent.setData(uri);
    this.sendBroadcast(intent);
}
```

Para o mostrarMensagem, utilizaremos um *Toast* que são mensagens exibidas no próprio *Intent* (próprio *layout*). São mensagens curtas exibidas dentro de um retângulo com bordas arredondadas na tela. Não tem interação nenhuma com o usuário.

Para o adicionarNaGaleria, usaremos uma nova *intent* do tipo *ACTION_MEDIA_SCANNER_SCAN_FILE*, que adicionará a imagem na galeria.

Nossa classe *camera.java,* então, ficará da seguinte forma:

```
package com.ifb.exemplo2;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import android.view.View;
import android.widget.Toast;

import java.io.File;

public class camera extends Activity {

    private static final int CAPTURAR_IMAGEM = 1;
    private Uri uri;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.camera);

    }

    public void capturarImagem(View v){
        boolean temCamera =
getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA);

        if(temCamera){
            File diretorio =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);

            String nomeImagem = diretorio.getPath() +"/" + System.currentTimeMillis() +
".jpg";

            uri = Uri.fromFile(new File(nomeImagem));
            Intent intent =new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

            startActivityForResult(intent,CAPTURAR_IMAGEM);
        }
    }
```

```
    @Override
    protected void onActivityResult(int requestCode, int
resultCode, Intent data) {
        if (requestCode == CAPTURAR_IMAGEM) {
            if (resultCode == RESULT_OK) {
                mostrarMensagem("Imagem capturada!");
                adicionarNaGaleria();
            } else {
                mostrarMensagem("Imagem não capturada!");
            }
        }
    }

    private void mostrarMensagem(String msg){
        Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
    }

    private void adicionarNaGaleria() {
        Intent intent = new
Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
        intent.setData(uri);
        this.sendBroadcast(intent);
    }


}
```

Agora, então, só nos resta acionar o método ao clicar no botão Foto da tela *camera.xml*. Podemos fazer isso em modo *text* ou *Design*. Dessa forma, vamos alterar pelo segundo modo dessa vez, clicando no botão Foto e indo a suas propriedades:

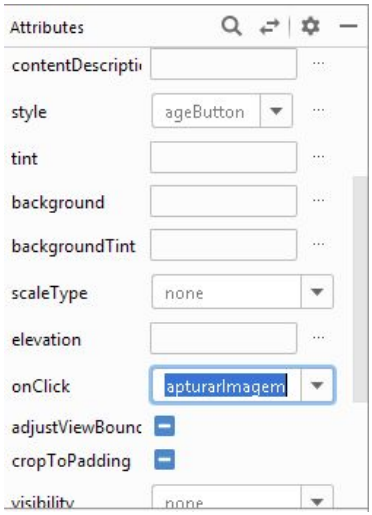


Figura 11: Alterando a propriedade *onClick*

4.1.2 Capturando vídeos

Vamos, agora, fazer a mesma coisa para criação de um menu de vídeos. Usaremos a imagem vídeo.png. Copie a imagem para a pasta *drawable*.

Nossa tela ficará da seguinte forma:

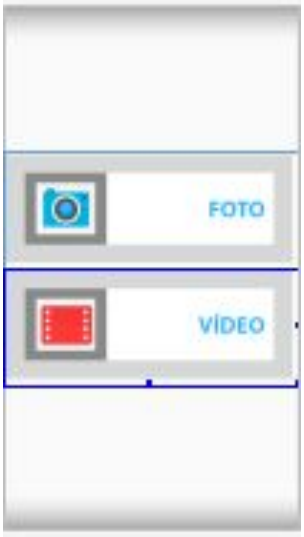


Figura 12: Protótipo com vídeo

O código da tela *camera.xml* ficará da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/and
roid"

xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <ImageButton
        android:id="@+id/btnCapturaFoto"
        android:layout_width="match_parent"
        android:layout_height="162dp"
        android:onClick="capturarImagem"
        android:src="@drawable/foto_btn" />

    <ImageButton
        android:id="@+id/btnCapturaVideo"
        android:layout_width="match_parent"
        android:layout_height="162dp"
        android:onClick="capturarVideo"
        android:src="@drawable/video_btn" />
</LinearLayout>
```

Precisaremos agora criar e alterar a classe *camera.java*.

A primeira coisa a fazer é criar uma variável da seguinte forma:

```
private static final int CAPTURAR_VIDEO = 2;
```

Criaremos agora um método *capturaVideo* com o código a seguir:

```
public void capturarVideo(View v){
    boolean temCamera =
getPackageManager().hasSystemFeature(PackageManagerEEATURE_CAMERA);

    if(temCamera){
        File diretorio =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
        String nomeVideo = diretorio.getPath() +"/" +
System.currentTimeMillis() + ".mp4";
        uri = Uri.fromFile(new File(nomeVideo));

        Intent intent =new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
        intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);
        intent.putExtra(MediaStore.EXTRA_DURATION_LIMIT, 5);
        startActivityForResult(intent,CAPTURAR_VIDEO);
    }

}
```

Alteraremos nosso método *onActivityResult* da seguinte forma:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if(data != null) {
        if (requestCode == CAPTURAR_IMAGEM) {
            if (resultCode == RESULT_OK) {
                mostrarMensagem("Imagem capturada!");
                adicionarNaGaleria();
            } else {
                mostrarMensagem("Imagem não capturada!"); } }

        else if(requestCode == CAPTURAR_VIDEO) {
            if (resultCode == RESULT_OK) {
                mostrarMensagem("Vídeo gravado com sucesso!");
                adicionarNaGaleria();
            }
            else { mostrarMensagem("Vídeo não gravado!"); }
        }
    }
}
```

Desse modo, nossa *camera.java* ficou da seguinte forma:

```
package com.ifb.exemplo2;

import android.Manifest;
import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.FileProvider;
import android.util.Log;
import android.view.View;
import android.widget.Toast;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class camera extends Activity {

    private static final int CAPTURAR_IMAGEM = 1;
    private static final int CAPTURAR_VIDEO = 2;
    private Uri uri;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.camera);

    }
```

```
public void capturarImagem(View v){
        boolean temCamera =
getPackageManager().hasSystemFeature(PackageManagerEEATURE_CAMERA);

        if(temCamera){
            File diretorio =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
            String nomeImagem = diretorio.getPath() +"/" +
System.currentTimeMillis() + ".jpg";
           uri = Uri.fromFile(new File(nomeImagem));
            Intent intent =new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

            startActivityForResult(intent,CAPTURAR_IMAGEM);
        }
    }

    public void capturarVideo(View v){
        boolean temCamera =
getPackageManager().hasSystemFeature(PackageManagerEEATURE_CAMERA);

        if(temCamera){
            File diretorio =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
            String nomeVideo = diretorio.getPath() +"/" +
System.currentTimeMillis() + ".mp4";
           uri = Uri.fromFile(new File(nomeVideo));

            Intent intent =new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
            intent.putExtra(MediaStoreEXTRA_VIDEO_QUALITY, 1);
            intent.putExtra(MediaStoreEXTRA_DURATION_LIMIT, 5);
            startActivityForResult(intent,CAPTURAR_VIDEO);
        }

    }
```

```
@Override
    protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
        if(data != null) {
            if (requestCode == CAPTURAR_IMAGEM) {
                if (resultCode == RESULT_OK) {
                    mostrarMensagem("Imagem capturada!");
                    adicionarNaGaleria();
                } else {
                    mostrarMensagem("Imagem não capturada!"); } }

            else if(requestCode == CAPTURAR_VIDEO) {
                if (resultCode == RESULT_OK) {
                    mostrarMensagem("Vídeo gravado com sucesso!");
                    adicionarNaGaleria();
                }
                else { mostrarMensagem("Vídeo não gravado!"); }
            }
        }
    }

    private void mostrarMensagem(String msg){
        Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
    }

    private void adicionarNaGaleria() {
        Intent intent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
        intent.setData(uri);
        this.sendBroadcast(intent);
    }


}
```

Vamos testar nossa aplicação!

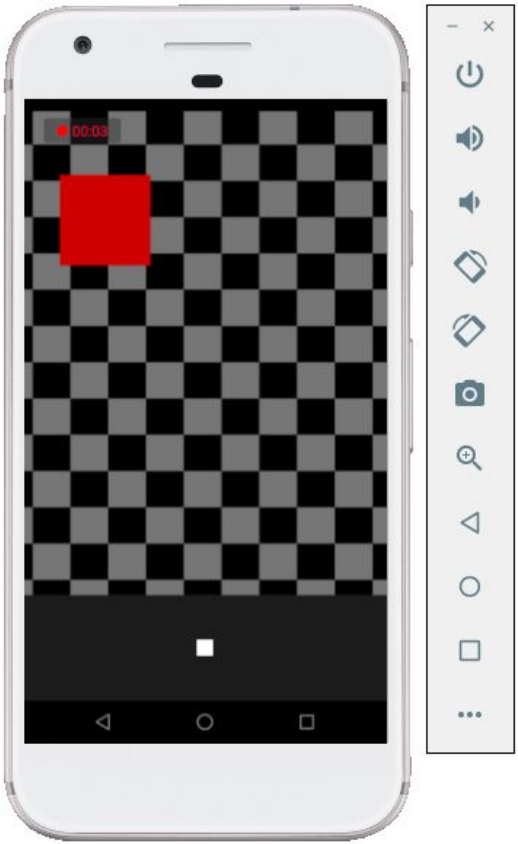


Figura 13: Funcionando o vídeo!

Pessoal, muitos outros recursos ainda poderiam ser inseridos em nosso aplicativo, entretanto, devido à carga horária do curso, não poderemos inserir todos, mas acredito que tivemos uma boa noção de uso dos recursos. Vamos para o próximo tópico!

4.2 Gerando o aplicativo instalável

Vamos agora gerar um instalável de nosso aplicativo. Vamos juntos!

4.2.1 Configurando o ícone da aplicação

Pessoal, vocês perceberam que, até agora, estávamos usando o ícone padrão do *android*. Vamos agora setar o ícone de nossa aplicação.

Para isso, criaremos um novo *ImageAsset* com a imagem *icon.png*. conforme a figura abaixo:

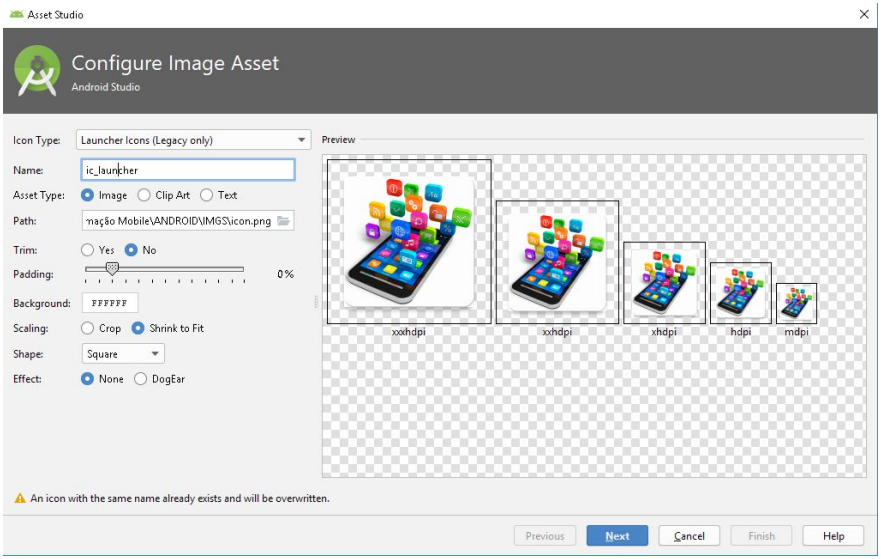


Figura 14: Configuração do ícone da aplicação

Criaremos com o nome *ic_launcher* para substituir o anterior. Clique em *Next*.

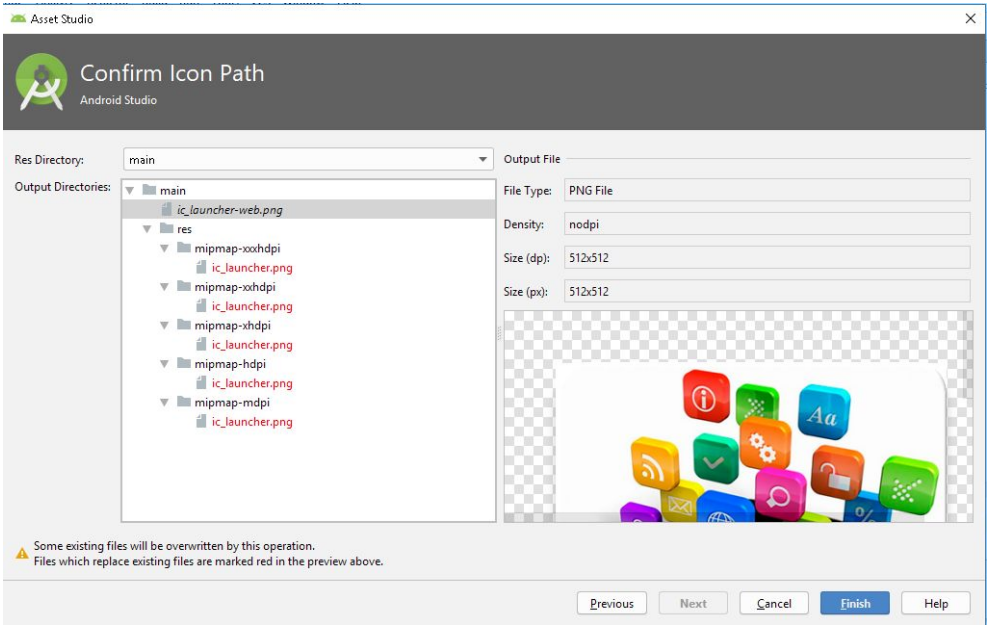


Figura 15: Tela informando da substituição dos arquivos

Clique em *Finish*, e observe que o ícone de nossa aplicação Exemplo2 foi alterado.

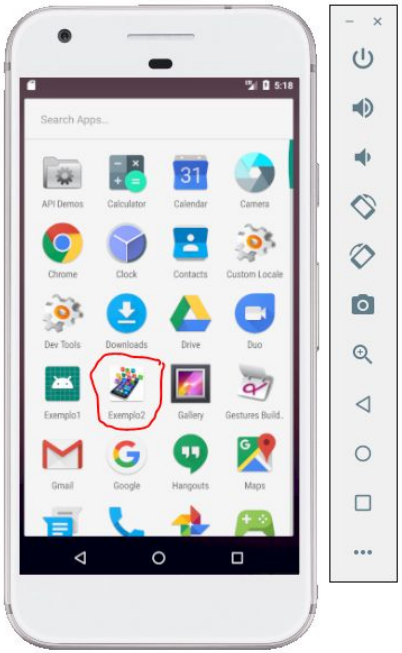


Figura 16: Ícone alterado do aplicativo

4.2.2 Gerando o *APK*

Para gerar nosso aplicativo para vendê-lo no *Google Play*, é necessário gerar um aplicativo certificado e alinhado.

Vamos começar gerando o *Apk* certificado.
Vamos no menu *Build* >> *Generated Signed Bundle / APK*...
Selecione *Android App Bundle* e clique em Next.

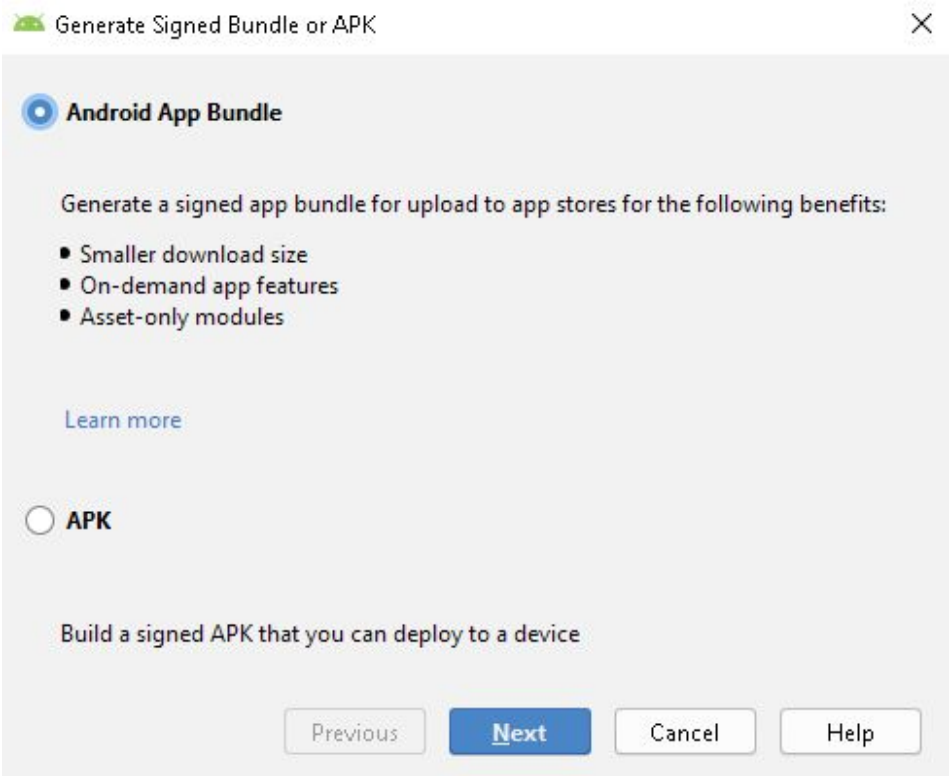


Figura 17: Seleção do tipo de geração de *apk*

Na próxima tela, clique em *New* para gerar um novo certificado.

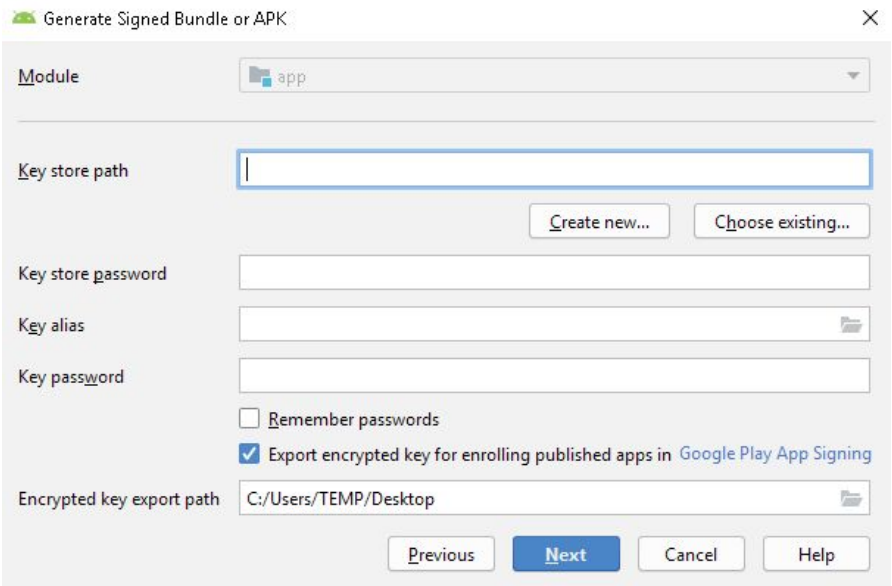


Figura 18: Gerando um novo certificado

Na próxima tela, configuraremos nosso certificado.

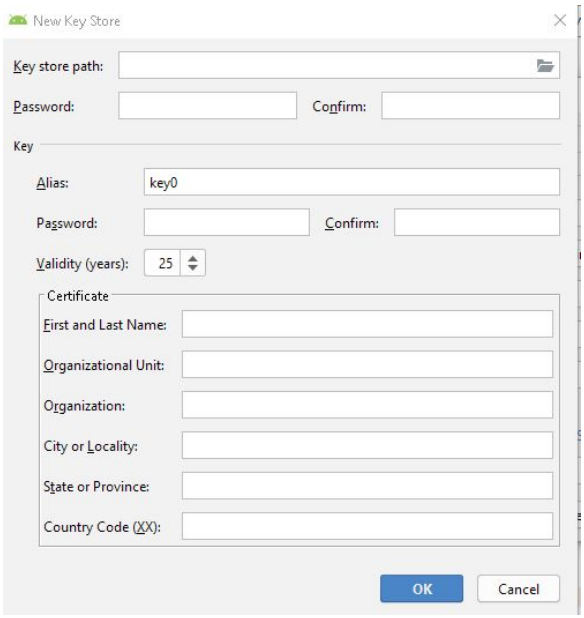


Figura 19: Gerando um novo certificado

Na configuração, defina uma senha (utilizar 123456 sempre), os demais dados de validade e o responsável pelo aplicativo. O código do país será sempre BR (Brasil com duas letras).

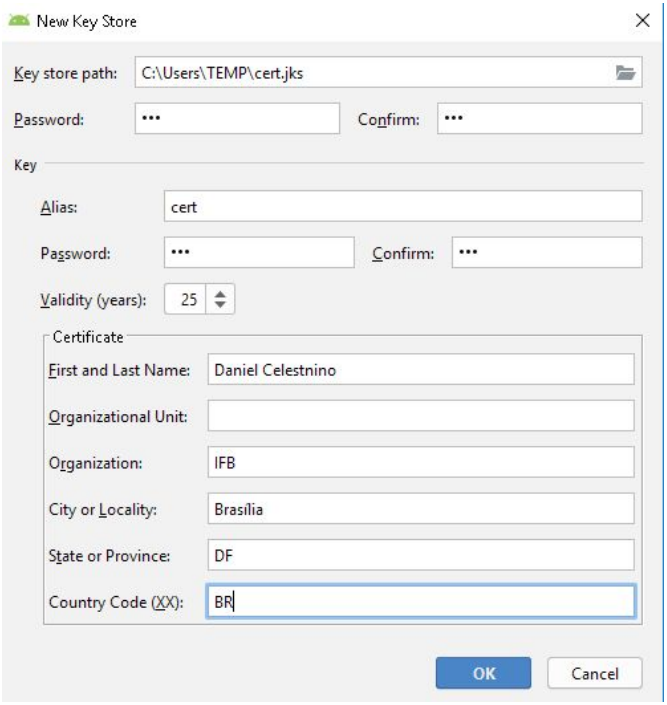


Figura 20: Configuração do certificado

Perceba que ao clicar em *OK*, nosso certificado foi criado:

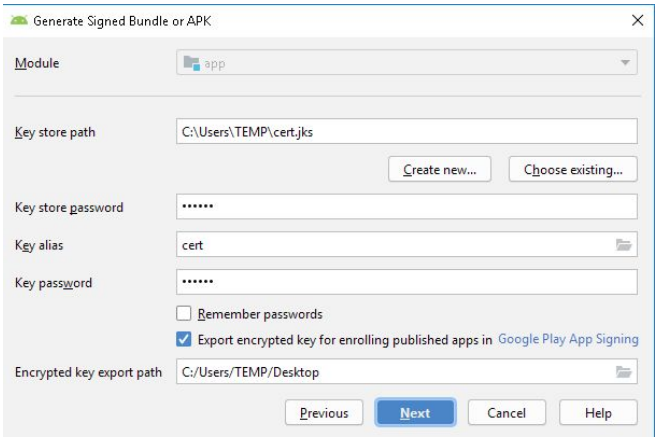


Figura 21: Tela com certificado criado

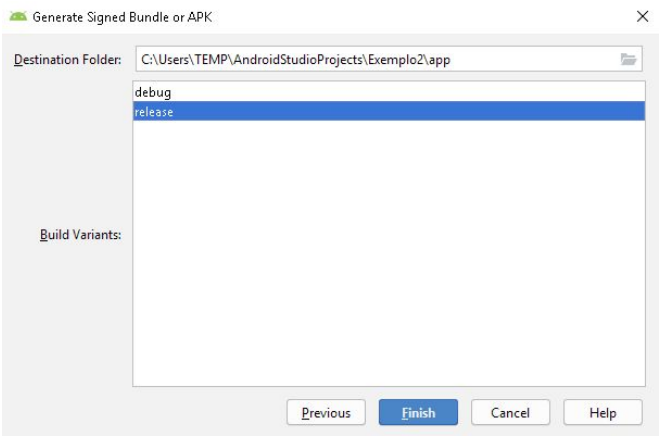


Figura 22: Tipo de geração de *apk*

Nessa tela, verifique onde será gerado o *APK* e selecione o tipo release, que é a versão pronta para ir à loja. Em seguida, clique em *Finish*.

Rode novamente o gerador mas agora selecionando a opção *APK*.

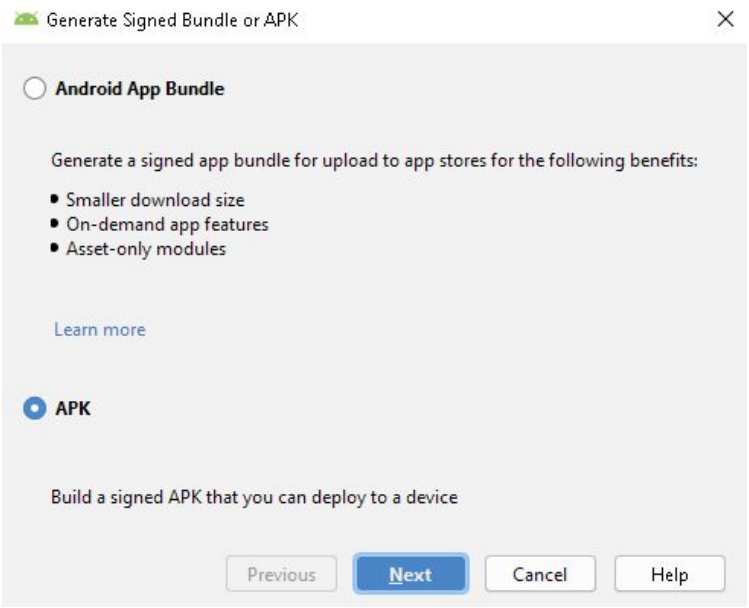


Figura 23: Gerando APK assinado

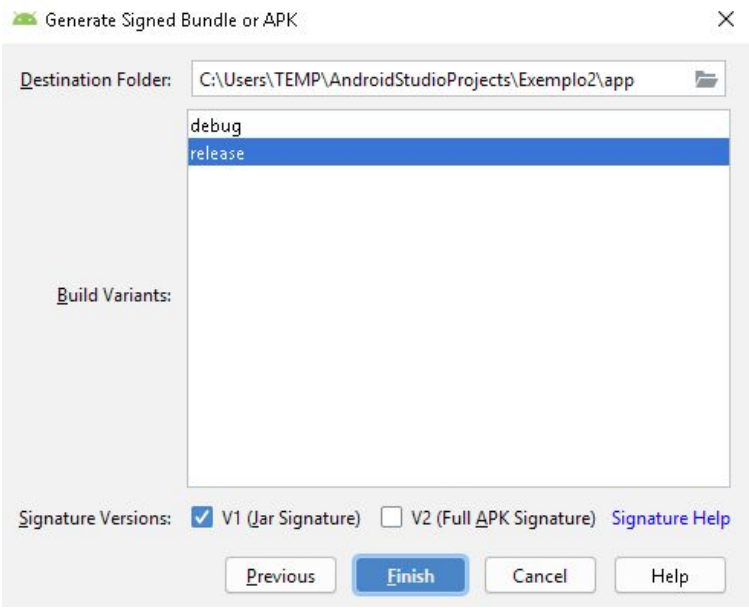


Figura 24: Gerando em modo release

Você verá que foi criada uma pasta *release* e terá um arquivo com extensão *.apk* no diretório. Provavelmente o nome será *app-relase.apk*.

Agora, precisaremos alinhar o aplicativo. Para isso, usaremos o aplicativo Zipalign, que você pode baixar do site https://forum.xda-developers.com/showthread.php?t=1385846.

Então, após baixar os arquivos, descompacte e cole o *apk* dentro da pasta do *ZipAlign*.

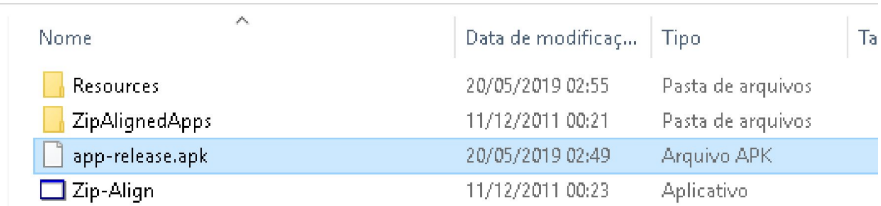


Figura 25: *ZipAlign*

Agora execute o *Zip-Align*.

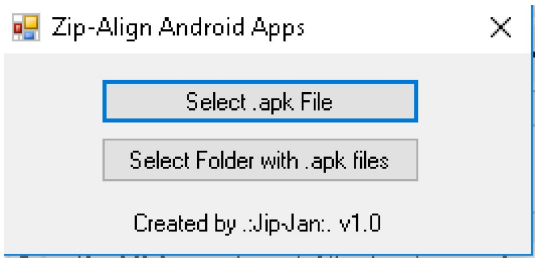
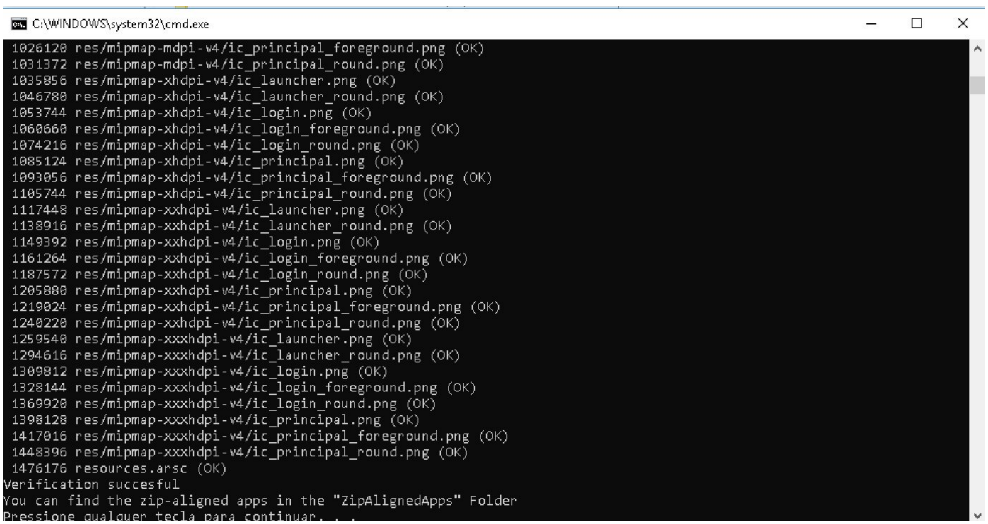


Figura 26: Seleção do *apk* no *ZipAlign*

Selecione o *apk* colado na pasta. Clique em sim na pergunta do *bat*. Agora selecione o *bat ZipAlign* e será chamado um *.bat* que assinará o aplicativo.

Se olharmos na pasta *ZipAlignedApps*, nossa aplicação estará lá.

Assim, finalizamos esse tópico com o *apk* pronto para ser publicado.



4.3 Publicando aplicativos nas lojas

Publicaremos o *APK* na loja. Vamos em frente!

4.3.1 Publicando o aplicativo

Pessoal, para fechar nosso estudo, vamos agora falar sobre como funciona a publicação nas lojas de aplicativos. Nosso foco maior continuará sendo a *Google* (sistema operacional Android) para efeito de aprendizagem, mas falaremos um pouco sobre as outras lojas.

Para publicação na *Google Play*, essa ação é feita pelo endereço: https://play.google.com/apps/publish. Será preciso pagar uma taxa de registro de $ 25 dólares. Após feito isso, aparecerá uma tela semelhante a esta:

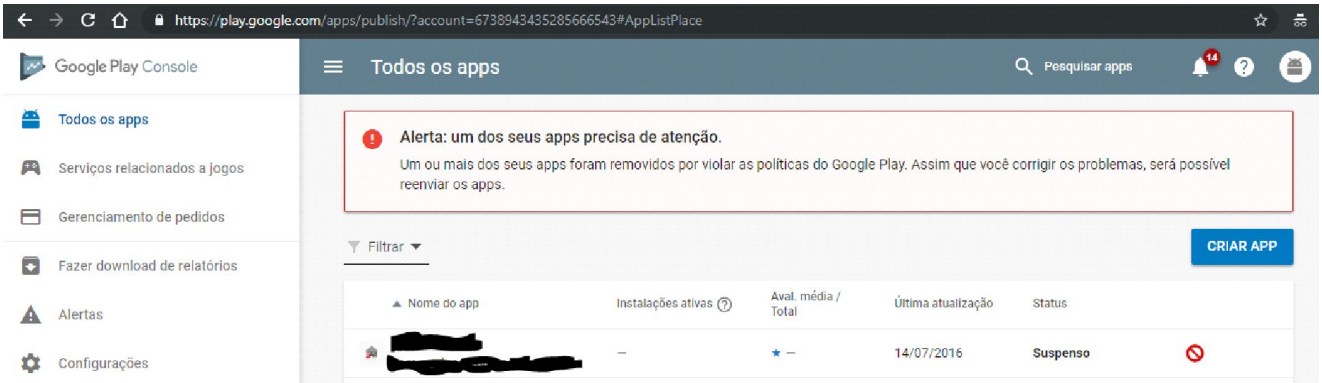


Figura 27: Tela de gerenciamento da *Google Play*

Esse *print* foi tirado de uma conta institucional que possui diversos aplicativos publicados. Logicamente, terei que ocultar alguns detalhes das informações por questões de sigilo.

Para criar um novo aplicativo, basta clicar em "*Criar App*".

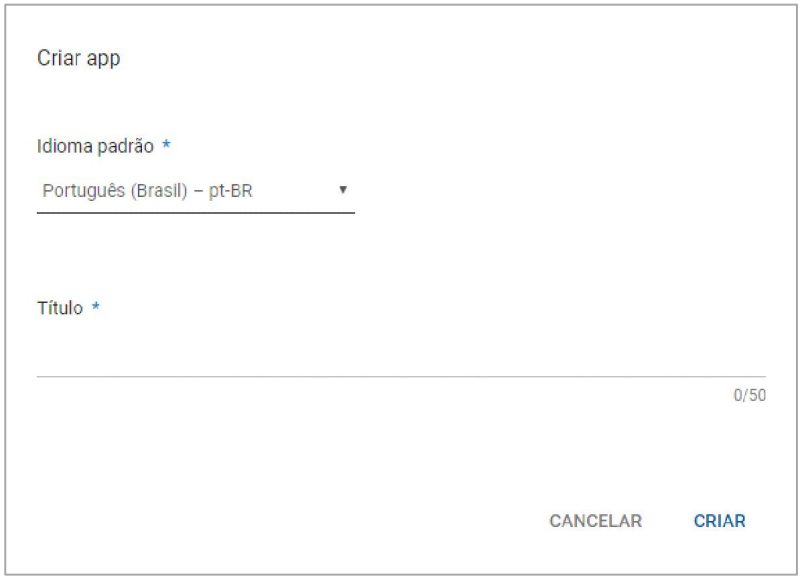


Figura 28: Criando *App*

Depois disso, você entrará numa tela para informar uma breve descrição e uma descrição completa, isso para cada idioma que você escolher dar suporte. Além disso, é preciso subir um ícone em alta resolução, um gráfico de recursos e, pelo menos, dois *screenshots* da *app*. E, a cada nova publicação de versão, você deve informar o que foi alterado.

A publicação demora algumas horas para estar disponível na loja.

-Legal, professor. Tem como ficar acompanhando o aplicativo? Saber quantos *downloads*? Avaliações?

- Tem sim!

A *Google Play* fornece as seguintes opções de acompanhamentos por aplicativo.

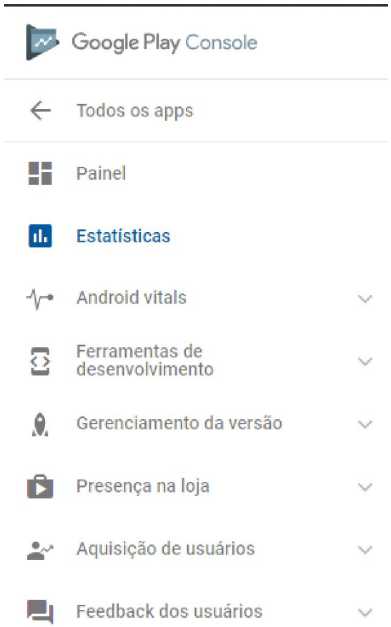


Figura 29: Opções da *GooglePlay*

Entre as principais informações disponíveis, podemos destacar as instalações ativas e as desinstalações, as avaliações (você ainda pode responder comentários), dentre outras.



Figura 30: Dados estatísticos da *GooglePlay*

Pessoal, chegamos ao final de nosso estudo. Aqui criamos mais alguns recursos como manipulação de imagens, aprendemos como gerar nosso *apk* e a publicá-lo na loja.

Para sintetizarmos tudo o que estudamos nesta quarta unidade vamos rever os principais pontos?



Bora rever!

Vimos, no primeiro tópico, a finalização de nossa aplicação, aprendendo novos recursos como manipulação de imagens e vídeos.

No segundo tópico, verificamos como setar o ícone da aplicação e gerar um *apk* certificado.

Por fim, no terceiro tópico, vimos como publicar nosso *apk* na loja.

Foi um prazer poder compartilhar esses conhecimentos com vocês! Espero que tenham gostado!

Até a próxima!

Referências

Bibliografia Básica

ALLEN, Sarah; GRAUPERA, Vidal; LUNDRIGAN, Lee. *Desenvolvimento Profissional Multiplataforma para Smartphone*: iPhone, Android, Windows Mobile e BlackBerry. São Paulo: Alta Books, 2012.

DARIVA, Roberto. *Gerenciamento de Dispositivos Móveis e Serviços de Telecom* - Estratégias de Marketing, Mobilidade e Comunicação. Rio de Janeiro: Campus, 2011.

NEIL, Tereza. *Padrões de Design para Aplicativos Móveis*. Rio de Janeiro: Novatec, 2012.

STARK, Jonathan; JEPSON, Brian. *Construindo Aplicativos Android com HTML, CSS e JavaScript*: Criando Aplicativos Nativos com Ferramentas Baseadas nos Padrões Web. Rio de Janeiro: Novatec, 2013.

Bibliografia Complementar

ARAÚJO, Everton Coimbra. *Desenvolvimento para Web com Java*. São Paulo: visual Books, 2010.

DEITEL, Harvey. M.; DEITEL, Paul .J. *Java: como programar*. 8. ed. São Paulo: Pearson- Prentice Hall, 2010.

LECHETA, Ricardo R. Google *Android* - Aprenda A Criar Aplicações Para Dispositivos Móveis Com o Android Sdk. 3. ed. Rio de Janeiro: Novatec, 2013.

MARINACCI, Joshua. *Construindo Aplicativos Móveis com Java*. Rio de Janeiro: Novatec, 2012.

MELO, Alexandre Altair de; LUCKOW, Décio Heinzelmann. *Programação Java para a Web*. São Paulo: Novatec, 2010.



Currículo do Professor Autor

Daniel Celestino de Freitas Pereira

Formado em Ciência da Computação pela Universidade Católica de Brasília (2007) e Pós-graduado em Governança de TI pelo Centro Universitário Unieuro (2010). Atuou, por muito tempo, como profissional de tecnologia na área de desenvolvimento de *Software*, trabalhando principalmente como programador e projetista de sistemas.

Em se tratando de Administração Pública, foi funcionário – como Analista Pleno – da Caixa Econômica Federal, na área de TI em 2009. Foi aprovado e nomeado no SERPRO em 2010 para a área de Desenvolvimento de *Software* e para Analista de Negócio. Ainda em 2010, tomou posse como servidor do Departamento de Trânsito do Distrito Federal (DETRAN/DF), no Cargo de Analista de Trânsito na especialidade de Tecnologia da Informação.

No DETRAN/DF, já ocupou diversos cargos gerenciais destacando-se o de Gerente da Gerência de Sistemas, Auditoria e Governança (GERSAG) e de Diretor (CIO – *Chief Information Officer*) da Diretoria de Tecnologia da Informação e Comunicação (DIRTEC). Foi cedido para a Secretaria de Estado de Planejamento, Orçamento e Gestão do Distrito Federal (SEPLAG/DF), desde 2013, onde permanece até o presente momento. Atualmente ocupa o cargo de Coordenador de Modernização de Serviços de TIC na Subsecretaria de Tecnologia da Informação e Comunicação (SUTIC).

Na área educacional, desde 2014, atua como professor conteudista e mediador de disciplinas virtuais em cursos de pós-graduação, além de orientador em Trabalhos de Conclusão de Cursos na Faculdade Unyleya.

Endereço para o CV Lattes: http://lattes.cnpq.br/9189385954807967

MÓDULO II

ARQUITETURA DE SISTEMAS MOBILE

Ronald Emerson Scherolt da Costa

Apresentação

O Módulo II desta obra, Arquitetura de Sistemas *Mobile*, busca despertar o interesse e a capacidade em conhecer a arquitetura e os conceitos de sistemas operacionais (SO) para *Desktop* e dispositivos *Mobile*, assim como articular os conhecimentos a respeito do funcionamento dos seus módulos de gerência.

Compreender o correto funcionamento dos módulos de gerência de um sistema operacional irão contribuir para melhorar a gestão dessas plataformas, como também, permitirão melhorar o desenvolvimento de sistemas e aplicativos para esses dispositivos.

A estrutura e o funcionamento de um SO são tópicos bastante abrangentes e extremamente importantes. Um SO não é executado como uma aplicação sequencial, com início, meio e fim. Suas rotinas são executadas sem uma ordem predefinida e compreender esse processo é importante tanto para um técnico, um administrador de sistemas ou para um desenvolvedor.

O módulo está estruturado em 4 unidades temáticas com tópicos específicos, conforme descrito abaixo.

Unidade 1: será abordada a introdução aos sistemas operacionais e sua história.

Unidade 2: serão tratados os conceitos básicos que envolvem o funcionamento dos sistemas operacionais, sua gerência interna e as suas funcionalidades.

Unidade 3: você irá aprender a articular os conhecimentos de sistemas operacionais, relacionando-os com os sistemas operacionais para dispositivos móveis.

Unidade 4: você iniciará uma jornada para conhecer o conceito de máquinas virtuais e a sua aplicação para dispositivos móveis.

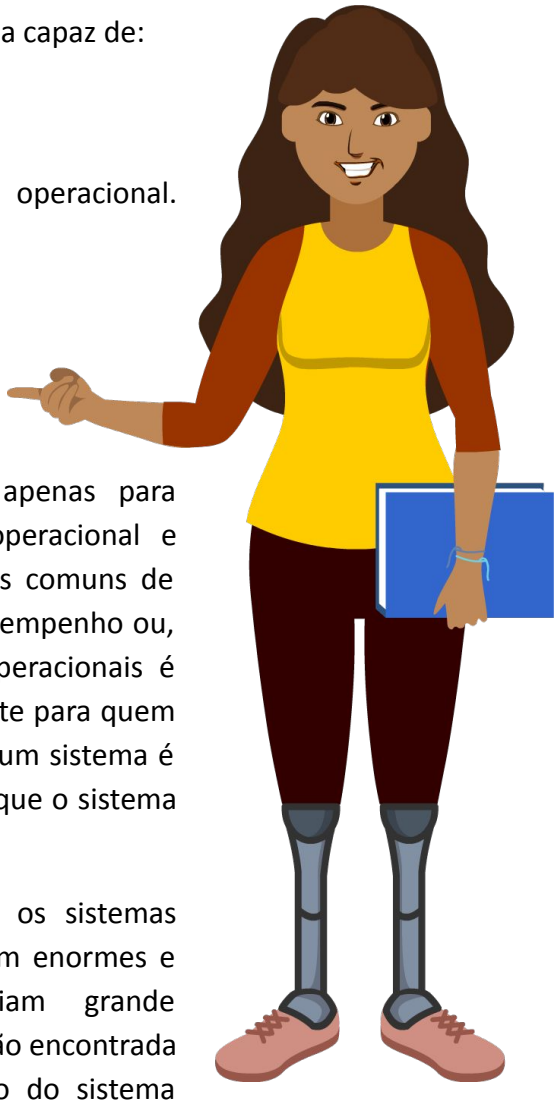
Vamos começar?

Unidade 1 SISTEMAS OPERACIONAIS - HISTÓRIA E PLATAFORMAS

Objetivos de aprendizagem

Esperamos que, ao final do estudo desta unidade você seja capaz de:

- Operar sistemas operacionais;
- Compreender a arquitetura de Sistemas Operacionais;
- Distinguir os módulos de gerência de um sistema operacional.



1.1 Introdução aos Sistemas Operacionais

Estudar os sistemas operacionais é importante não apenas para compreender os mecanismos que compõem um sistema operacional e permitir a execução de tarefas, mas também para evitar erros comuns de programação, que podem resultar em programas com baixo desempenho ou, até mesmo, a perda de informações. Estudar os sistemas operacionais é importante, também, pelo aspecto da segurança e principalmente para quem irá administrar os sistemas computacionais, pois para defender um sistema é necessário saber quais as vulnerabilidades e as possíveis falhas que o sistema possui.

Começaremos o nosso caminho de aprendizado sobre os sistemas operacionais relembrando que os primeiros computadores eram enormes e possuíam uma programação bastante complexa. Exigiam grande conhecimento de *hardware* e de linguagem de máquina. A solução encontrada foi a criação de uma camada entre o hardware e o usuário do sistema computacional.

Essa camada foi denominada Sistema Operacional e serviu para o encapsulamento das interfaces de *hardware*. Dessa forma, a interação com o computador se tornou mais fácil, confiável e eficiente.

De acordo com Machado (2011), "o Sistema Operacional tem por objetivo funcionar como uma interface entre o usuário e o computador, tornando sua utilização mais simples, rápida e segura".

Neste mesmo alinhamento, Maziero (2017, p. 1) define um sistema computacional e um sistema operacional da seguinte forma:

> "um sistema de computação é constituído basicamente por *hardware* e *software*. O *hardware* é composto por circuitos eletrônicos (processador, memória, portas de entrada/saída, etc.) e periféricos eletro-óptico-mecânicos (teclados, mouses, discos rígidos, unidades de disquete, CD ou DVD, dispositivos USB, etc.). Por sua vez, o software de aplicação é representado por programas destinados ao usuário do sistema, que constituem a razão final de seu uso, como editores de texto, navegadores Internet ou jogos. Entre os aplicativos e o *hardware* reside uma camada de *software* multifacetada e complexa, denominada genericamente de Sistema Operacional."

Para Tanenbaum (2010, p. 2), o sistema operacional realiza basicamente duas funções não relacionadas:

> [...] "fornecer aos programadores de aplicativos (e aos programas aplicativos naturalmente) um conjunto de recursos abstratos claros em vez de recursos confusos de *Hardware* e gerenciar esses recursos de *Hardware*."

Assim, podemos compreender que um sistema computacional deve permitir a solução de problemas e a execução adequada de programas dos usuários.

Avançando na definição apresentada por Maziero, podemos considerar que um sistema computacional é composto por três componentes básicos e fundamentais: as aplicações, os usuários e o hardware.

O primeiro componente, as aplicações, definem como os recursos do sistema serão utilizados para resolver os problemas computacionais dos usuários. São exemplos de aplicações: compiladores, banco de dados, programas comerciais e jogos.

O segundo componente, os usuários, são os utilizadores do sistema computacional. São exemplos de usuários: as pessoas, as máquinas e outros computadores ou sistemas computacionais.

O terceiro componente, o hardware, são os recursos básicos de computação e se dividem em três subsistemas básicos: Unidade Central de Processamento, Memória Principal e Dispositivos de Entrada e Saída. A Figura 1 demonstra essa interação entre os usuários, aplicações e o hardware, sempre mediada pelo Sistema Operacional.

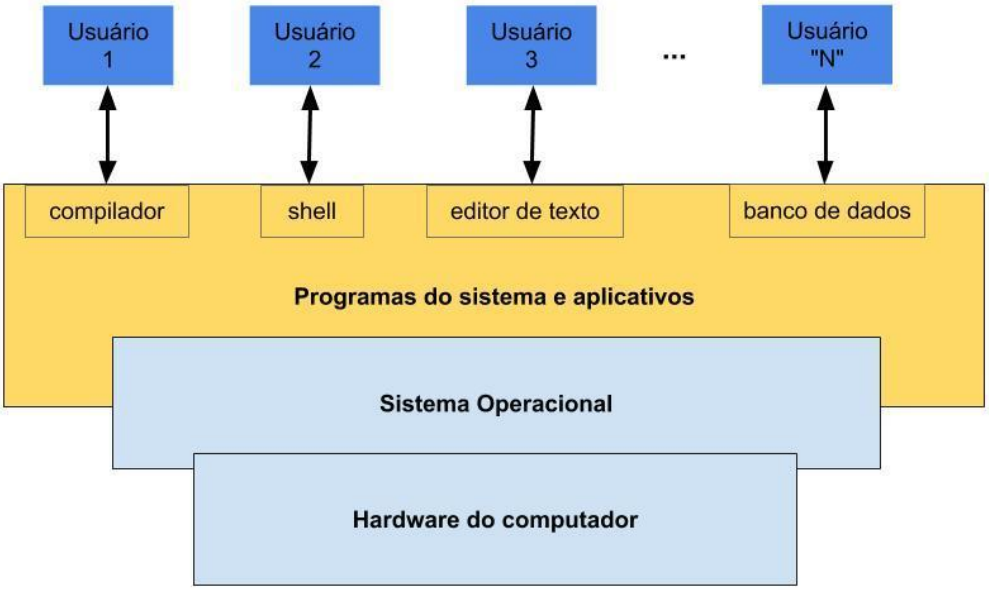


Figura 1 - Componentes básicos de um sistema computacional.
Fonte: traduzida e adaptada de http://www.komputersalatiga.com/2019/08/complete-computer-component-image-and-its-functions.html

O sistema operacional atua como uma camada de *software* entre os programas aplicativos dos usuários finais e o *hardware*. Ele é uma estrutura complexa de *software*, bastante ampla, que incorpora aspectos de alto nível, como a interface gráfica e os programas utilitários, e de baixo nível, como a gerência de memória e os *drivers* de dispositivos.

Um sistema operacional pode ser multitarefas e multiusuários, como demonstrado na Figura 1. Neste caso, ele deverá permitir que cada processo e usuário utilizem os recursos de *hardware* e *software* de forma independente, sem se preocupar com os outros *softwares* e usuários que estão utilizando o sistema ao mesmo tempo.

A função de um sistema operacional pode ser estabelecida por meio de duas abordagens: *top-down* (de cima para baixo) ou *bottom-up* (de baixo para cima). A abordagem *top-down* considera que a partir da máquina real (*hardware*) podemos criar uma máquina estendida ("abstrata" ou "virtual") onde o sistema operacional é uma extensão do *hardware* que implementa uma interface para as aplicações (programas). A visão *bottom-up* descreve que o sistema operacional é um controlador de recursos do sistema, ou seja, gerencia os recursos de *hardware* disponíveis para as aplicações.

A finalidade principal de um sistema operacional pode ser sintetizada em duas palavras-chave: “abstração” e “gerência”.

A **abstração de recursos** é necessária para facilitar o acesso aos recursos de *hardware* de um sistema de computação, já que esta pode ser uma tarefa complexa considerando as características específicas que cada dispositivo físico pode apresentar, além da complexidade de suas interfaces. Essa abstração pode tornar os aplicativos independentes do *hardware*.

A **gerência de recursos** é importante para definir a prioridade de acesso dos aplicativos sobre os recursos de *hardware*, quando dois aplicativos necessitam acessar o mesmo recurso, por exemplo. Assim é o Sistema Operacional que deve definir a prioridade de acesso (gerenciar políticas) resolvendo eventuais disputas e conflitos. Sobre esses aspectos de abstração e gerência de recursos do sistema operacional, Maziero (2017, p.4) reforça que:

> "Um sistema operacional visa abstrair o acesso e gerenciar os recursos de *hardware*, provendo aos aplicativos um ambiente de execução abstrato, no qual o acesso aos recursos se faz através de interfaces simples, independentes das características e detalhes de baixo nível, e no qual os conflitos no uso do *hardware* são minimizados."

Desta forma, podemos compreender o sistema operacional como uma camada transparente para o usuário permitindo que o computador seja utilizado de forma conveniente e eficiente ocultando a complexidade do *hardware*.

A Figura 2 demonstra melhor essa interação entre o usuário (e seus *softwares* e aplicações) e o *hardware* do computador (dispositivos físicos) mediado pelo sistema operacional.

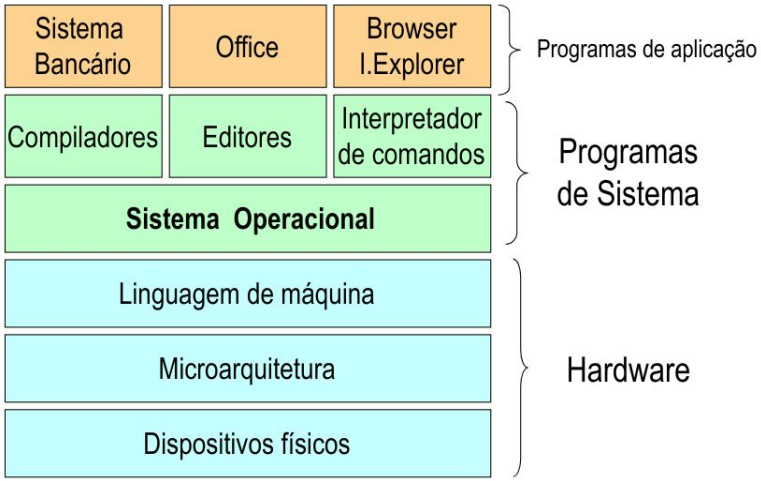


Figura 2 - Sistema operacional - Interface entre o usuário e o hardware.

Fonte: elaborado pelo autor.

A Figura 3 permite compreender que o sistema operacional também é responsável por oferecer interfaces padronizadas de acesso ao hardware e permitir uma visão homogênea dos dispositivos de hardware. O usuário clica em um arquivo, na sua aplicação, e o sistema operacional faz a interface de acesso ao hardware e aos dados no disco de forma transparente.

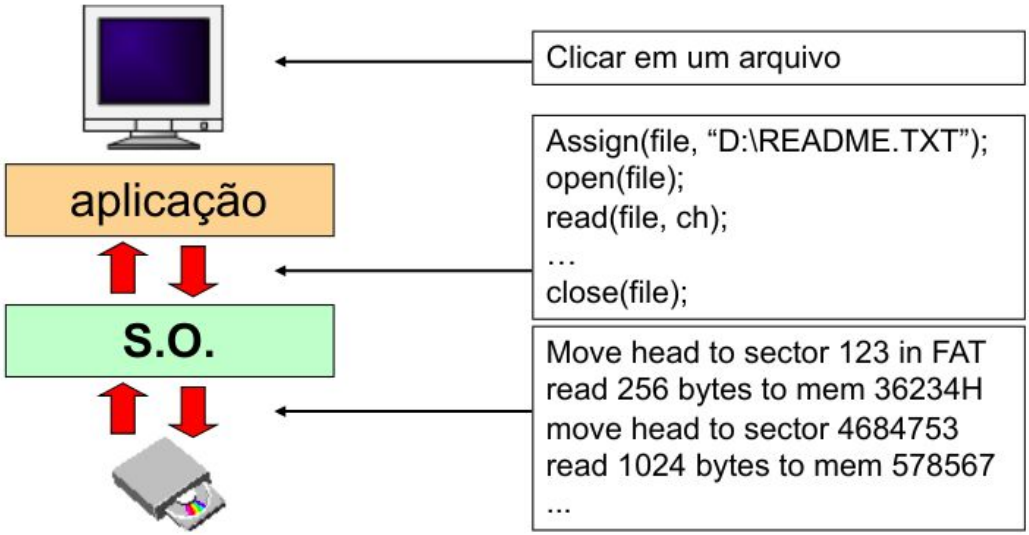


Figura 3 - Sistema operacional como máquina estendida (abstrata).
Fonte: elaborado pelo autor.

O sistema operacional como gerente de recursos faz a gestão do uso do processador, do espaço em memória, do acesso a arquivos, à conexões de rede e a dispositivos externos, sempre buscando evitar conflitos.

Cada sistema operacional oferece serviços e programas específicos, porém apresentam funções em comum. Vejamos algumas:

- **Execução de programas:** capacidade de carregar, executar e finalizar um programa;
- **Operações de I/O (entrada e saída):** deve fornecer meios para controlar arquivos ou dispositivos de I/O;
- **Manipulação do sistema de arquivos:** ler, gravar, criar e excluir arquivos;
- **Comunicação:** comunicação entre processos;
- **Detecção de erros:** indicar falhas de CPU, memória ou dispositivos de I/O e tomar medidas adequadas;
- **Alocação de recursos:** gerenciar recursos de memória, CPU ou dispositivos I/O;
- **Contabilização:** manter um registro dos usuários que utilizam os recursos do computador referente a quantidade e que tipo de recursos;
- **Proteção:** A proteção visa garantir que todo acesso aos recursos do sistema seja controlado, evitando conflitos e integridade dos dados.

Podemos desta forma, considerar que um sistema operacional deve possuir características desejáveis.

Vejamos a tabela a seguir para compreendermos melhor quais características são essas:

Característica	Definição	Exemplo
Concorrência	Existência de várias atividades ocorrendo paralelamente.	Execução simultânea de "*jobs*" (tarefas), E/S paralela ao processamento.
Compartilhamento	Uso coordenado e compartilhado de recursos de *Hardware* e *Software*.	Custo de equipamentos, reutilização de programas, redução de redundâncias, etc.
Armazenamento de dados	Capacidade de armazenamento a longo prazo.	Preservação das informações.
Não determinismo	Atendimento de eventos que podem ocorrer de forma imprevisível.	Atender a demandas inesperadas do *hardware*, *softwares* ou dos usuários.
Eficiência	Baixo tempo de resposta, pouca ociosidade da CPU e alta taxa de processamento.	Ampliar o desempenho do SO.
Confiabilidade	Pouca incidência de falhas e exatidão dos dados computados.	Ampliar o desempenho do SO.
Mantenabilidade	Facilidade de correção ou incorporação de novas características.	Facilidade de evolução.
Pequena dimensão	Simplicidade e baixa ocupação da memória.	Facilidade de uso.

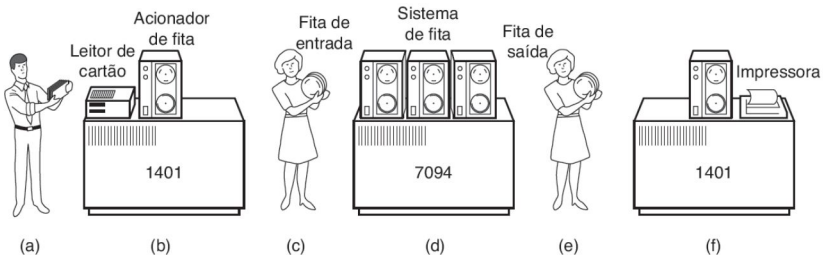
1.2 História dos Sistemas Operacionais

Não podemos aprofundar o tema de sistemas operacionais sem antes fazer uma rápida revisão sobre o seu desenvolvimento histórico. Esse processo permitirá entender como surgiram determinadas características e padrões de design que continuaram se desenvolvendo nas décadas seguintes. Conhecer os fatores que motivaram diferentes desenvolvimentos podem ajudar a prever e prevenir problemas.

Quando os Sistemas Operacionais surgiram, o usuário era o programador e o operador da máquina ao mesmo tempo. Existia muita interação humana no processamento das tarefas. Assim como nas arquiteturas de hardware, os Sistemas Operacionais também passaram por um processo evolutivo, classificado em gerações ou fases.

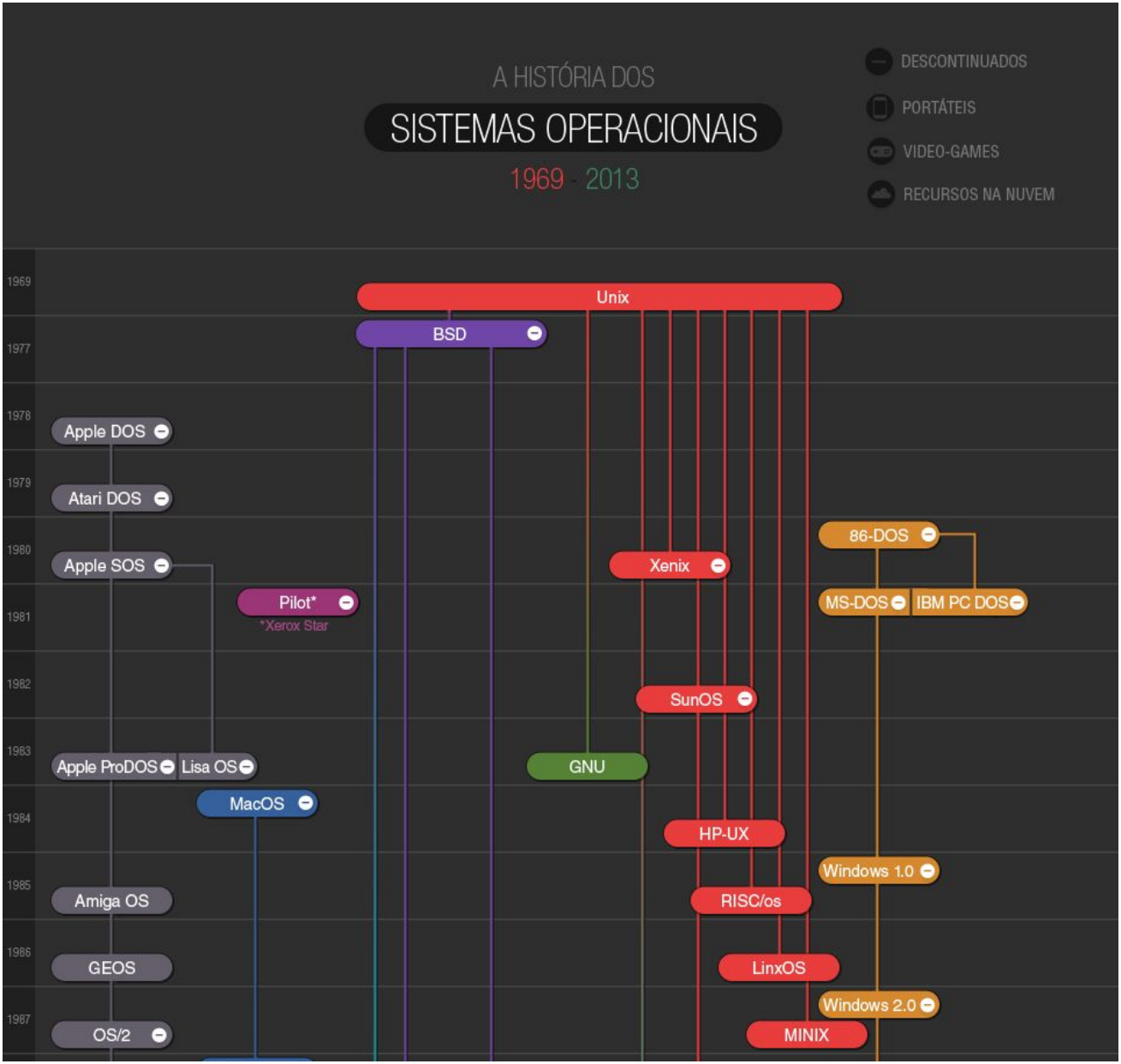
Vejamos a evolução histórica dos Sistemas Operacionais, agrupada em 5 fases:

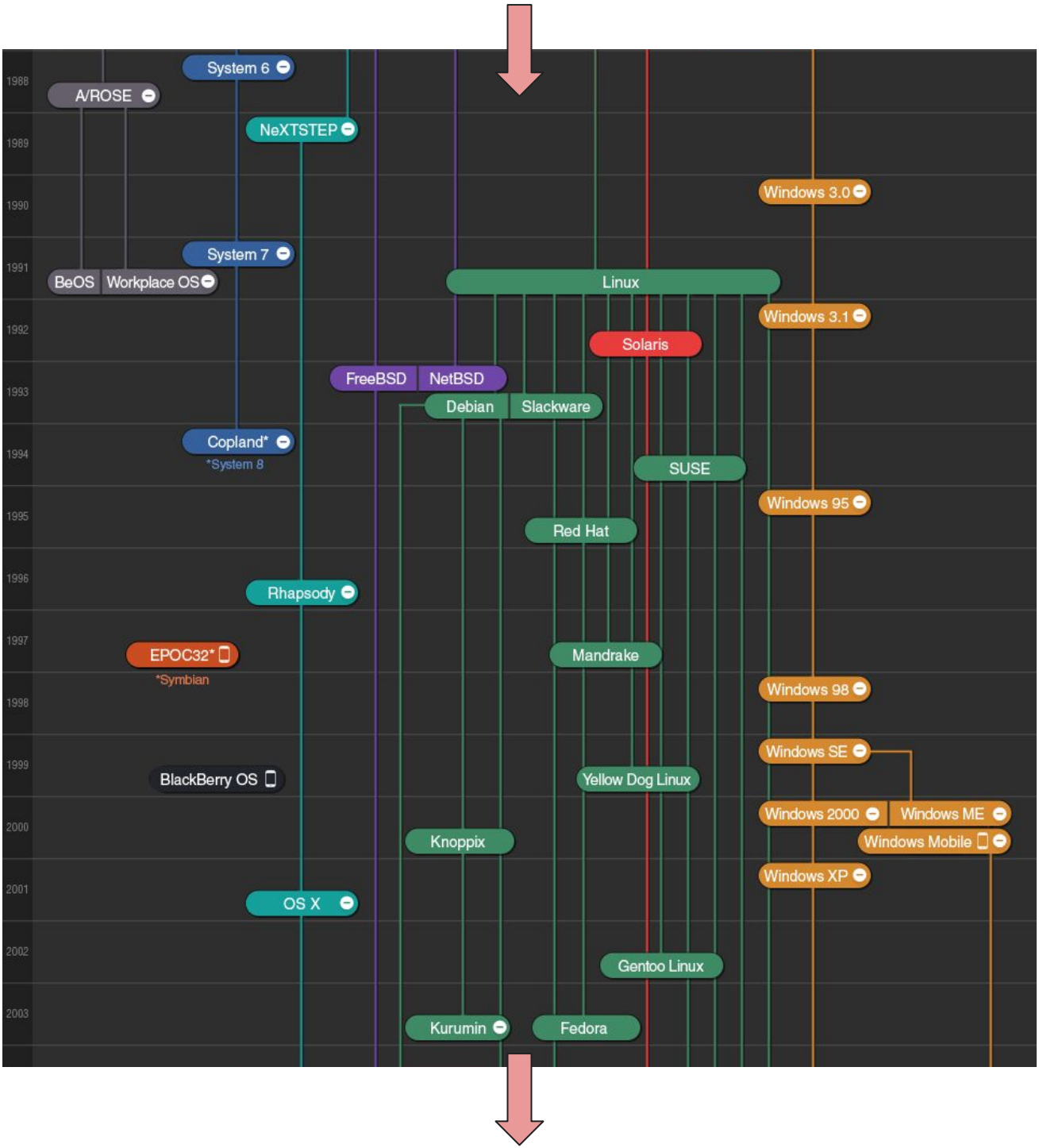
Fase Inicial (Fase 0) *Computadores são uma ciência experimental e exótica. Não precisa de sistema operacional*	• Programação através de "*plugs*". • Usuário presente todo o tempo e toda atividade é sequencial. • Conjuntos de cartões manualmente carregados para executar os programas. • Primeiras bibliotecas, utilizadas por todos. • O usuário é programador e operador da máquina ao mesmo tempo. **Problema: muita espera!** • Usuário tem que esperar pela máquina … • Máquina tem que esperar pelo usuário … • Todos têm que esperar pela leitora de cartões!

1ª Fase *Altos Preços. Computadores são caros. Pessoas são baratas.*	• SO surge com o objetivo básico de automatizar a preparação, a carga e a execução de programas. • SO torna a utilização do computador mais eficiente, desacoplando as atividades das pessoas das atividades do computador. • Mais tarde: otimização do uso dos recursos de *hardware* pelos programas. • SO funciona como um monitor *batch*, continuamente carregando um *job*, executando e continuando com o próximo *job*. Se o programa falhasse, o SO salvava uma cópia do conteúdo de memória para o programador depurar. **Exemplo de um sistema em lotes (*batch*) antigo:**  (a) Os programadores levam os cartões para o 1401. (b) O 1401 grava os lotes de tarefas nas fitas. (c) O operador leva a fita de entrada para o 7094. (d) 7094 executa o processamento. (e) O operador leva a fita de saída para o 1401. (f) 1401 imprime as saídas Figura 4 - Sistema em Lote Fonte: traduzido de: https://es.slideshare.net/jairakolatronic/evolucion-de-los-sistemas-operativos-32711361

2ª Fase *Produtividade - Custo/Benefício* *Computadores são rápidos; pessoas são lentas; ambos são caros.*	• "*Timesharing*" interativo: permitir que vários usuários utilizem a mesma máquina simultaneamente. • Um terminal para cada usuário. • Manter os dados "*on-line*": utilização de sistemas de arquivos estruturados. **Problema:** • Como prover tempo de resposta razoável?
3ª Fase *Produtividade - Custo/Benefício* *Computadores são baratos; pessoas são caras. Dar um computador para cada pessoa.*	• *Workstation* pessoal (SUN - Stanford University Network, Xerox Alto) • *Apple II* • *IBM PC* • *MacIntosh*
4ª Fase *Popularização Computadores Pessoais (PCs) em todo o planeta.*	• Redes possibilitam aparecimento de novas aplicações importantes. **Problemas:** • As pessoas ainda continuam esperando por computadores • Vírus, *worms*, *hackers*...

Os sistemas operacionais estão presentes nos mais diversos dispositivos, desde computadores pessoais até dispositivos móveis. A história do desenvolvimento dos sistemas operacionais pode ser estudada por meio de uma linha do tempo.


A HISTÓRIA DOS
SISTEMAS OPERACIONAIS
1969 - 2013
DESCONTINUADOS
PORTÁTEIS
VIDEO-GAMES
RECURSOS NA NUVEM
1969
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
Unix
BSD
Apple DOS
Atari DOS
Apple SOS
Pilot*
*Xerox Star
Xenix
86-DOS
MS-DOS
IBM PC DOS
SunOS
Apple ProDOS
Lisa OS
GNU
MacOS
HP-UX
Windows 1.0
Amiga OS
RISC/os
GEOS
LinxOS
Windows 2.0
OS/2
MINIX


1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
System 6
A/ROSE
NeXTSTEP
Windows 3.0
System 7
BeOS
Workplace OS
Linux
Windows 3.1
Solaris
FreeBSD
NetBSD
Debian
Slackware
Copland*
*System 8
SUSE
Windows 95
Red Hat
Rhapsody
EPOC32*
*Symbian
Mandrake
Windows 98
Windows SE
BlackBerry OS
Yellow Dog Linux
Windows 2000
Windows ME
Windows Mobile
Knoppix
Windows XP
OS X
Gentoo Linux
Kurumin
Fedora

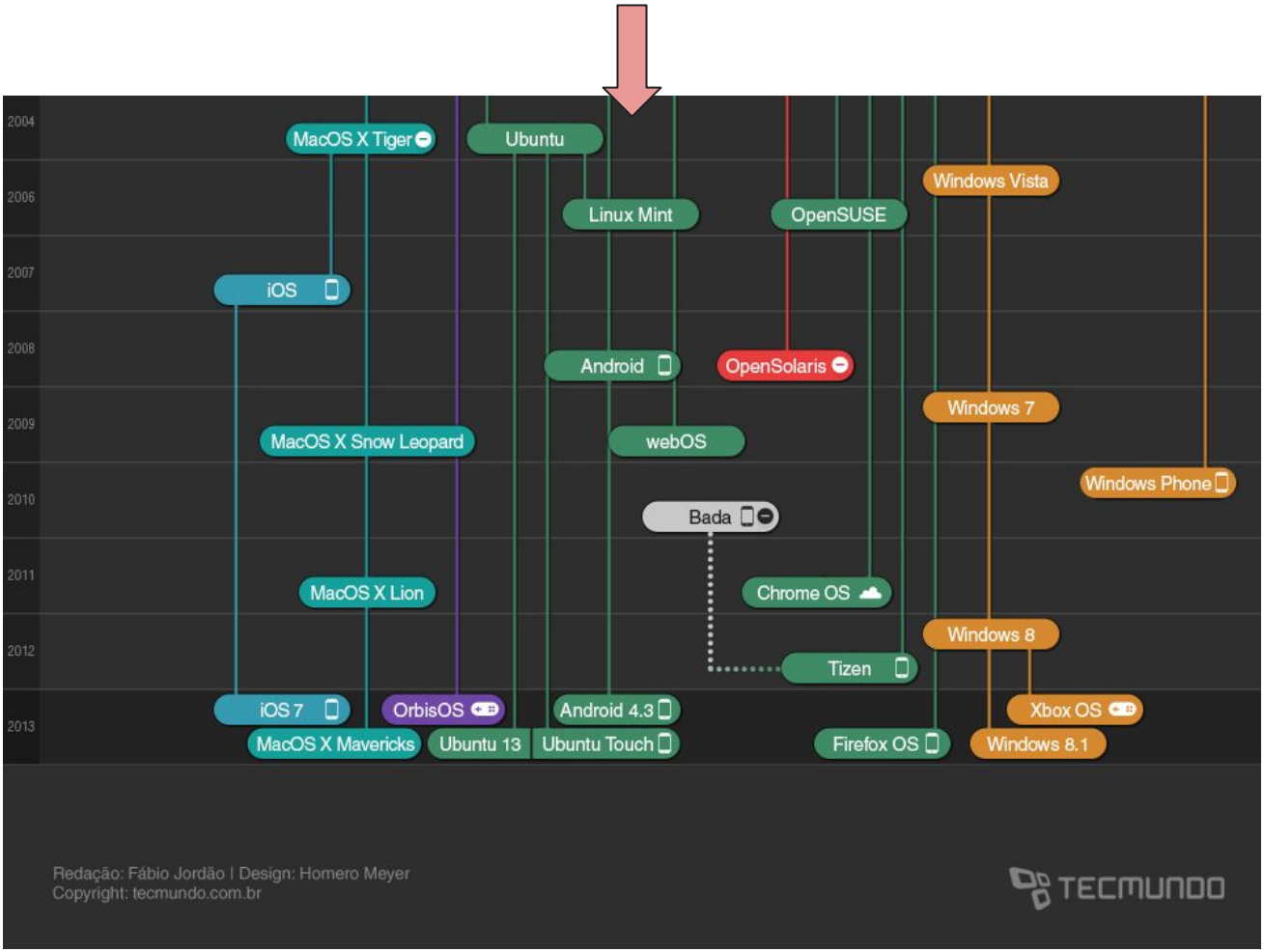


Figura 5 - Linha do tempo dos Sistemas operacionais.
Fonte: Site Techmundo - Disponível em:
https://www.tecmundo.com.br/sistema-operacional/2031-a-historia-dos-sistemas-operacionais-ilustracao-.htm



Como surgiram os sistemas operacionais?

Porque eles são tão importantes?

Para entender um pouco mais sobre sistemas operacionais e para responder às questões acima, assista ao vídeo que selecionamos. Ele trata da história dos sistemas operacionais.

Acesse:
https://www.youtube.com/watch?v=9rC9GilX1Io

1.3 História da Mobilidade

Nos últimos anos, grande parte do desenvolvimento no mundo da computação voltou-se para o modelo de computador representado, genericamente, por dispositivos móveis. O interesse por essas plataformas cresceu tanto que é necessário abordar a questão buscando sempre perceber as semelhanças com o restante dos equipamentos de informática.

Para fazer isso, no entanto, primeiro é necessário abordar a definição: o que é um dispositivo móvel ? Quais são os limites da sua definição?

Que fronteiras podem ser definidas? É difícil encontrar limites claros e rígidos para o que este conceito engloba.

Precisamos considerar que um dispositivo móvel é um computador concebido não apenas em nível de *hardware*, mas também, com design e interface de usuário, projetado para atender e organizar as demandas do usuário em suas tarefas diárias e cotidianas. Certamente, essa definição deixará alguma área cinzenta, pois está longe de ser perfeita e permitirá alguma ambiguidade. Um exemplo desse processo está nas versões mais recentes de alguns ambientes para dispositivos móveis (windows 8, gnome e *unity* do GNU/Linux) que buscam unificar a experiência do usuário incorporando conceitos dos *Desktop*.

Para começar nossa caminhada pela história da mobilidade e entender o surgimento dos sistemas operacionais desses equipamentos móveis, vamos refletir e tentar responder a essa pergunta inicial: você imagina quantos celulares (dispositivos móveis) há no Brasil?

De acordo com dados da Anatel, o Brasil terminou outubro de 2018 com 233,3 milhões de celulares e a densidade de 111,34 celulares para cada 100 habitantes. Incrível, não?

Esse resultado é fácil de compreender considerando que a grande maioria desses aparelhos faz muito mais do que chamadas telefônica. Foi-se o tempo em que um celular servia apenas para substituir os telefones, com a facilidade de podermos carregá-los conosco. Os telefones mais inteligentes, ou smartphones, são verdadeiros computadores de bolso, com capacidade de processamento superior a muito dos grandes computadores da década de 1980. Uma pesquisa da Fundação Getúlio Vargas (FGV), publicada em abril de 2016, apontou que existem 168 milhões de *smartphones* no Brasil. Isso quer dizer que, daqueles mais de 250 milhões de celulares que existem no País, 66% são máquinas inteligentes.

Certamente, essa proporção irá aumentar, devido à natural e gradual queda de preços dos equipamentos mais sofisticados.

a) Uso de Celulares e Smartphones no Brasil

Hoje, graças aos avançados dos sistemas que fazem esses aparelhos funcionarem, os chamados sistemas operacionais, dispomos de uma enormidade de aplicativos que nos permitem realizar diversas ações. Atualmente, podemos utilizar esses equipamentos para pagar contas, pedir comida, fazer compras, jogar, acessar redes sociais e trocar mensagens, assistir TV, filmes e ouvir músicas. Com eles podemos ainda ler livros e revistas, tirar e enviar fotos e filmes, e até mesmo chamar um táxi. Mas as facilidades não terminam aqui.

Além dessas funcionalidades descritas, podemos ainda controlar outros equipamentos, consultar a previsão do tempo, checar um trajeto de deslocamento e receber informações sobre o trânsito. Tudo está se integrando em um único equipamento. Esse processo é chamado de convergência digital.

Na verdade, essa lista cresce diariamente, pois sempre há alguém pensando em uma nova utilidade para esses aparelhos. Passamos cada vez mais a integrar a tecnologia ao nosso cotidiano e a depender de equipamentos móveis para muitas de nossas atividades diárias: lembrar o que temos de fazer, avisar com quem temos de falar e escolher o melhor caminho a seguir.

Não podemos esquecer que cada equipamento desses possui um hardware com inúmeras funcionalidades e necessita de um sistema operacional para coordenar e controlar os aplicativos e *softwares*, servindo de interface entre os usuários e os dispositivos disponíveis.

O acesso fácil e contínuo a redes sociais e aplicativos de troca de mensagens mudou os hábitos de milhões de pessoas, que passaram a ficar conectadas com amigos – da vida real ou virtuais – mesmo estando a muitos quilômetros de distância.

A dependência é tão grande que muita gente diz se sentir “sem roupas” quando esquece o celular em casa, e isso não deixa de ser verdade!



Vício Digital

Quando a dependência dos dispositivos móveis e a necessidade de estar conectado 24 horas por dia ultrapassa os limites do razoável, trata-se de vício digital. Mas o que seria razoável? Difícil de dizer. O fato é que, se a utilização dos dispositivos móveis, ao invés de ajudar, começar a atrapalhar a vida e o trabalho de uma pessoa, esse deve ser um sinal de alerta para que seus hábitos sejam revistos.

Vamos ler um pouco mais sobre o tema?

Visite:
https://repositorio.ufpb.br/jspui/handle/123456789/1867

b) Outros Equipamentos Móveis

Falamos muito dos celulares e *smartphones*, mas há uma série de outros equipamentos móveis. Observe a tabela a seguir com um pequeno exemplo descrevendo os equipamentos móveis mais comuns disponíveis atualmente.

Tablets	**Terminais de pagamento**	***Smartwatches***
		
Imagem disponível em: http://shopfacil.vteximg.com.br/arquivos/ids/11848376/image-d86ad6240abb4a239f7ccc68db0250dd.jpg?v=636595737675500000	Imagem disponível em: http://melhormaquinadecartao.com/wp-content/uploads/10-Melhores-Ma%CC%81quinas-de-Carto%CC%83es-Pelo-Celular-Aplicativo-Android-e-iOS-.png	Imagem disponível em: https://www.altonivel.com.mx/assets/images/Estructura_2016/Tecnologia/smart-watch.jpg
Maiores e mais confortáveis para se utilizar com aplicações que exigem muita digitação ou com as que têm telas mais complexas, os *tablets* utilizam, basicamente, os mesmos aplicativos que os *smartphones*, pois são baseados nos mesmos sistemas operacionais.	Máquinas móveis conectadas em rede, os terminais de pagamento de lojas, restaurantes e postos de combustível têm sistemas internos, programas e aplicativos que os fazem executar funções específicas, por isso, também podem ser considerados dispositivos móveis, embora sejam de utilização mais restrita.	Evoluindo em direção a miniaturização e integração dos equipamentos às pessoas – dispositivos para "vestir" –, os *smartwatches* (relógios inteligentes) já estão disponíveis no mercado. Embora ainda limitados a servir de conexão aos *smartphones*, esses relógios deverão tornar-se equipamentos independentes em breve, assumindo funções que hoje são realizadas pelos *smartphones* e *tablets*.

O crescimento no número de dispositivos que empregam sistemas operacionais não para. E o futuro promete ser ainda mais cheio de dispositivos conectados! Vamos entender o por quê?

c) Internet das Coisas

Criado em 1999, pelo *Massachusetts Institute of Technology* (MIT), o conceito da "Internet das Coisas" está cada vez mais se desenvolvendo. O MIT prevê que todos (ou quase todos) os objetos existentes, como carros, lâmpadas e canetas estarão conectados na próxima década. Estudos realizados pelo instituto, dão conta que cada um de nós está em contato com cerca de 1.000 a 5.000 objetos nesse atribulado dia a dia moderno.

Pense em quanta informação pode ser gerada se soubermos em tempo real (conectados) quais são e onde estão todos esses objetos e equipamentos?	 Imagem disponível em: http://guiadecompras.casasbahia.com.br/imagens/2017/04/internet-coisas-dia-a-dia.jpg
Imagine acessar esses dispositivos em tempo real. Verificar se é hora de manutenção, ou controlá-los para coletar dados e executar determinadas tarefas.	
Ou saber o que está ocorrendo em torno deles em tempo real, ou até mesmo se um produto dentro da geladeira está vencendo ou com validade expirada. Incrível, não? Isso tudo já é possível!	

Cada vez mais baratos e populares, a conectividade sem fio com esses dispositivos já é uma realidade. Seus pequenos *chips* e processadores, integrados a essa conectividade real, ampliam as possibilidades de uso e emprego. O limite está no *hardware*, no sistema operacional empregado, nos *softwares* e aplicativos, e é claro na nossa imaginação.



De onde vieram os smartphones tão populares e indispensáveis para nós?

Vamos assistir a dois vídeos que irão nos ajudar a compreender o surgimento dos dispositivos móveis. Como surgiu o celular? Como evoluiu a cultura da mobilidade e da conectividade em dispositivos móveis ?

Para saber um pouco mais e para entender como essa evolução aconteceu, separamos dois vídeos interessantes.

O primeiro é do CANAL NOSTALGIA no YOUTUBE narrando à evolução do celular. Acesse: https://www.youtube.com/watch?v=lNbjAiEUQQU

O segundo é do Discovery Ciência e conta a história do celular. Acesse: https://www.youtube.com/watch?v=yjiB_Yw05RE

Agora que já conhecemos a história dos sistemas operacionais e os principais trechos da história da mobilidade, vamos observar, de forma mais detalhada, a evolução tecnológica das telecomunicações que nos trouxe até o estágio atual. Sim, entender como a conectividade e a possibilidade de acesso à Internet nos dispositivos móveis são importantes para compreender como ocorreu a evolução dos dispositivos móveis e seu sistema operacional. Para isso, vamos seguir os caminhos das duas tecnologias que deram origem aos dispositivos móveis modernos: a telefonia celular e a computação móvel.

d) Telefonia Celular

A ideia de ter um telefone móvel nasceu em 1947. No entanto, à época, os recursos tecnológicos disponíveis não permitiram criar nada além de telefones que funcionavam como radiocomunicadores, e só eram viáveis em veículos automotores e embarcações, devido, principalmente, ao volume e ao peso de seus transmissores e baterias.

Durante a Segunda Guerra Mundial, versões rudimentares desses aparelhos portáteis de radiocomunicação foram carregadas pelas tropas americanas. Em cada equipe de campo havia um soldado responsável pelas comunicações que levava o equipamento em uma grande e pesada mochila. O alcance e a duração das baterias eram limitados e não permitiam um funcionamento regular por muito tempo. Mesmo assim, essa “mobilidade” inspirou os pesquisadores a pensarem em equipamentos móveis para uso no dia a dia.

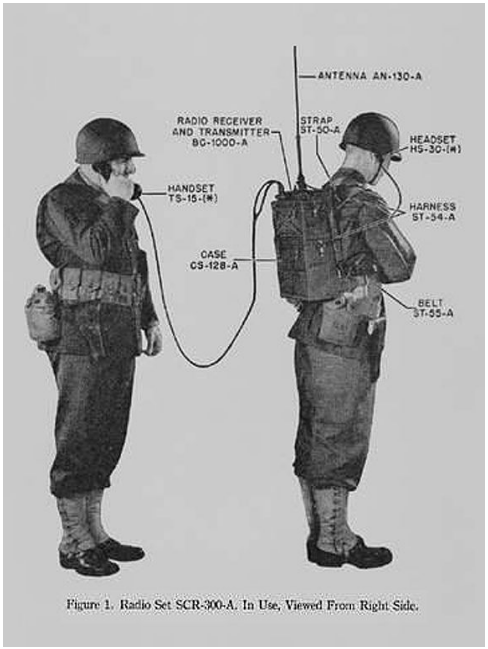


Figura 6 - Rádio comunicador portátil - SCR-300 - Inspiração para o celular.
Fonte: Wikipedia. Disponível em
https://es.wikipedia.org/wiki/SCR-300#/media/File:Scr300.png

Em 1973, os primeiros testes de uma rede de telefonia celular deram certo. Nesse momento, foi definido o modelo de funcionamento que é utilizado até hoje, com uma rede de estações – chamadas rádio base – que fornece uma cobertura de sinal para os aparelhos. Dessa forma, um aparelho celular deve estar ao alcance de uma dessas estações para que funcione. Somente dez anos após os testes iniciais, o primeiro aparelho celular foi comercializado. Naquela época, ele pesava cerca de 1 kg e tinha 30 cm de altura. O equipamento se chamava DynaTAC e era produzido pela Motorola.



Fonte: Techmundo. Disponível em <https://www.dinheirovivo.pt/wp-content/uploads/2017/04/dyna.jpg >



Fonte: Dinheirovivo. Disponível em <https://s.glbimg.com/po/tt/f/original/2011/07/08/2007computex_e21forum-martincooper.jpg >

Figura 7 - Motorola Dynatac 8000X, um verdadeiro celular portátil.

Era inevitável que a tecnologia evoluísse, tornando os aparelhos menores e mais fáceis de utilizar. No início da década de 1990, foram lançados novos aparelhos que já podiam ser carregados por uma pessoa, sem grande esforço, em uma bolsa ou presos a um cinto. Essa revolução tecnológica popularizou a telefonia celular e provocou uma verdadeira explosão na quantidade de consumidores no mundo. No Brasil, essa tecnologia começou a chegar em 1990, mas ainda restrita e muito cara.

A ampliação da área de cobertura das operadoras e a redução dos valores dos serviços para o consumidor final, no Brasil, ocorreram quase dez anos depois. Foi a partir de 1998 que as redes brasileiras de telefonia celular sofreram uma grande expansão. Além da evolução dos aparelhos, era necessário que houvesse uma ampliação na oferta de serviços, por meio de novas funcionalidades.

A primeira dessas funcionalidades foi o serviço de mensagens curtas, conhecidas como SMS (*short message service*). A inovação das mensagens curtas foi criada pela Nokia, na Finlândia, também no início dos anos 1990, e foi logo adotada por outros fabricantes, o que contribuiu para a evolução dos aparelhos celulares, que passaram também a trocar mensagens de texto.

Com a modernização das redes e o aumento da velocidade de transmissão disponível, os fabricantes enxergaram um novo mundo de possibilidades por meio da transmissão de dados. Inicialmente, os próprios fabricantes e as operadoras colocavam certos conteúdos disponíveis para *download*, ainda em redes fechadas, mas não demorou muito para que o acesso à internet fosse disponibilizado para celulares.

No início da década de 2000, a internet passou por uma verdadeira explosão de crescimento de oferta e utilização em todo o mundo. A conexão dos celulares a essa rede proporciona a disponibilização de uma gama nunca imaginada de novos serviços.

Surgiu então, uma verdadeira corrida pelo fornecimento de serviços móveis. Vendia-se a ideia de que era possível trabalhar de qualquer lugar e de que os computadores pessoais estavam com os dias contatos. Na verdade, estávamos ainda muito longe disso, mas a velocidade com que a tecnologia evoluiu foi impressionante.

Para entender a evolução dos sistemas operacionais e dispositivos é preciso entender como a conectividade e a oferta de novos serviços surgiram. Com o passar do tempo, a tecnologia empregada na transmissão de sinais dos celulares evoluiu rapidamente. Com isso, iniciou-se uma verdadeira guerra entre os fabricantes de equipamentos para se estabelecer um novo padrão tecnológico.

A maioria dos celulares comercializados no início da década de 2000 utilizava duas tecnologias amplamente difundidas na América do Norte: a TDMA (*Time Division Multiple Access*) e a CDMA (*Code Division Multiple Access*). Como a CDMA era mais evoluída, foi a mais utilizada no início da implantação das redes de telefonia celular no nosso país.

Em paralelo, empresas de telefonia europeias desenvolveram um sistema chamado GSM (*Global System for Mobile Communications*), que, além de suportar padrões mais elevados de velocidade e qualidade na transmissão, trazia uma novidade chamada módulo SIM (*Subscriber Identity Module*), o conhecido chip, ou cartão, que identifica o assinante ou “dono" da linha telefônica.

Não demorou para o GSM se tornar o novo padrão mundial, principalmente porque os fabricantes viram uma grande oportunidade: com a tecnologia GSM, o aparelho não ficava mais “preso” à linha telefônica. Dessa forma, era possível que o usuário demorasse menos tempo para trocar de equipamento e até tivesse mais de um aparelho, bastando colocar seu cartão SIM no aparelho que desejasse utilizar. A tecnologia GSM pode ser considerada um marco na história da telefonia e foi responsável por tornar os aparelhos celulares objetos mais ligados à moda, ou seja, facilmente substituíveis pelo usuário. Hoje essa é a tecnologia mais utilizada no mundo.

As novas funcionalidades dos aparelhos e a crescente demanda por transmissão de voz e dados moveram o desenvolvimento de redes de transmissão de segunda geração, as chamadas 2G, mais adequadas para suportar o novo padrão de utilização.

A transformação da telefonia celular de, apenas, mais um canal de voz para um sistema mais completo de comunicação foi considerada o ponto de partida para a consolidação da computação móvel e o surgimento dos sistemas operacionais para dispositivos móveis.

A tecnologia dos aparelhos e as novas redes 3G e 4G tornaram-se ferramentas de uso diário de milhares de pessoas ao redor do mundo. Vimos até aqui que a evolução da telefonia celular proporcionou o acesso a voz e dados em um aparelho móvel. É nesse ponto de nossa História que as tecnologias começam a se confundir com a evolução da computação móvel. Vamos ver?

e) Computação Móvel

A computação móvel nasceu em meados de 1992, com o lançamento do handheld Newton pela Apple. Handheld era o nome dado aos aparelhos que podiam ser utilizados em uma mão, também chamados de PDA (*Personal Digital Assistant*) ou assistente pessoal digital.

O Newton tinha 1 MB de memória e tela sensível ao toque, uma inovação para a época. Apesar disso, não chegou a ser um sucesso. Embora muita gente falasse dele, suas vendas foram inferiores a mil unidades no primeiro ano de comercialização.

Além de ser considerado grande, pesado e caro, o aparelho não dispunha de aplicativos que justificassem sua compra pela maioria das pessoas. Este é o grande segredo dos dispositivos móveis: para ter sucesso, eles têm de disponibilizar aplicativos úteis para as pessoas. Aplicativos? Pense e responda: para que a maioria das pessoas utiliza seus telefones celulares hoje? Com certeza, não é para fazer ligações.

O mercado dos dispositivos móveis começou a mudar em 1996, com o lançamento do Palm Pilot, um PDA produzido pela empresa norte-americana U.S. Robotics, conhecida fabricante de componentes para computadores que, anos depois, criou outra empresa chamada Palm, dado o sucesso do produto.

O Palm Pilot teve várias versões e gerações, mas o que o tornou popular e responsável pela mudança nos hábitos dos consumidores foi a sua concepção: era um aparelho compacto, com uma inovadora interface por caneta, que permite fazer anotações à mão, de uma forma bastante eficiente. Devido ao êxito dessa nova plataforma, outros fabricantes lançaram dispositivos semelhantes, entre eles a HP, a Casio e a NEC. A Microsoft apostou na criação de um sistema operacional para ser o "padrão" dessas máquinas: o chamado Windows CE, que evoluiu para um sistema chamado Pocket PC no ano 2000.

Outros fabricantes, ainda no final do século XX, lançaram aparelhos que tinham pequenos teclados, o que limitava suas funcionalidades e determinava um tamanho mínimo. No entanto, a maquininha trazia agenda de contatos, agenda de compromissos, funcionalidades de alertas e lembretes, alguns jogos e uma interface com computadores para a sincronização dos dados, o que a tornou um sucesso! A vida estava bem melhor para quem tinha que trabalhar com mobilidade, pelo menos na questão de equipamentos de uso pessoal.

No início do ano 2000, os executivos mais modernos começavam a levar consigo dois aparelhos: um celular para fazer suas ligações e um handheld para manter sua agenda de compromissos e agenda de contatos, além de utilizar alguns aplicativos simples, como alarmes e lembretes.

Foi então que os fabricantes começaram a pensar em juntar as duas coisas. Será que os dois aparelhos não poderiam se tornar um só?

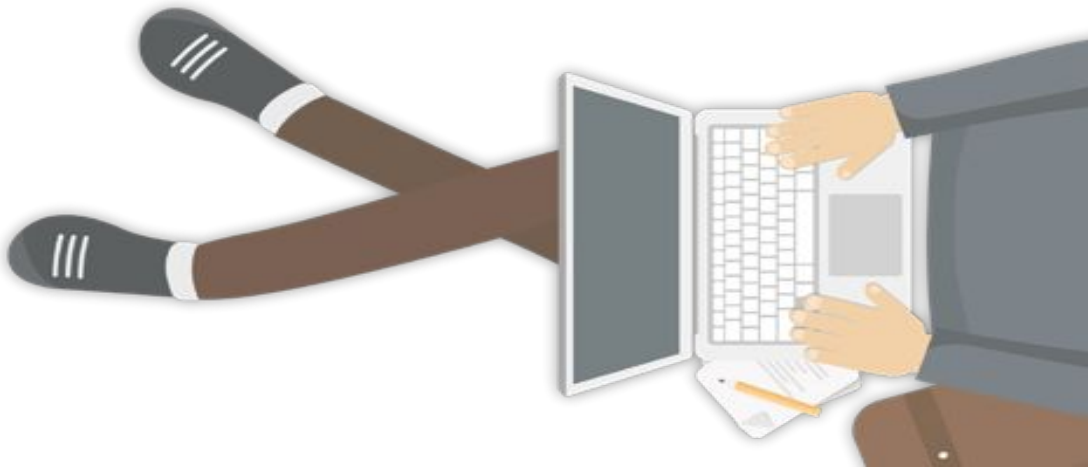
Uma das primeiras criações em direção à integração do celular com o handheld foi o Palm Pilot 7, que tinha todas as funcionalidades do tradicional PDA, mas com um telefone celular incorporado, com direito a anteninha e tudo mais.

No entanto, devido à mudança de tecnologia de transmissão de dados, que levou o GSM a ser o novo padrão mundial, esse aparelho teve vida curta e não chegou a ser comercializado no Brasil.

A partir dessa nova demanda para dispositivos móveis, os fabricantes de celulares do mundo todo começaram, então, a criar suas próprias máquinas inteligentes, com sistemas operacionais próprios.

Foi uma nova corrida, dessa vez para tentar definir um padrão para o smartphone. É importante que você conheça os sistemas que estão em uso, pois é para eles que se desenvolvem os aplicativos e são direcionadas as novidades. No decorrer desses anos, os dispositivos móveis evoluíram e os equipamentos mais antigos foram substituídos.

Em 2009, a pioneira Palm trocou seu Palm OS por um novo sistema operacional, chamado WebOS, cujas versões subsequentes foram utilizadas pela HP e LG. A Samsung criou, em 2010, o sistema operacional Bada, nome que significa "oceano" em coreano. O Bada foi incorporado à linha de smartphones Wave, com o objetivo de oferecer uma opção de baixo custo frente ao BlackBerry e ao iPhone. Nesse caminho, também desapareceram os sistemas Bada, Symbian e WebOS.





Estatísticas

Para saber mais sobre estatísticas de celulares no Brasil, acesse:
http://www.teleco.com.br/ncel.asp

1.4 Sistemas Operacionais Desktop

Um sistema operacional para *desktop* pode ser entendido como o sistema operacional desenvolvido para um computador de um usuário final (pode ser um *notebook* ou um computador pessoal). É claro que existem sistemas operacionais para servidores que normalmente irão prover algum tipo de serviço específico (como servidor *web*, serviço de rede ou outra funcionalidade). Porém, a evolução do hardware dos computadores levou a uma aproximação dos sistemas operacionais.

Vamos conhecer um pouco sobre os sistemas operacionais mais utilizados em computadores e dispositivos móveis.

1.4.1 GNU / Linux

O Sistema operacional GNU / Linux surgiu quando Linus Torvalds, um estudante Finlandês de ciência da computação, anunciou uma versão prévia de um kernel para substituir o Minix em um *newsgroup Usenet* (comp.os.minix). A parte mais importante de um sistema operacional é o kernel. No SO GNU/Linux, o componente do kernel é o Linux. O restante do sistema consiste em outros programas, muitos dos quais escritos por, ou, para o projeto GNU. Como apenas o kernel sozinho não forma um sistema operacional utilizável, o correto é utilizar o termo “GNU/Linux”.

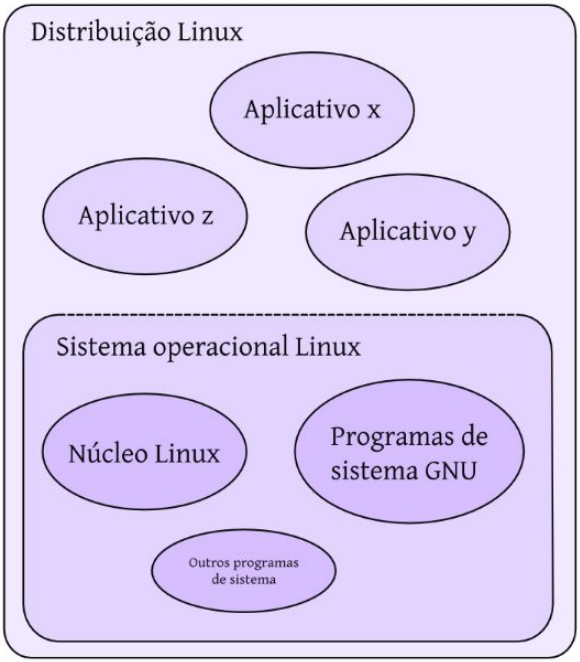


Figura 8 - Diagrama de uma distribuição GNU/Linux

Fonte: Wikipedia. Disponível em https://upload.wikimedia.org/wikipedia/commons/3/34/Diagrama_de_Venn_Linux_-_kernel%2C_so%2C_distribui%C3%A7%C3%A3o_%2B_explica%C3%A7%C3%B5es_%28pt-BR%29.png

O sistema operacional GNU/Linux é o resultado da contribuição de um grande número de empresas e pessoas, formando uma comunidade. Na verdade, o sistema GNU/Linux é um componente central, o qual é ramificado em vários produtos diferentes. Eles são chamados de distribuições.

As distribuições mudam completamente a aparência e o funcionamento SO de acordo com os objetivos da comunidade. As distribuições variam desde grandes e completos sistemas (mantidos por empresas) até pequenas distribuições que cabem em um cartão de memória USB ou rodam em computadores antigos (geralmente desenvolvidas por voluntários). As distribuições mais conhecidas são UBUNTU, DEBIAN, SUSE e REDHAT.

O GNU/Linux possui diversas interfaces gráficas e elas não podem ser confundidas com o Sistema Operacional, pois é apenas uma aplicação gráfica (interface) para tornar mais fácil o acesso ao sistema. As interfaces gráficas mais utilizadas são o KDE, GNOME e XFCE.



O que é Linux? - Conheça as principais distribuições!

Vamos conhecer um pouco mais sobre a história do GNU/LINUX e suas distribuições. O canal DIOLINUX apresenta esse sistema operacional, suas características e as principais distribuições.

Acesse:
https://www.youtube.com/watch?v=5nX4UFQt_JQ&t=110s

O site Distrowatch apresenta as principais distribuições GNU/Linux disponíveis.

Acesse:
https://distrowatch.com/

1.4.2 Windows

Os primeiros computadores não possuíam interface gráfica e o sistema era acessado apenas por comandos de texto. A usabilidade era bastante deficiente e complicada. A chegada da interface gráfica mudou essa perspectiva. A Microsoft iniciou com o sistema operacional MS-DOS (Microsoft Disk Operating System). As primeiras versões do Windows eram uma junção do MS-DOS com interface gráfica. A versão NT do Windows foi o primeiro sistema operacional da Microsoft a abandonar o MS-DOS. O Windows 95 (1995) apresentou o conceito plug and play (adição de periféricos), da barra ferramentas e do botão iniciar. O Windows 98 (1998), a partir do estouro da Internet, apresentou grandes novidades com o navegador IE4 (Internet Explorer) e o suporte ao padrão USB. O Windows 10 lançado em 2014 apresentou uma interface para dispositivos touchscreen e múltiplas áreas de trabalho.



Como surgiu o Windows?

É inegável a popularidade que o sistema operacional da Microsoft, o odiado e amado Windows, possui no segmento em que atua. O Windows não nasceu da forma como o conhecemos e nem com todos os recursos com os quais estamos familiarizados. Houve um processo gradual de evolução em que a Microsoft aprendeu quais eram as necessidades das pessoas e aperfeiçoou funcionalidades para equipar o seu programa.

Visite:
https://www.tecmundo.com.br/windows-10/64136-windows-1-windows-10-29-anos-evolucao-do-so-microsoft.htm

1.5 Sistemas Operacionais para Dispositivos Móveis e Mercado

Os dispositivos móveis (telefones celulares, smartphones, tablets e outros) ocupam cada vez mais tempo e espaço em nossas vidas. Com o advento da conectividade (internet), seus processadores estão mais velozes, há mais memória e um salto considerável no que diz respeito ao armazenamento foi dado.

Aquilo que era comum apenas aos computadores de mesa realizar, agora pode ser feito em qualquer lugar, a qualquer hora do dia, direto da palma da sua mão. Os dispositivos móveis, principalmente os smartphones, revolucionaram a nossa forma de acessar a internet e usar os aplicativos. Neste ambiente de revolução, estão também as empresas querendo "marcar sua posição" na guerra da mobilidade. Para os dispositivos móveis, seguindo a mesma linha do que ocorreu com os computadores pessoais, surgiram diversos sistemas operacionais. Vamos conhecer a seguir alguns deles.

1.5.1 Android

Projetado pela empresa Google, o Android baseia a maior parte de sua operação em software livre (um kernel Linux, uma máquina virtual Java e muitas das bibliotecas de sistema comuns ao GNU/Linux), adicionando uma camada de serviços proprietários. A estratégia do Google tem sido o oposto da Apple: em vez de fabricar seus próprios dispositivos, concede licenças para o uso do Android a praticamente todos os fabricantes de hardware, embarcando assim o seu sistema operacional na grande maioria dos modelos de smartphones e tablets.

A Google lançou, comercialmente, o sistema operacional *Android* em setembro de 2008, com o objetivo de oferecer ao mercado *smartphones* com grande variedade de recursos e com um custo mais baixo, já que a promessa era ter um sistema operacional sem custo, reduzindo o preço final dos aparelhos.

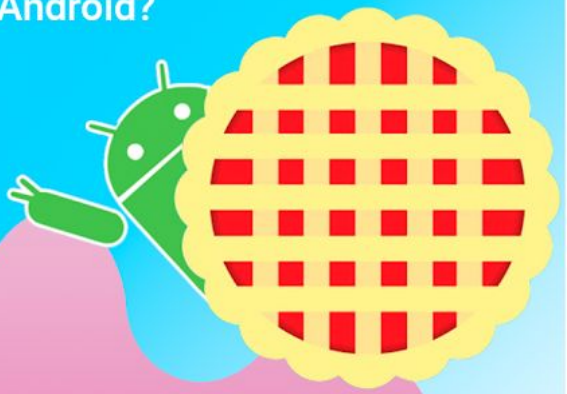
O *Android* foi logo adotado por grandes fabricantes, entre eles as coreanas Samsung e LG, a japonesa Sony e a americana Motorola, cuja divisão de equipamentos móveis pertenceu ao Google entre 2011 e 2014.

A utilização pelos grandes fabricantes, o custo baixo e a relativa facilidade de uso, colocaram, rapidamente, o *Android* à frente dos outros sistemas. Essas particularidades também incentivaram a realização de melhorias constantes, tanto na sofisticação dos dispositivos, quanto nas características e funcionalidades do próprio sistema operacional.

O sistema operacional *Android* passou por diversas versões desde o seu lançamento, todas acompanhadas de uma eficiente campanha de marketing, que associou os nomes das versões à doces.

Em 2008 e 2009, foram lançadas as primeiras duas versões, 1.0 e 2.0, foram chamadas de "A" e "B" respectivamente. Ainda em 2009, foram lançadas as versões *Cupcake* (bolo de caneca), *Donut* (rosquinha) e *Eclair* (bomba de chocolate), mantendo a ordem das letras, mas com a referência aos doces. Nos anos seguintes, tivemos a *Froyo* (sorvete de iogurte), *Gingerbread* (biscoito de gengibre), *Honeycomb* (favo de mel), *Ice Cream Sandwich* (um tipo de sorvete com biscoitos), *Jelly Bean* (bala de goma), *KitKat* (o famoso chocolate), *Lollipop* (pirulito), *Marshmallow*, e a *Nougat* (torrone), que é a versão 7.0 de agosto de 2016.

Você conhece as versões do sistema Android?



Este mês foi lançado ao público o **Android Pie**, a 9ª versão do sistema operacional. Aproveite a novidade para rever as diferenças e avanços de cada versão. Confere aí!

EVOLUÇÃO DO ANDROID



Android 1.0
Alpha

- Navegador Web simples;
- App Market para aplicativos;
- Suporte para câmera simples.



Android 1.6
Donut

- Interface para a programação de apps com reconhecimento de gestos;
- API de programação para uso de *text-to-speech*.



Android 2.1
Eclair

- Suporte para conexão Bluetooth 2.1;
- Adição e sincronização de várias contas;
- Flash, zoom digital e efeitos de cor nas fotos.



Android 2.3
Gingerbread

- Videochamadas;
- Suporte a NFC;
- Suporte a sensores de movimentos;
- Suporte a aparelhos com câmeras frontais.



Android 3.0
Honeycomb

Versão do Android projetada especialmente para tablets.



Android 4.0
Ice Cream Sandwich

- Controle de limite de dados utilizados;
- Android Beam – compartilhamento de dados por NFC;
- Tela inicial personalizada com organização de widgets;
- Unificou o SO de tablets e smartphones.



Figura 9 - Versões do SO *Android*
Fonte: Ilhasoft. Disponível em http://www.ilhasoft.com.br/wp-content/uploads/2018/08/Infografico-android.jpg

A versão mais atual do *Android* é a 11(*). A versão *Oreo* (8.0) apresentou funcionalidades integradas ao sistema como otimizações de performance (*runtime* e bateria), além do *Google Play Protect* e *Play Console*.

(*) versão mais atual em fevereiro de 2021.



A história do Android...

O canal TecMundo apresenta a série de história da tecnologia com a trajetória do Android, o sistema operacional móvel da Google. O vídeo conta a origem e a evolução das versões até chegar na Orion.

Assista o vídeo:
https://www.youtube.com/watch?v=5K4pEk19nhs

1.5.2 Apple iOS

O sistema operacional da Apple é projetado exclusivamente para o hardware produzido por ela. O iOS foi o primeiro a implementar a interface de usuário multitoque e, em grande medida, pode ser visto como responsável pela explosão e universalização no uso de dispositivos móveis. Assim como o sistema operacional para seus computadores de mesa, o MacOS X, o iOS é baseado no *kernel Darwin*, derivado do *FreeBSD*, um sistema livre, similar ao Unix.

Em 2007, a Apple lançou o iPhone, que era considerado por muitos apenas uma versão conectada do iPod. Ainda restrito a algumas operadoras americanas, como a AT&T, não se podia imaginar, na época de seu lançamento, o sucesso que esse dispositivo faria.

O sistema operacional iPhone OS, que, mais tarde, passou a se chamar iOS, aliado a um *hardware* compacto e de alta qualidade, elevou o iPhone a símbolo de status e tornou-o objeto de desejo, fator muito sabiamente trabalhado pelo marketing da Apple.

Não demorou muito para que o iPhone começasse a “roubar” adeptos do *BlackBerry* dentro das corporações, tornando-se o novo “queridinho” dos executivos.

1.5.3 Windows Phone

A Microsoft oferece uma versão de seu sistema operacional, compatível com a API do *Windows*, mas compilada para o processador ARM. Este sistema operacional não conseguiu ganhar grande popularidade, em claro contraste com o seu domínio na computação tradicional de desktop. O principal fabricante que vende equipamentos com o Windows Phone é a Nokia (que, após ser a empresa líder em telefonia, foi adquirida pela própria Microsoft).

A *Microsoft* havia desenvolvido o *Windows Mobile* a partir do *Pocket PC*, que, como vimos, foi uma evolução do Windows CE. A primeira versão do *Windows Mobile*, chamada Ozone, foi lançada em junho de 2003 e, a partir dessa versão, diversas outras versões já foram lançadas até hoje.

Por conta da competição de mercado, a *Microsoft* buscou parceiros, como a *Nokia*, para a fabricação de equipamentos que utilizassem seu sistema *Windows Phone*. No entanto, mesmo com a aquisição da *Nokia* para produção de equipamentos do Windows *Mobile* (hoje *Windows Phone*), a *Microsoft* não obteve uma boa participação de mercado com seu sistema operacional móvel. Em meados de 2016, a *Microsoft* anunciou que o Windows 10 será a única nomenclatura de sistemas operacionais para os PCs, consoles de jogos e dispositivos móveis, incluindo o *Surface*, misto de tablet e PC.

1.5.4 Symbian

Embora este sistema operacional já tenha sido declarado oficialmente morto, seu efeito no desenvolvimento inicial do segmento foi fundamental e não pode ser ignorado. O Symbian foi a principal plataforma da Nokia em seus dias de glória, assim como para muitos outros fabricantes.Quase todas as empresas que anteriormente vendiam equipamentos com o Symbian mudaram sua oferta para os sistemas Android.

A Ericsson, a Motorola e a Nokia uniram-se para criar o Symbian, sistema operacional adaptado de tecnologias para PDAs que chegou ao mercado em 2001, incorporado a alguns telefones dessas empresas.
O Symbian cumpriu seu papel à época, mas ficou obsoleto em menos de uma década, frente aos novos conceitos de sistemas operacionais.

1.5.5 BlackBerryOS

Em 2002, uma empresa canadense de nome RIM (Research in Motion Limited), lançou seu primeiro smartphone, o BlackBerry. Apesar de, em sua primeira versão, o BlackBerry ser um equipamento grandalhão e pouco charmoso, reconhecido de longe por sua cor azul, ele se tornou um padrão global para uso executivo rapidamente.

Com funcionalidades que agradavam em cheio as organizações e conferiam mobilidade aos executivos, o Blackberry dominou o mercado por anos, até o dia em que concorrentes à altura apareceram no horizonte. Assim como a RIM, que criou seus equipamentos movidos por um sistema operacional próprio, o BlackBerryOS, outros fabricantes continuaram a lançar aparelhos e sistemas com a mesma intenção: aparecer frente aos consumidores, que, a essa altura, já tinham dificuldades em entender qual seria o melhor para eles.



Que tal conhecer uma timeline (linha do tempo) completa dos sistemas operacionais com narração?

Essa é a proposta do vídeo elaborado pelo Christian Bruno. descrevendo a linha do tempo dos sistemas operacionais para computadores e dispositivos móveis.

Acesse:
https://www.youtube.com/watch?v=h1CEtMk1CYo

Fica claro, a partir dos sistemas que acabamos de apresentar, bem como da grande maioria dos sistemas operacionais usados para dispositivos móveis, que a diferenciação entre o segmento móvel e a computação tradicional (desktop e servidor) não está no próprio sistema operacional, mas em camadas mais altas. No entanto, a diferença vai muito além de uma mudança na interface do usuário. As características desses dispositivos indubitavelmente determinam questões substantivas e de fundo.

Vejamos algumas das características mais notórias:

a) Armazenamento em estado sólido - A primeira característica notória ao se manusear um telefone ou tablet é que ele não é mais feito com a mesma noção de fragilidade que sempre acompanhou o computador. Os discos rígidos são dispositivos de alta precisão mecânica, e um pequeno acidente pode significar a sua perda absoluta e definitiva. Dispositivos móveis operam com armazenamento em estado sólido, ou seja, em componentes eletrônicos sem partes móveis;

b) Multitarefa, mas monocontexto - A popularização da computação móvel levou a uma forte redução nas expectativas de multitarefa, principalmente porque nos dispositivos móveis há uma carência de memória virtual, e, a memória disponível se torna uma mercadoria escassa. Logo, o sistema operacional é forçado a limitar o número de processos interativos em execução. As interfaces de usuário usadas pelos sistemas móveis abandonam a metáfora da área de trabalho, para voltar ao processo de vários aplicativos em execução, porém com um único programa visível em todos os momentos. Os usuários abrem vários aplicativos, mas normalmente não os finalizam. Se eles não ocuparem toda a memória ficarão abertos para facilitar um novo acesso futuro, de forma mais rápida. O próprio sistema operacional definirá regras de quando os aplicativos deverão ser finalizados e com qual prioridade a memória será liberada;

c) Consumo elétrico - A economia do consumo elétrico tem duas vertentes principais: por um lado, o desenvolvimento de *hardware* mais energeticamente eficiente, independentemente da maneira em que opera e, por outro, a criação de mecanismos por meio dos quais um equipamento de informática pode detectar mudanças no padrão de atividade (ou o operador pode indicar uma mudança na resposta esperada), e este último reage reduzindo sua demanda (que é tipicamente obtida pela redução da velocidade de certos componentes do equipamento). A economia de energia que esses padrões de uso proporcionam não se deve apenas ao *hardware* cada vez mais eficiente usado pelos dispositivos móveis, mas também à programação de aplicativos nos quais os desenvolvedores procuram explicitamente padrões eficientes, suspensão fácil, e minimização na necessidade de acordar o *hardware*;

d) Ambiente em mudança - Nos *desktops* e servidores as mudanças de ambiente não são tão comuns. Normalmente não há grandes mudanças de rede, configurações ou consumo de energia. Uma das mudanças mais complexas de se implementar em sistemas operacionais de dispositivos móveis foi a de fornecer a plasticidade necessária nestes diferentes aspectos: o dispositivo móvel deve ser mais enérgico (consumir mais) em suas mudanças no perfil de energia, respondendo a um ambiente em mudança. Você pode aumentar ou diminuir o brilho da tela, dependendo do brilho ao redor, ou desabilitar determinadas funcionalidades se ela já estiver em níveis críticos de carga. Com relação à rede, por exemplo, você deve ser capaz de aproveitar as conexões fugazes enquanto o usuário está em movimento, iniciando eventos como a sincronização. Finalmente, é claro, a interface do usuário: os dispositivos móveis não têm uma orientação natural # exclusiva, como fazem os computadores. As interfaces do usuário devem ser projetadas para que possam ser reconfiguradas agilmente antes da rotação da tela;

e) O jardim murado - Uma consequência indireta (e não técnica) do nascimento de plataformas móveis é a popularização de um modelo de distribuição de *software* conhecido como jardim murado (plataforma fechada). A Apple, a Microsoft e a Google adotaram esse modelo de distribuição de aplicativos. A Apple com a Apple Store, a Microsoft com o Windows Phone Store e a Google com o *Google Play*. A peculiaridade deste modelo é que qualquer desenvolvedor pode criar um aplicativo e enviá-lo, porém as fabricantes se reservam no direito de aprová-lo, ou excluí-lo a qualquer momento. Ou seja, esse modelo permite que elas se tornem uma espécie de juiz, que pode determinar o que um usuário pode ou não instalar e executar.





Bora rever!

Essa unidade apresentou uma introdução sobre os Sistemas Operacionais para computadores e dispositivos móveis. Foi possível perceber a importância de se conhecer o surgimento e evolução do sistema operacional, particularmente para dispositivos móveis.

Os sistemas operacionais estão presentes nos mais diversos dispositivos, desde computadores pessoais até dispositivos móveis. O sistema operacional atua como uma camada de *software* entre os programas aplicativos dos usuários finais e o *hardware*. Ele é uma estrutura complexa de *software*, bastante ampla, que incorpora aspectos de alto nível, como a interface gráfica e os programas utilitários, e de baixo nível, como a gerência de memória e os drivers de dispositivos.

Foi possível observar que o sistema operacional atua como uma camada transparente para o usuário permitindo que o computador seja utilizado de forma conveniente e eficiente, ocultando a complexidade do *hardware*. O sistema operacional como gerente de recursos faz a gestão do uso do processador, do espaço em memória, do acesso à arquivos, conexões de rede e dispositivos externos, sempre buscando evitar conflitos.

A diferenciação entre o segmento móvel e a computação tradicional (desktop e servidor) não está no próprio sistema operacional, mas em camadas mais altas. A diferença vai muito além de uma mudança na interface do usuário e passa por características marcantes como armazenamento e consumo de energia.

Na próxima unidade vamos entender o funcionamento dos sistemas operacionais e explorar mais a sua arquitetura.

Glossário

***Batch*:** lote, conjunto, agregar as coisas em conjuntos;

***Desktop*:** parte da interface gráfica de sistemas operacionais que exibe, no vídeo, representações de objetos usualmente presentes nas mesas de trabalho, como documentos, arquivos, pastas e impressoras; área de trabalho;

***Hackers*:** em informática, *hacker* é um indivíduo que se dedica, com intensidade incomum, a conhecer e modificar os aspectos mais internos de dispositivos, programas e redes de computadores;

***Hardware*:** conjunto dos componentes físicos (material eletrônico, placas, monitor, equipamentos periféricos etc.) de um computador;

Interface gráfica: é um conceito da forma de interação entre o usuário do computador e um programa por meio de uma tela ou representação gráfica, visual, com desenhos, imagens, etc;

***Job*:** trabalho ou tarefa;

Linguagem de máquina: um programa em código de máquina consiste de uma sequência de bytes que se tratam de instruções a serem executadas pelo processador;

***Mobile*:** que se move; móvel, móbil;

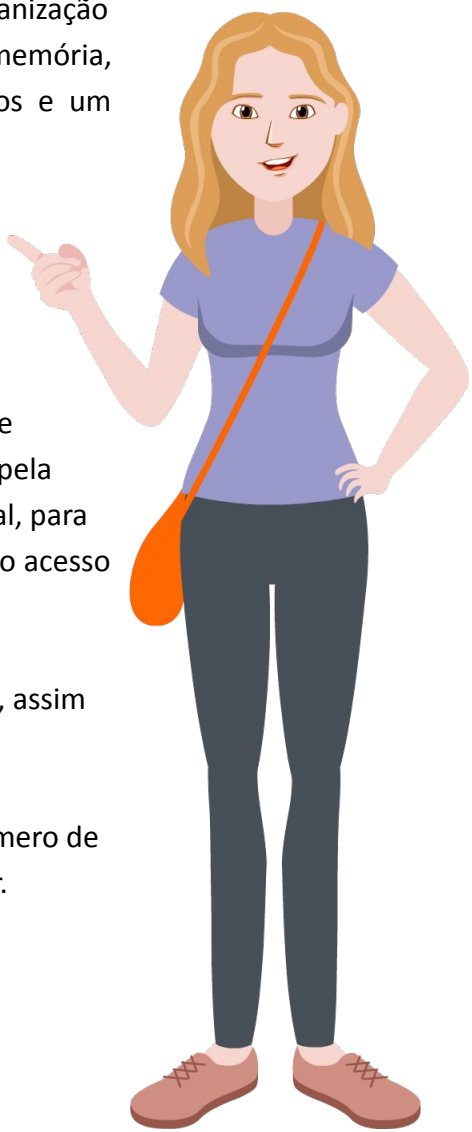
***Timesharing*:** tempo compartilhado;

***Worms*:** (termo da língua inglesa que significa, literalmente, "verme") é um programa autorreplicante, diferente de um vírus.

Unidade 2 ENTENDENDO OS SISTEMAS OPERACIONAIS

Como vimos, o sistema operacional é uma camada de *software* que atua entre o *hardware* e os programas aplicativos, utilizados pelos usuários finais.

Todo sistema operacional possui aspectos básicos em sua organização sendo constituído normalmente por: um núcleo, um gerenciador de memória, um gerenciador de E/S (entradas e saídas), um sistema de arquivos e um processador de comandos/interface com o usuário.



O núcleo é responsável pela gerência do processador, tratamento de interrupções, comunicação e sincronização entre processos, enquanto o gerenciador de memória é responsável pelo controle e alocação de memória aos processos ativos.

O gerenciador de entradas e saídas é responsável pelo controle e execução de operações de E/S, pela otimização do uso de periféricos e pela interface de comunicação com o usuário. Dentro do sistema operacional, para manipular informações, existe um sistema de arquivos encarregado pelo acesso e integridade dos dados residentes na memória secundária (discos e dispositivos de armazenamento).

O processador de comandos atua como interface com o usuário, assim como o processador de E/S, também, é responsável pela interface de comunicação com o usuário.

Os sistemas operacionais podem ser classificados quanto ao número de usuários e, também, quanto ao número de tarefas que podem executar. Vejamos:

Quanto ao número de usuários:

- Monousuário:
  - Projetados para suportar um único usuário.
  - Ex: MS-DOS, *Windows* 3x, *Windows* 9x.

- Multiusuário:
  - Projetados para suportar várias sessões de usuários.
  - Ex: *Windows* NT(2000), UNIX.

Quanto ao número de tarefas:

- Monotarefa:
  - Capazes de executarem apenas uma tarefa (um aplicativo) de cada vez;
  - Os recursos computacionais estão inteiramente dedicados a um único programa/tarefa;
  - A UCP fica ociosa durante muito tempo enquanto o programa aguarda por um evento (digitação de um dado, leitura do disco, etc.);
  - A memória principal é subutilizada caso o programa não a preencha totalmente;
  - Os periféricos são dedicados a um único usuário;
  - Não existem grandes preocupações com a proteção de memória;
  - A complexidade de implementação é relativamente baixa. Ex: MS-DOS

- Multitarefa:
    - Capazes de executarem várias atividades simultaneamente, como uma compilação e um processamento de texto;
    - Vários programas competem pelos recursos do sistema;
    - O objetivo é manter mais de um programa em execução “simultaneamente”, dando a ilusão de que cada programa/usuário tem a máquina dedicada para si;
    - A ideia é tirar proveito do tempo ocioso da UCP durante as operações de E/S. Enquanto um programa espera por uma operação de leitura ou escrita, os outros podem serem processados no mesmo intervalo;
        - Maximização do uso do processador e da memória;
        - Maior taxa de utilização do sistema como um todo (redução do custo total máquina/homem);
    - Suporte de *hardware*:
        - Proteção de memória;
        - Mecanismo de interrupção (sinalização de eventos);
        - Discos magnéticos (acesso randômico aos programas, melhor desempenho em operações de E/S) para implementação de memória virtual;

    Ex: *Windows*, OS/2, Unix.

2.1 Abstração e Gerência de Recursos

Dentro dos sistemas computacionais, cada hardware tem suas peculiaridades e cabe ao sistema operacional gerenciar essas diferenças de forma transparente. Por exemplo, um processador de textos (software) não necessita saber como ocorre o processo de acesso e gravação de um arquivo (hardware). Ele não deve se preocupar em como os dispositivos são acessados.

Compete ao sistema operacional prover interfaces de acesso aos dispositivos, facilitando aos softwares formas mais simples de usar esses recursos que as interfaces de baixo nível. Neste processo o sistema operacional tornará os aplicativos independentes do hardware, ou seja, ele irá definir interfaces de acesso homogêneas (padronizadas) para dispositivos com tecnologias distintas (diferentes).

O sistema operacional também é o responsável pela definição das políticas para o gerenciamento do uso dos recursos de hardware pelos aplicativos (softwares), efetuando a resolução de possíveis disputas e conflitos que possam ocorrer. Imagine que dois softwares querem, por exemplo, acessar um arquivo ao mesmo tempo, em um único dispositivo de leitura. Será o sistema operacional que fará a mediação do acesso e uso do processador, do disco e da memória, por exemplo.

2.2 Tipos de sistemas operacionais

Batch

Nos sistemas operacionais do tipo *batch* todos os programas para execução são colocados em uma fila. Assim, o processador recebia um programa após o outro, realizando o processamento em sequência, ampliando o grau (capacidade) de utilização do sistema. Atualmente, os sistemas operacionais em *batch* são pouco utilizados, porém o termo "lote" é empregado até hoje. Esse termo ainda é utilizado para definir um conjunto de comandos que são executados sem a interferência do usuário.

b) Rede

Os sistemas operacionais em rede tem suporte nativo para a operação em rede e é a maioria dos sistemas operacionais atuais. Podem ser utilizados para o compartilhamento de recursos de vários computadores e, também, podem compartilhar os seus próprios recursos. Atuam de forma independente, e caso a conexão entre um dos nós sofra qualquer problema, os demais continuam operando normalmente. É claro que com a desconexão de um nós alguns recursos poderão se tornar indisponíveis, mas isso não irá interferir no funcionamento dos demais recursos.

c) Distribuído

O sistema operacional distribuído apresenta a característica principal de que os recursos de cada máquina estão disponíveis globalmente, de forma transparente aos usuários. Assim, o sistema operacional distribuído se apresenta para o usuário e suas aplicações como um único sistema centralizado. Para o usuário e aplicações é como se não existisse uma rede de computadores, ou seja, o usuário não percebe qual computador da rede está utilizando. Esse tipo de sistema operacional ainda não é uma realidade de mercado. Como exemplo podemos citar o sistema operacional Amoeba[1], que pode ser executado em diversas plataformas, incluindo SPARC, i386, 68030, Sun 3/50 e Sun 3/60, utilizando o *FLIP* (Fast-Local-Internet-Protocol) como protocolo de rede. Trata-se de uma suíte de protocolos *Internet* que provê transparência, segurança e gerenciamento de rede. É nesse sistema operacional que a linguagem de programação *Python* foi originalmente desenvolvida.

d) Multiusuário

A maioria dos sistemas operacionais atuais é considerada multiusuário, pois permite a utilização por múltiplos usuários simultâneos. Esse tipo de sistema operacional deve suportar a identificação do "dono" de cada recurso dentro do sistema, ou seja, os recursos como arquivos, processos e até as conexões de rede devem ser executados com a identificação inequívoca a qual o usuário pertence. Naturalmente, por questões de segurança, esse tipo de sistema operacional irá implementar (impor) regras de controle de acesso para impedir o uso desses recursos por usuários não autorizados.

e) Desktop

Os sistemas operacionais para *desktop* são conhecidos como sistema operacional "de mesa", pois normalmente são aplicados em computadores pessoais (usuários domésticos), portáteis (*notebooks*) ou *desktops* comuns (corporativos). São sistemas operacionais que dão suporte a atividades comuns, tradicionais e corriqueiras. Possuem ambiente gráfico (interface) amigável, com suporte a rede (conectividade) e grande nível de interatividade.



f) Servidor

Os sistemas operacionais para servidor trabalham com a gestão de grandes quantidades de recursos, como discos, memórias e processadores. São preparados para trabalhar com múltiplos usuários conectados ao mesmo tempo, até mesmo por diferentes meios (conexões remotas e locais). Neste tipo de sistema operacional o suporte a rede é nativo, ou seja, é parte integrante do sistema operacional não havendo versão sem esse tipo de suporte.

Atualmente, é uma prática comum de mercado desenvolver sistemas operacionais que possuem versões para *Desktop* e para Servidores (Ex: *Ubuntu* Server ou *Windows Server*).

g) Embutido

Os sistemas operacionais embutidos são aqueles instalados em *hardwares* com pouca capacidade de processamento (adaptados), como celulares (*smartphones*), câmeras, GPS, calculadoras, tocadores de MP3, PDAs, *tablets* e outros pequenos dispositivos. Normalmente são aplicados em *hardwares* que possuem funções específicas, tendo memória limitada, processador mais lento e *display* de pequenas dimensões. Neste caso, tanto o sistema operacional e as aplicações são projetados para minimizar o uso do processador (redução do consumo da bateria). Podem fazer uso de tecnologias *wireless*, como *Bluetooth* e *Wi-fi*, para acesso remoto a *e-mail* e navegação *Web*. Porém, é importante ressaltar que alguns dispositivos móveis e *hardwares* para ambientes de testes e simulações cresceram tanto em capacidade de memória e processamento que estão utilizando versões de sistemas operacionais semelhantes ou até iguais aos utilizados em *Desktop*, como, por exemplo, os celulares (*smartphones*).

h) Tempo real

Os Sistemas Operacionais de Tempo Real são sistemas utilizados quando há necessidade de um comportamento temporal previsível, ou seja, é importante considerar o tempo (parâmetro fundamental) de resposta no melhor caso e pior caso de operação.

Essa categoria de sistema operacional possui dois tipos:

- *soft real-time systems (ou sistema de tempo real não crítico)* - neste tipo de sistema operacional de tempo real o descumprimento do prazo é aceitável e não causa dano permanente. Temos como exemplo os sistemas de áudio digital, multimídia e telefones digitais. Nestes casos, a perda de prazo implica em degradação do serviço prestado (gravação de CD);
- *hard real-time systems (ou sistema de tempo real crítico)* - neste outro tipo de sistema operacional de tempo real a perda de prazo pode causar grandes prejuízos econômicos ou ambientais (usina nuclear, caldeiras industriais). Essa categoria de sistema operacional deve fornecer garantia absoluta de que determinada ação ocorrerá em determinado momento.

2.3 Funcionalidades e conceitos de *hardware*

Além das funcionalidades básicas oferecidas pela maioria dos sistemas operacionais, várias outras vêm se agregar aos sistemas modernos, para cobrir aspectos complementares, como a interface gráfica, suporte de rede, fluxos multimídia, gerência de energia, etc.

As funcionalidades do sistema operacional geralmente são interdependentes: por exemplo, a gerência do processador depende de aspectos da gerência de memória, assim como a gerência de memória depende da gerência de dispositivos e da gerência de proteção. Alguns autores [Silberschatz et al., 2001, Tanenbaum, 2003] representam a estrutura do sistema operacional conforme indicado na Figura 1. Nela, o núcleo central implementa o acesso de baixo nível ao *hardware*, enquanto os módulos externos representam as várias funcionalidades do sistema.

Um sistema operacional possui diversas funcionalidades e deve atuar em muitas frentes. O sistema possui inúmeros recursos que devem ser gerenciados, como por exemplo, processador, memória, dispositivos físicos, arquivos e proteção.

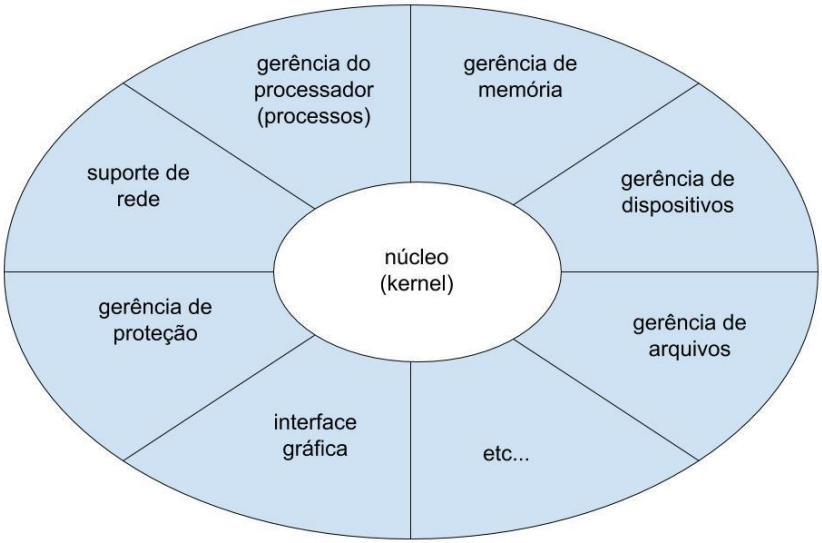


Figura 1 - Os módulos de gerência do sistema operacional são interdependentes.
Fonte: : [Silberschatz et al., 2001, Tanenbaum, 2003]

Uma regra importante a ser observada na construção de um sistema operacional é a separação entre os conceitos de política e mecanismo. Como política consideram-se os aspectos de decisão mais abstratos, que podem ser resolvidos por algoritmos de nível mais alto, como por exemplo, decidir a quantidade de memória que cada aplicação ativa deve receber, ou qual o próximo pacote de rede a enviar para satisfazer determinadas especificações de qualidade de serviço.

Por outro lado, como mecanismo consideram-se os procedimentos de baixo nível usados para implementar as políticas, ou seja, atribuir ou retirar memória de uma aplicação, enviar ou receber um pacote de rede, etc. Os mecanismos devem ser suficientemente genéricos para suportar mudanças de política sem necessidade de modificações. Essa separação entre os conceitos de política e mecanismo traz uma grande flexibilidade aos sistemas operacionais, permitindo alterar sua personalidade (sistemas mais interativos ou mais eficientes) sem ter de alterar o código que interage diretamente com o *hardware*.

O sistema operacional interage diretamente com o *hardware* disponibilizando serviços às aplicações. Ainda hoje, a grande maioria dos computadores com apenas um processador segue a arquitetura básica definida por János (John) Von Neumann, a cerca de 40 anos, intitulada como “arquitetura Von Neumann”. A principal característica desse modelo é que o programa a ser executado reside na memória junto com os dados, conhecida como “programa armazenado”.

É por meio de um ou mais barramentos (para a transferência de dados, endereços e sinais de controle) que os principais elementos constituintes do computador estão interligados.

A composição normal de um sistema computacional típico é possuir um ou mais processadores com a responsabilidade de executar as instruções das aplicações, armazenadas em uma área de memória que contém as aplicações em execução (seus códigos e dados) assim como dispositivos periféricos que permitem o armazenamento de dados e a comunicação com o mundo exterior, como discos rígidos, terminais e teclados.

A parte central de um sistema computacional é o processador o qual tem a função de continuamente ler instruções e dados da memória ou de periféricos, fazer o processamento e encaminhar novamente o resultado para a memória ou a outros dispositivos (periféricos). O processador contém a unidade lógica aritmética (ULA) para fazer operações lógicas e cálculos, alguns registradores especiais (ponteiro de pilha, contador de programa, *flags* de status, etc.) e um grupo de registradores para armazenar dados de trabalho.

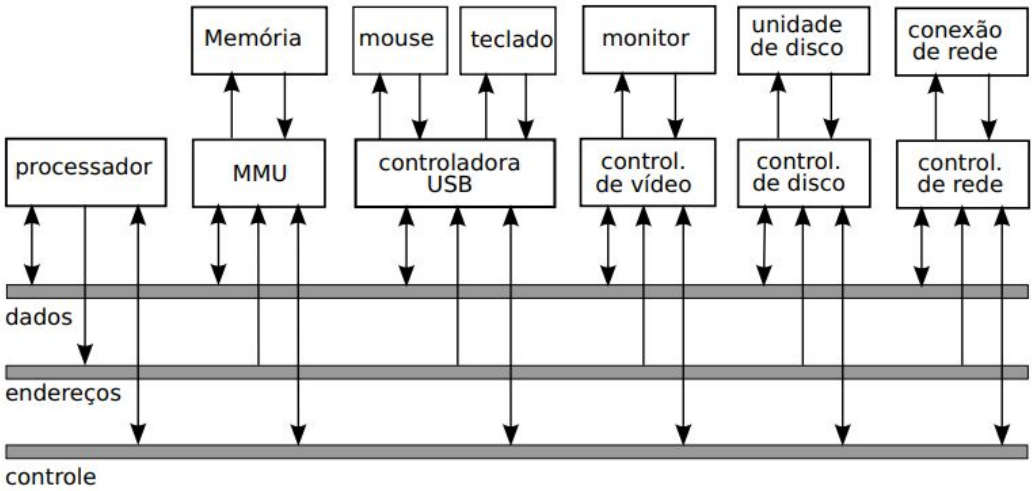


Figura 2 - Arquitetura de um computador típico.
Fonte: Maziero (2017, p.10)

Os barramentos são responsáveis por realizar todas as transferências de dados entre o processador, a memória e os periféricos. Temos:

- o barramento de endereços indica a posição de memória (ou o dispositivo) a acessar;
- o barramento de controle indica a operação a efetuar (leitura ou escrita); e
- o barramento de dados transporta a informação indicada entre o processador e a memória ou um controlador de dispositivo.

Normalmente, a mediação de acesso à memória é realizada por um controlador específico (que pode estar fisicamente dentro do próprio processador) chamado Unidade de Gerência de Memória (MMU - *Memory Management Unit*). Sua função é analisar cada endereço solicitado pelo processador, validá-los, efetuar as conversões de endereçamento necessárias e executar a operação solicitada pelo processador (leitura ou escrita de uma posição de memória).

Os controladores são circuitos específicos que permitem o acesso aos periféricos do computador (discos, teclado, monitor, etc.). Assim, o controlador USB permite acesso ao *mouse*, teclado e outros dispositivos USB externos, a placa de vídeo permite o acesso ao monitor e a placa *ethernet* dá acesso à rede.

O processador vê cada dispositivo representado por seu respectivo controlador. As portas de entrada/saída endereçáveis dão acesso aos controladores. Assim, a cada controlador é atribuída uma faixa de endereços de portas de entrada/saída.

Temos duas alternativas de comunicação quando um controlador de periférico tem uma informação importante a fornecer ao processador:

- Informar o processador através do barramento de controle, enviando a ele uma requisição de interrupção (IRQ – *Interrupt ReQuest*);

- Esperar até que o processador o consulte, o que poderá ser demorado caso o processador esteja ocupado com outras tarefas (o que geralmente ocorre).

Quando os circuitos do processador recebem uma requisição de interrupção, eles suspendem seu fluxo de execução corrente e desviam para um endereço pré-definido, onde está uma rotina de tratamento de interrupção (*interrupt handler*). A rotina é responsável por tratar a interrupção, ou seja, executar as ações necessárias para atender ao dispositivo que a gerou. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando recebeu a requisição.

É função do sistema operacional gerenciar os recursos do *hardware*, e fornecê-los às aplicações de acordo com as suas necessidades. Para garantir a integridade dessa gerência, é primordial garantir que as aplicações não consigam acessar o *hardware* diretamente. Esse acesso só deve ocorrer por meio de pedidos ao sistema operacional, que irá avaliar e intermediar todos os acessos ao *hardware*.

Mas como impedir que o *hardware* seja acessado diretamente pelas aplicações?

Para permitir diferenciar os privilégios de execução dos diferentes tipos de *software*, os processadores modernos contam com dois ou mais níveis de privilégio de execução. Esses níveis são controlados por *flags* especiais nos processadores, e as formas de mudança de um nível de execução para outro são controladas estritamente pelo processador.

Podemos considerar dois níveis básicos de privilégio:

- Nível núcleo: também denominado nível supervisor, sistema, monitor ou ainda *kernel space*. Para um código executando nesse nível, todo o processador está acessível: todos os recursos internos do processador (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Além disso, todas as instruções do processador podem ser executadas. Ao ser ligado, o processador entra em operação neste nível.

- Nível usuário (ou *userspace*): neste nível, somente um subconjunto das instruções do processador, registradores e portas de entrada/saída estão disponíveis. Instruções "perigosas" como *HALT* (parar o processador) e *RESET* (reiniciar o processador) são proibidas para todo código executando neste nível. Além disso, o *hardware* restringe o uso da memória, permitindo o acesso somente a áreas previamente definidas. Caso o código em execução tente executar uma instrução proibida ou acessar uma área de memória inacessível, o *hardware* irá gerar uma exceção, desviando a execução para uma rotina de tratamento dentro do núcleo, que provavelmente irá abortar o programa em execução (e também gerar a famosa frase "este programa executou uma instrução ilegal e será finalizado", no caso do *Windows*).

É fácil perceber que, em um sistema operacional convencional, o núcleo e os drivers operam no nível núcleo, enquanto os utilitários e as aplicações operam no nível usuário, confinados em áreas de memória distintas, conforme ilustrado na Figura 3.

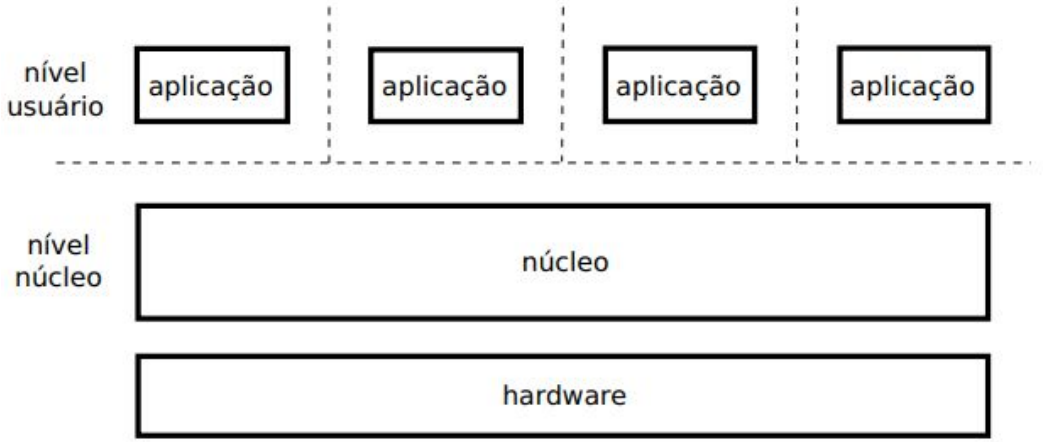


Figura 3 - Separação entre o núcleo e as aplicações
Fonte: Maziero (2017, p.15)

Os mapeamentos de memória realizados pela MMU nos acessos em nível usuário impõem o confinamento de cada aplicação em sua área de memória, provendo robustez e confiabilidade ao sistema, já que garante que uma aplicação não poderá interferir nas áreas de memória de outras aplicações ou do núcleo.

Porém, essa proteção introduz um novo problema: como acessar, a partir de uma aplicação, as rotinas oferecidas pelo *kernel* (núcleo) para o acesso ao *hardware* e suas abstrações?

Em outras palavras, como uma aplicação sem ter privilégio, para acessar as portas de entrada/saída correspondentes e sem poder invocar o código do núcleo que implementa esse acesso (pois esse código reside em outra área de memória), poderá acessar a placa de rede para enviar/receber dados?

A solução para esse contratempo está no mecanismo de interrupção. Uma instrução especial, implementada pelos processadores, permite acionar o mecanismo de interrupção de forma intencional, sem depender de eventos externos ou internos.

As chamadas de sistema (*system call ou syscall*) são assim denominadas porque são realizadas pela ativação de procedimentos do núcleo usando interrupções de *software* (ou outros mecanismos correlatos).

Todas as operações envolvendo abstrações lógicas (criação e finalização de tarefas, operadores de sincronização e comunicação, etc.) ou o acesso a recursos de baixo nível (periféricos, arquivos, alocação de memória, etc.) são definidas por chamadas de sistema pelos sistemas operacionais.

É por meio de uma biblioteca do sistema (*system library*), que prepara os parâmetros, invoca a interrupção de software e retorna à aplicação os resultados obtidos, que normalmente as chamadas de sistema são oferecidas para as aplicações em modo usuário.

Centenas de chamadas de sistema distintas, para as mais diversas finalidades são implementadas pela maioria dos sistemas operacionais.

A API (*Application Programming Interface*) de um sistema operacional é definida pelo conjunto de chamadas de sistema oferecidas por seu núcleo.

A Win32 é um exemplo de API bem conhecida. Ela é oferecida pelos sistemas *Microsoft* derivados do *Windows* NT. Outro exemplo é a API POSIX que define um padrão de interface de núcleo para sistemas UNIX.

2.4 Estrutura de um sistema operacional

Não podemos considerar o sistema operacional como um bloco de *software* fechado e único rodando sobre o *hardware*. Sua composição real vai além disso. Ele é composto por diversos componentes, em que cada um possui um objetivo e uma funcionalidade complementar. A tabela a seguir apresenta os componentes mais importantes de um sistema operacional típico.

Núcleo	É a parte central (coração) do sistema operacional responsável por gerenciar os recursos do *hardware* que são utilizados pelas aplicações. Implementa as principais abstrações que os programas aplicativos utilizam.
Drivers	São módulos específicos de código que permitem acessar determinados dispositivos físicos. Cada dispositivo (discos rígidos IDE, SCSI, portas USB, placas de vídeo, etc.) necessita de um *driver* específico. O *driver* normalmente é construído pelo próprio fabricante do *hardware*. É disponibilizado em linguagem de máquina (de forma compilada) para ser acoplado ao sistema operacional permitindo o uso do dispositivo físico.
Código de inicialização	É responsável por realizar o carregamento do núcleo do sistema operacional em memória e começar a sua execução. Para inicializar o *hardware* necessitamos de um conjunto de tarefas bastante complexas, como reconhecer os dispositivos instalados, configurá-los e testá-los de forma correta para seu uso futuro.
Programas utilitários	São aplicativos (programas) que permitem o uso simples e fácil do sistema operacional, disponibilizando funcionalidades complementares ao núcleo (*kernel*) permitindo a formatação de discos e mídias, manipulação de arquivos (criar, mover, remover), configuração de novos dispositivos, terminal, interpretador de comandos, interface gráfica, gerenciador de janelas, entre outros.

Quadro 1 - Principais componentes de um sistema operacional

A Figura 4 apresenta a inter relação entre as diversas partes do sistema operacional. É importante considerar que o relacionamento e a interligação entre esses componentes podem variar de acordo com o sistema operacional, conforme veremos no tópico 2.5 - Arquiteturas de sistemas operacionais.

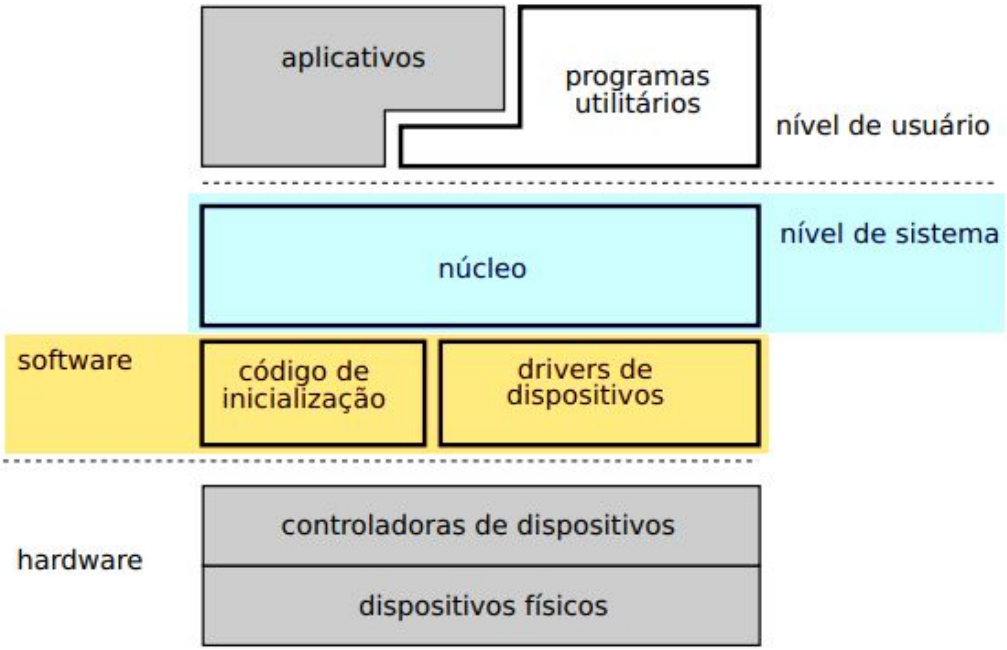


Figura 4 - Estrutura de um sistema operacional
Fonte: Adaptado de Maziero (2017, p.9)

2.5 Arquiteturas de sistemas operacionais

As várias partes que compõem o sistema podem ser organizadas de diversas formas. Podem-se separar as suas funcionalidades e modularizar o seu projeto, mesmo que a definição de níveis de privilégio imponha uma estruturação mínima a um sistema operacional.

Vejamos agora as arquiteturas mais populares para a organização de sistemas operacionais:

a) Sistemas monolíticos

Quando todos os componentes do núcleo operam em modo de núcleo e se inter-relacionam conforme suas necessidades, sem restrições de acesso entre si, pois o código no nível núcleo tem acesso pleno a todos os recursos e áreas de memória, estamos atuando em um sistema operacional monolítico.

Em um sistema operacional monolítico não há barreiras impedindo acessos, e qualquer componente do núcleo pode acessar os demais componentes, toda a memória ou mesmo dispositivos periféricos diretamente. A grande vantagem dessa arquitetura é seu desempenho. Como resultados da interação direta entre componentes têm sistemas mais compactos.

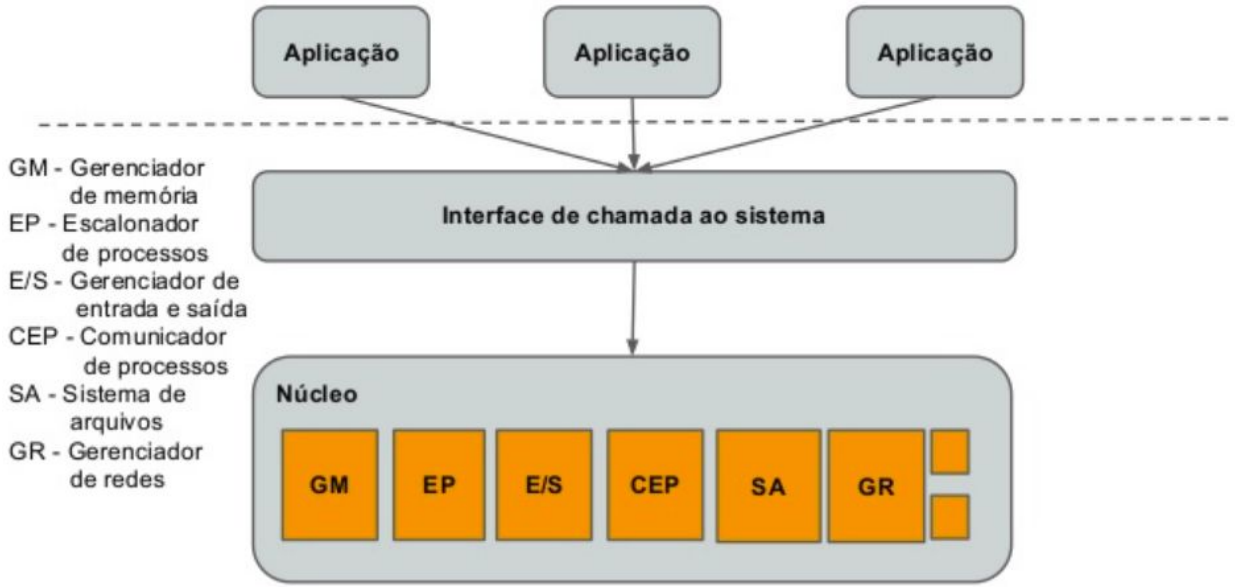


Figura 5 - Uma arquitetura monolítica.
Fonte: elaborado pelo autor.

Os primeiros sistemas operacionais foram organizados utilizando a arquitetura monolítica. Temos como exemplos, os sistemas UNIX antigos e o MS-DOS. Devido às limitações do *hardware* sobre o qual executam apenas os sistemas operacionais embarcados continuam atualmente empregando arquitetura monolítica. Mesmo o núcleo do GNU/*Linux*, que nasceu monolítico, mas vem sendo modificado e modularizado embora uma boa parte de seu código ainda permaneça no nível de núcleo.

b) Sistemas em camadas

Podemos utilizar camadas para estrutura de forma mais elegante um sistema operacional. Enquanto a camada superior define a interface do núcleo para as aplicações (as chamadas de sistema), as camadas intermediárias provêm níveis de abstração e gerência cada vez mais sofisticados. Por fim, a camada mais baixa realiza a interface com o *hardware*.

O modelo de referência OSI (*Open Systems Interconnection*) implementou essa abordagem de estruturação de *software* fazendo muito sucesso no domínio das redes de computadores, logo seria algo natural esperar sua adoção no domínio dos sistemas operacionais.

Porém, não foi bem assim, pois alguns inconvenientes limitam sua aceitação nesse contexto. Vejamos:

- O pedido de uma aplicação demora mais tempo para chegar até o dispositivo periférico ou recurso a ser acessado, devido ao empilhamento de várias camadas de *software* prejudicando o desempenho do sistema;

- Não é óbvio como dividir as funcionalidades de um núcleo de sistema operacional em camadas horizontais de abstração crescente, pois essas funcionalidades são interdependentes, embora tratem muitas vezes de recursos distintos.

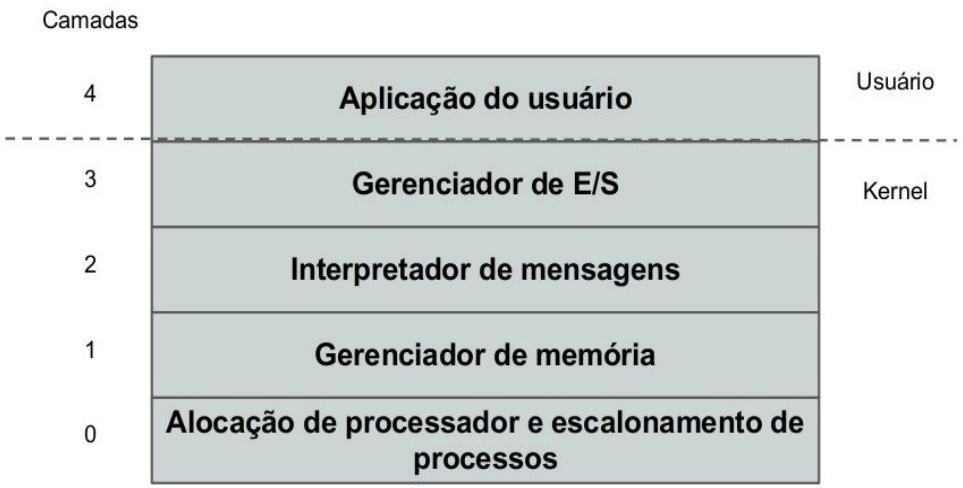


Figura 6 - Uma arquitetura em camadas.
Fonte: elaborado pelo autor.

A estruturação em camadas é adotada apenas parcialmente hoje em dia, em decorrência desses inconvenientes. Assim como o *Windows* NT e seus sucessores que utilizavam a camada HAL – *Hardware Abstraction Layer*, muitos sistemas implementam uma camada inferior de abstração do *hardware* para interagir com os dispositivos. Como exemplos de sistemas fortemente estruturados em camadas temos o IBM OS/2 e o MULTICS. Esses sistemas operacionais também organizam em camadas alguns subsistemas como a gerência de arquivos e o suporte de rede (seguindo o modelo OSI).

c) Sistemas micronúcleo (microkernel)

Essa abordagem torna os núcleos de sistema menores. Por isso ela foi chamada de micronúcleo (ou µ-kernel). Os sistemas operacionais *microkernel* retiraram do núcleo todo o código de alto nível (normalmente associado às políticas de gerência de recursos). É outra possibilidade de estruturação em oposição aos sistemas monolíticos. Os sistemas operacionais micronúcleo deixam dentro do núcleo somente o código de baixo nível necessário para interagir com o *hardware* e criar as abstrações fundamentais (como a noção de atividade).

Nesse tipo de abordagem, *microkernel*, o código de acesso aos blocos de um disco rígido seria mantido no núcleo. Já as abstrações de arquivo e diretório seriam criadas e mantidas por um código fora do núcleo, executando da mesma forma que uma aplicação do usuário.

Somente a noção de atividade, de espaços de memória protegidos e de comunicação entre atividades é implementada normalmente em um micronúcleo.

Políticas de uso do processador e da memória, o sistema de arquivos e o controle de acesso aos recursos, considerados aspectos de alto nível, são implementados fora do núcleo, em processos que se comunicam usando as primitivas do núcleo. A Figura 7 ilustra essa abordagem.

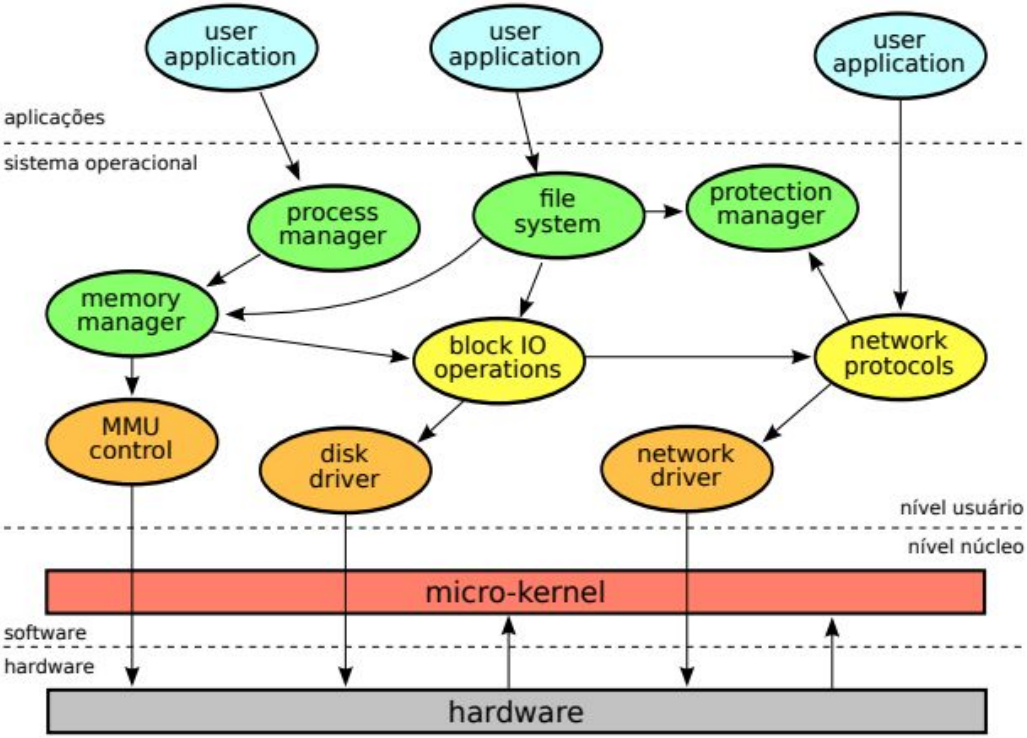


Figura 7 - Visão geral de uma arquitetura micronúcleo.
Fonte: Maziero (2017, p.20)

As interações entre componentes e aplicações, em um sistema micronúcleo, são feitas através de trocas de mensagens. Desta forma, se uma aplicação deseja acessar (abrir) um arquivo no disco rígido, envia uma mensagem para o gerente de arquivos que se comunica com o gerente de dispositivos para obter os blocos de dados relativos ao arquivo desejado.

Devido às restrições impostas pelos mecanismos de proteção do *hardware* os processos não podem se comunicar diretamente. Dessa maneira, todas as mensagens são transmitidas através de serviços do micronúcleo, como mostra a Figura 7.

Essa abordagem também foi chamada cliente/servidor, pois os processos têm de solicitar "serviços" uns dos outros, para poder realizar suas tarefas.



Utilização real do *microkernel*

Durante os anos 80 os micronúcleos foram muito investigados e as principais vantagens são sua robustez e flexibilidade. Os sistemas Mach e Chorus são dois exemplos clássicos de sistemas operacionais *microkernel*. O sistema MacOS X da Apple e o Digital UNIX tem suas raízes no sistema Mach, e adotam parcialmente essa estruturação (*microkernel*). O QNX empregado em sistemas embarcados e de tempo real é um dos poucos exemplos de micronúcleo amplamente utilizado.

Se um subsistema do sistema operacional micronúcleo tiver problemas, os níveis de privilégio e os mecanismos de proteção de memória irão deter o erro, impedindo que o restante do sistema seja afetado pela instabilidade.

Outro aspecto interessante é a possibilidade de customização do sistema operacional, pois é possível iniciar somente os componentes necessários ou mais adequados às aplicações que serão executadas.

Uma desvantagem desse tipo de sistema operacional é o desempenho que fica prejudicado devido ao custo associado às trocas de mensagens entre componentes, diminuindo assim a aceitação desta abordagem.

d) Máquinas virtuais

A execução de programas e bibliotecas em uma determinada plataforma computacional depende sempre da compilação para essa plataforma específica, respeitando o conjunto de chamadas do sistema operacional e o conjunto de instruções do processador.

A analogia de execução de um sistema operacional sobre uma plataforma de *hardware* é a mesma. Ele só poderá ser executado sobre a plataforma se for compatível com ela.

Assim, é importante entender que uma biblioteca só funciona sobre o *hardware* e o sistema operacional para os quais foi projetada. Da mesma forma, as aplicações também têm de obedecer a interfaces pré-definidas. Assim, de forma geral, um sistema operacional só funciona sobre o *hardware* para o qual foi construído.

Quando observamos os sistemas operacionais atuais percebemos que as interfaces de baixo nível são pouco flexíveis. Existem proteções e geralmente não é possível criar novas instruções de processador ou novas chamadas de sistema, ou ainda mudar sua semântica.



Mas como executar um Sistema Operacional sobre um *hardware* para o qual ele não foi criado?

Para resolver essa questão **podemos utilizar técnicas de virtualização** e contornar os problemas de compatibilidade entre os componentes de um sistema.

E o que é Virtualização afinal?
Virtualização (em computação) é a criação de uma versão virtual de alguma coisa, como um sistema operacional, um servidor, um dispositivo de armazenamento (*storage*) ou recurso de rede.

Vamos saber mais?
Acesse:
http://www.jvasconcellos.com.br/unijorge/wp-content/uploads/2012/01/cap4-v2.pdf

Podemos criar uma camada intermediária de *software* que disponibilize aos demais componentes serviços com outra interface. Isso é possível usando os serviços oferecidos por um determinado componente do sistema.

O sistema computacional visto através dessa camada é denominado máquina virtual, uma vez que por meio dessa camada de compatibilidade podemos acoplar interfaces distintas permitindo que um programa desenvolvido para uma plataforma A possa ser executado sobre uma plataforma distinta B.

Um exemplo de máquina virtual pode ser observado na Figura 8. Neste exemplo temos uma camada de virtualização sobre uma plataforma de *hardware Sparc* permitindo a execução de um sistema operacional *Windows* e suas aplicações, distinta daquela para a qual foi projetado (Intel/AMD).

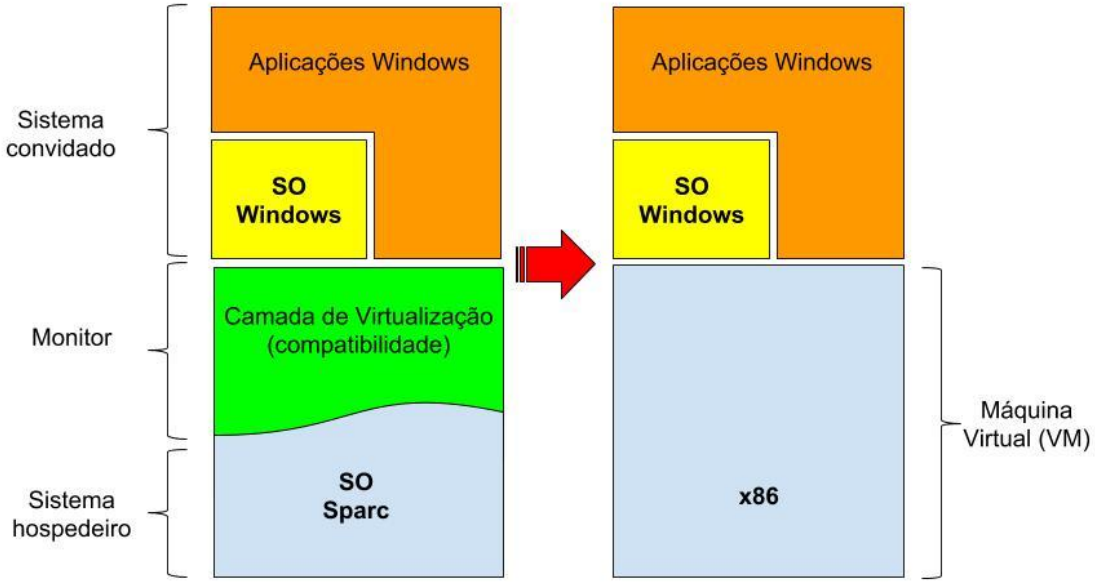


Figura 8 - Visão geral de uma máquina virtual.

Fonte: elaborada pelo autor.

Observando a Figura 8, podemos ver que um ambiente de máquina virtual (VM) consiste de três partes básicas:

a) **o sistema hospedeiro (*host system*)**, ou seja, o sistema real que contém os recursos reais de *hardware* e *software* do sistema;

b) **o sistema convidado (*guest system*)**, ou seja, o sistema virtual que executa sobre o sistema real; em alguns casos, vários sistemas virtuais podem coexistir, executando sobre o mesmo sistema real;

c) **o hipervisor** ou **monitor de virtualização (VMM - *Virtual Machine Monitor*)**, ou seja, a camada de virtualização que constrói as interfaces virtuais a partir da interface real.



Como podemos fazer virtualização em nosso *desktop*?

Separamos um vídeo aula sobre virtualização, preparada pelo Professor Ramos em seu canal. Vamos conhecer conceitos básicos sobre Virtualização e a Instalação do Virtual BOX e do Pacote de Extensão do mesmo.

Vamos aprender um pouco mais sobre o tema?

Acesse:
https://www.youtube.com/watch?v=_7vCg7gKTTU

Essa é apenas uma pitada de virtualização. Veremos mais detalhes nas próximas unidades. Agora, vamos retornar o nosso olhar para as gerências dos sistemas operacionais e aprofundar um pouco mais.

Os sistemas operacionais modernos possuem outras gerências, além dessas que vamos apresentar a seguir. Pode existir gerência de proteção, por exemplo, a qual define políticas de acesso para os sistemas em rede e multiusuários, criando usuários e grupos e suas permissões, além de registrar os recursos por usuários. Podemos ter ainda gerência de energia (dispositivos móveis, por exemplo), gerência de rede e de recursos multimídia. Separamos as mais importantes e comuns a todos os sistemas operacionais para estudar.

2.6 Gerência de processos

A gerência de processos é conhecida também como gerência de processador ou de atividades, pois é responsável por coordenar os processos e atividades em execução. A gerência do processador é necessária para distribuir a capacidade de processamento de forma justa. É importante lembrar que atuar de forma justa é diferente de igual.

A sincronização de atividades, assim como a priorização, é importante para evitar conflitos e dependem de uma comunicação eficiente entre os processos que estão sendo executados.

Sobre a gerência de atividades de um sistema operacional, Maziero (2017, p.27) afirma que:

> "Um sistema de computação **quase sempre tem mais atividades a executar que o número de processadores disponíveis**. Assim, **é necessário criar métodos para multiplexar o(s) processador(es) da máquina entre as atividades presentes**. Além disso, como as diferentes tarefas têm necessidades distintas de processamento, e nem sempre a capacidade de processamento existente é suficiente para atender a todos, **estratégias precisam ser definidas para que cada tarefa receba uma quantidade de processamento que atenda suas necessidades**." (grifo nosso).

Precisamos compreender o conceito de programa e tarefa, pois são conceitos diferentes já que possuem estados diferentes. Vejamos:

De um lado está o programa que representa um conceito estático, e pode ser entendido como um conjunto de uma ou mais sequências de instruções (comandos de linguagem) escritas para resolver um problema específico, constituindo assim um *software*, uma aplicação ou utilitário.

Do outro lado, está a tarefa que representa um conceito dinâmico, pois possui um estado interno definido a cada instante, interagindo com usuários, periféricos ou outras tarefas. A tarefa, executada pelo processador, é compreendida como as sequências de instruções definidas em um programa para realizar seu objetivo.

Sobre a execução de tarefas, Maziero (2017, p. 29) afirma que:

> "[...] em um computador, o processador tem de executar todas as tarefas submetidas pelos usuários. Essas tarefas geralmente têm comportamento, duração e importância distintas. Cabe ao sistema operacional organizar as tarefas para executá-las e decidir em que ordem fazê-lo."

A organização básica do sistema de gerência de tarefas evoluiu conforme evoluíram os sistemas operacionais. Desta forma, temos o sistema monotarefa, o sistema multitarefa e o sistema de tempo compartilhado.

No sistema monotarefa, bastante primitivo, cada programa binário era carregado do disco para a memória e executado até sua conclusão. Sua aplicação era basicamente cálculos numéricos, normalmente com fins militares, e esses sistemas eram dependentes da interação de um operador humano. Para diminuir essa interação surgiu a figura de um programa monitor que era executado para gerenciar a fila de execução de programas, armazenados em disco. O monitor é considerado o precursor dos sistemas operacionais.

Com a evolução do *hardware* surgiram os sistemas multitarefas, já que a velocidade do processador era maior que a velocidade de comunicação com os dispositivos de entrada e saída e o processador ficava ocioso nos momentos de transferência de informação entre a memória e o disco. Assim, a solução foi permitir ao processador suspender uma tarefa enquanto aguardava a transferência de dados externas e passar a executar outra tarefa. Quando finalizasse a transferência de dados da tarefa suspensa o processador retomava a sua execução do ponto que parou. Esse processo de alternância entre tarefas demandava mais memória, e também a necessidade de definir mecanismos para suspender e retomar tarefas. Aqui, surgiu a preempção que é o ato de retirar recursos de uma tarefa (neste caso o processador). Sistemas que implementam esse conceito são conhecidos como *sistemas preemptivos*, e são bem mais produtivos, pois conseguem executar várias tarefas ao mesmo tempo.

Os sistemas de tempo compartilhado surgiram para evitar os erros de *loop infinito* (onde o sistema nunca termina de executar uma tarefa) e para permitir a criação de programas com interatividade. Esse novo conceito de sistema utilizava o *time-sharing* (tempo compartilhado) em que a atenção do processador era dividida entre as tarefas (atividades) por um *quantum* (delta ou taxa) de tempo. Esse conceito permitia que ao término desse tempo uma tarefa perdesse a atenção do processador, e este se dedicasse a olhar a fila de tarefas prontas para execução. Esse conceito toma por base as interrupções geradas pelo temporizador programável do *hardware*.

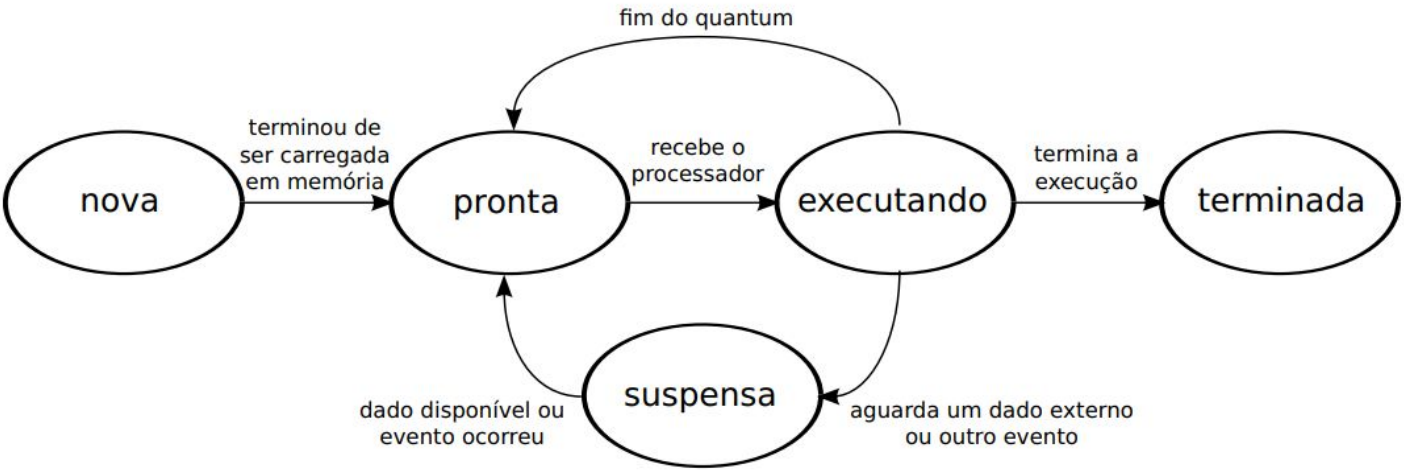


Figura 9 - Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

Fonte: Maziero (2017, p.33)



Como essas tarefas e processos podem ser vistas e gerenciadas em um sistema operacional real?

No *Linux*, um processo é uma instância de um *software* em execução, ou seja, um programa que está sendo usado. Em outros sistemas, os processos também ganham o nome de tarefas (*tasks*). Quem usou ou usa o *Windows* provavelmente já usou o Gerenciador de Tarefas, um aplicativo que permite, entre coisas, encerrar *softwares* que apresentam comportamentos indesejáveis, fazendo o sistema travar como um todo, por exemplo.

Vamos saber mais?
Acesse:
https://canaltech.com.br/linux/conheca-6-comandos-para-gerenciar-processos-do-linux/

2.7 Gerência de memória

A gerência de memória busca fornecer a cada aplicação em execução um espaço próprio de memória, que seja independente e isolado das demais aplicações. Para viabilizar esse processo de forma eficiente a gerência de memória também faz uso do disco como memória complementar, já que a memória principal normalmente tem tamanho menor. Normalmente, apenas as aplicações prioritárias são mantidas na memória principal, de acesso mais rápido, enquanto as que não estão sendo utilizadas no momento (não tem prioridade) podem ser deslocadas para a memória complementar (discos). É importante ressaltar que esse processo é feito de maneira transparente pelo sistema operacional e a aplicação desconhece o tipo de memória em uso.

Sobre a gerência de memória, Maziero (2017, p. 114) afirma que:

> "A memória principal é um componente fundamental em qualquer sistema de computação. Ela constitui o "espaço de trabalho" do sistema, no qual são mantidos os processos, threads, bibliotecas compartilhadas e canais de comunicação, além do próprio núcleo do sistema operacional, com seu código e suas estruturas de dados. O hardware de memória pode ser bastante complexo, envolvendo diversas estruturas, como caches, unidade de gerência, etc, o que exige um esforço de gerência significativo por parte do sistema operacional. Uma gerência adequada da memória é essencial para o bom desempenho de um computador."

2.8 Gerência de entrada e saída

A gerência de entrada e saída também é conhecida como gerência de dispositivos. Seu papel é gerenciar os diversos dispositivos que podem ser conectados ao sistema operacional como *pen drive*, *disquetes*, discos IDE, SCSI, ATA, SATA e etc… Esse acesso facilitado é realizado por meio de *drivers* (ou módulos de kernel no caso do GNU/*Linux*) permitindo o uso de forma comum e transparente.

Sobre a gerência de entrada e saída, Maziero (2017, p. 205) destaca que:

> "Um computador é constituído basicamente de um ou mais processadores, memória RAM e dispositivos de entrada e saída, também chamados de periféricos. **Os dispositivos de entrada/saída permitem a interação do computador com o mundo exterior de várias formas**[...]
>
> Já em ambientes industriais, **é comum encontrar dispositivos de entrada/saída específicos para a monitoração e controle de máquinas e processos de produção**, como tornos de comando numérico, braços robotizados e processos químicos. Por sua vez, o computador embarcado **em um carro conta com dispositivos de entrada para coletar dados do combustível e do funcionamento do motor e dispositivos de saída para controlar a injeção eletrônica e a tração dos pneus,** por exemplo. É bastante óbvio que um computador não tem muita utilidade sem dispositivos periféricos, pois o objetivo básico da imensa maioria dos computadores é receber dados, processá-los e devolver resultados aos seus usuários, sejam eles seres humanos, outros computadores ou processos físicos/químicos externos." (grifo nosso).

2.9 Sistemas de arquivos

A gerência de arquivos é construída sobre a gerência de dispositivos criando uma abstração (tornando transparente e padronizado) de arquivos e diretórios. Essa gerência permite ainda que outros dispositivos possam ser utilizados como arquivos (gravar arquivos em uma saída TCP no UNIX, por exemplo).

Um processo deve ter a capacidade de ler e gravar grande volume de dados em dispositivos de armazenamento de forma permanente. Deve ser capaz, também, de compartilhar esses dados armazenados com outros processos.

Sobre a gerência de arquivos, Maziero (2017, p. 163) afirma que:

> "Um sistema operacional tem por finalidade permitir que os usuários do computador façam a execução de aplicações, como editores de texto, jogos, reprodutores de áudio e vídeo, etc. Essas **aplicações processam informações como textos, músicas e filmes, armazenados sob a forma de arquivos em um disco rígido ou outro meio**." (grifo nosso).

Armazenar e recuperar dados são atividades essenciais para qualquer tipo de aplicação. O sistema operacional (SO) para organizar essas informações de forma estruturada emprega a implementação de arquivos, os quais são gerenciados pelo SO de maneira a facilitar o acesso dos usuários a esses conteúdos.

O sistema de arquivos é a parte mais visível de um SO e a manipulação de arquivos deve ocorrer de forma transparente e uniforme, independente do dispositivo de armazenamento que o SO está utilizando.

A definição de arquivo pode ser compreendida como um conjunto de dados armazenados em um dispositivo físico não volátil, com um nome ou outra referência que permita sua localização posterior. Considerando que um dispositivo de armazenamento pode conter milhões de arquivos é necessário organizar os arquivos em uma estrutura de diretórios que permita o acesso organizado.

Temos um sistema de arquivos quando em um dispositivo fazemos a organização física e lógica dos arquivos e diretórios, ou seja, temos uma imensa estrutura de dados armazenada de forma persistente em um dispositivo físico.

Podemos ter uma diversidade de sistemas de arquivos diferentes. Cada sistema operacional pode utilizar um ou mais sistemas de arquivos diferentes. Como, exemplo, temos o NTFS (nos sistemas *Windows*), Ext2/Ext3/Ext4 (*Linux*), HPFS (MacOS), FFS (Solaris) e FAT (usado em *pendrives* USB, máquinas fotográficas digitais e leitores MP3).

Os atributos de arquivos são informações de controle e dependendo do sistema de arquivos podem variar. Alguns atributos como tamanho, criador, data de criação e proteção, estão presentes na maioria dos sistemas de arquivos. Alguns atributos podem ser modificados apenas pelo sistema operacional, enquanto outros, podem ser modificados pelo usuário.

A depender da forma como os arquivos estão organizados o sistema de arquivos poderá recuperar os arquivos de maneira diferente. O acesso sequencial era utilizado em fitas magnéticas. O acesso direto passou a ser utilizado em discos magnéticos, porém apenas quando os arquivos possuem tamanhos fixos. Finalmente, o acesso indexado, um método mais sofisticado, utiliza índices e chaves para organizar os arquivos. Quando a aplicação precisa de um arquivo ela indica a chave e o SO busca na área de índices, e localiza o ponteiro correto para o arquivo por meio da chave fornecida.

As operações comuns que um sistema de arquivos realiza são: criação, escrita no arquivo, leitura, busca no arquivo, exclusão, truncar um arquivo, anexar e renomear.

A estrutura de diretórios é a forma como o SO organiza logicamente os diferentes arquivos armazenados em um dispositivo físico. Chamamos de diretório a estrutura de dados em que cada entrada guarda informações sobre a localização física, nome, organização e demais atributos. Ao abrir um arquivo o SO procura a sua entrada na estrutura de diretórios, armazenando as informações a respeito dos atributos e localização do arquivo em uma tabela mantida na memória principal.

Um sistema de diretórios normalmente é representado como uma estrutura de diretórios em árvore, conforme a Figura 10.

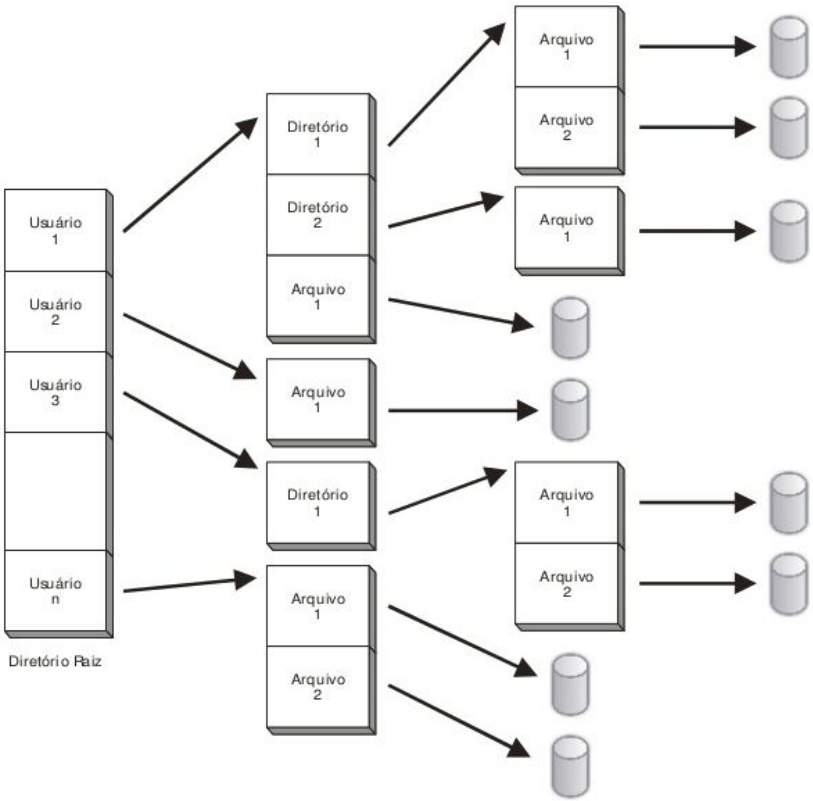


Figura 10 - Estrutura de diretórios em árvore.
Fonte: elaborado pelo autor.



Bora rever!

Essa unidade permitiu aprofundar os conhecimentos sobre os Sistemas Operacionais para computadores e dispositivos móveis. Compreendemos a importância da organização das diversas gerências presentes nos sistemas operacionais. Foi possível observar que todo sistema operacional possui aspectos básicos em sua organização, sendo constituído normalmente por: um núcleo, um gerenciador de memória, um gerenciador de E/S (entradas e saídas), um sistema de arquivos e um processador de comandos/interface com o usuário. Percebemos que os sistemas operacionais podem ser classificados quanto ao número de usuários e, também, quanto ao número de tarefas que podem executar.

Dentro dos sistemas computacionais, cada *hardware* tem suas peculiaridades e cabe ao sistema operacional gerenciar essas diferenças de forma transparente. Por exemplo, um processador de textos (*software*) não necessita saber como ocorre o processo de acesso e gravação de um arquivo (*hardware*). Ele não deve se preocupar em como os dispositivos são acessados.

Compete ao sistema operacional prover interfaces de acesso aos dispositivos, facilitando aos *softwares* formas mais simples de usar esses recursos que as interfaces de baixo nível. Neste processo o sistema operacional tornará os aplicativos independentes do *hardware*, ou seja, ele irá definir interfaces de acesso homogêneas (padronizadas) para dispositivos com tecnologias distintas (diferentes).



Bora rever!

O sistema operacional também é o responsável pela definição das políticas para o gerenciamento do uso dos recursos de *hardware* pelos aplicativos (*softwares*), efetuando a resolução de possíveis disputas e conflitos que possam ocorrer.

Além das funcionalidades básicas oferecidas pela maioria dos sistemas operacionais, várias outras vêm se agregar aos sistemas modernos, para cobrir aspectos complementares, como a interface gráfica, suporte de rede, fluxos multimídia, gerência de energia, etc.

As funcionalidades do sistema operacional geralmente são interdependentes: por exemplo, a gerência do processador depende de aspectos da gerência de memória, assim como a gerência de memória depende da gerência de dispositivos e da gerência de proteção.

Na próxima unidade veremos conceitos básicos do sistema operacional *Android* e como desenvolver aplicativos para diferentes sistemas operacionais. Teremos a oportunidade de aprofundar sobre a arquitetura dos sistemas operacionais para dispositivos móveis (*Android, Apple iOS* e *Windows Phone*).

Glossário

***desktop*:** é uma palavra da língua inglesa que designa o ambiente principal do computador. Literalmente, o termo tem o significado de "em cima da mesa". Era frequentemente utilizado para designar um computador de mesa por oposição ao laptop que é o computador portátil.

E/S: entradas e saídas.

***flags*:** é um mecanismo lógico que funciona como semáforo: uma entidade detém como ativa uma determinada *flag* se a característica associada a essa *flag* estiver presente.

interface: é o nome dado para o modo como ocorre a "comunicação" entre duas partes distintas e que não podem se conectar diretamente.

periféricos: periféricos são aparelhos ou placas de expansão que enviam ou recebem informações do computador.

randômico: aleatório, contingente, fortuito. A palavra aleatoriedade exprime quebra de ordem, propósito, causa, ou imprevisibilidade em uma terminologia não científica. Um processo aleatório é o processo repetitivo cujo resultado não descreve um padrão determinístico, mas segue uma distribuição de probabilidade.

registradores: o registrador ou registo de uma CPU é a memória dentro da própria CPU que armazena n bits. Os registradores estão no topo da hierarquia de memória, sendo assim, é uma mídia mais rápida e financeiramente mais custosa de se armazenar dados.

***wireless*:** ou rede sem fio é uma infraestrutura das comunicações sem fio que permite a transmissão de dados e informações sem a necessidade do uso de cabos.

Unidade 3

ARQUITETURA DOS SISTEMAS OPERACIONAIS DE DISPOSITIVOS MÓVEIS

O sistema operacional para dispositivos móveis é um tipo de sistema operacional desenvolvido para *smartphones*, *tablets*, PDAs ou outros. São dispositivos que possuem tela *touch* e uma série de funcionalidades que realmente os tornam móveis e portáteis.

Ainda que existam dispositivos portáteis que possuem algumas características móveis como *laptops* e notebooks, os sistemas operacionais utilizados nesses equipamentos não são considerados móveis, pois são na maioria das vezes, apenas versões comuns de sistemas operacionais já existentes.

Atualmente, devido à convergência tecnológica, estamos passando por um processo tão grande de miniaturização de componentes que essa distinção está se tornando pouco precisa e, alguns sistemas operacionais mais recentes, estão sendo aplicados tanto em computadores quanto em dispositivos móveis, ou seja, feitos para ambos os tipos de equipamentos. É uma forma de tornar única a experiência do usuário que pode utilizar o mesmo sistema operacional em ambos os equipamentos.

Sistemas operacionais móveis, normalmente, combinam recursos úteis para uso móvel ou portátil (tela sensível ao toque, celular, *Bluetooth*, Wi-Fi, GPS de navegação móvel, câmera fotográfica, câmera de vídeo, reconhecimento de voz e leitor de música), com as características dos sistemas operacionais de um computador pessoal.

O funcionamento das aplicações é influenciado pela arquitetura do sistema operacional a qual, também, interfere no processo de desenvolvimento dessas aplicações.

3.1 Arquitetura do Sistema Android

O sistema Android foi desenvolvido com o conceito de plataforma aberta, baseada no sistema operacional GNU/Linux, o que confere aos fabricantes de dispositivos a liberdade de alterar seu código-fonte e, dessa forma, criar versões que melhor se adaptem a seus equipamentos.

O sistema Linux possui diversos componentes, com diversas interfaces gráficas e bibliotecas, assim como disponibiliza muitas ferramentas para a criação de aplicativos.

Além do Google, há um grupo de empresas que contribui com o desenvolvimento do Android, entre elas a Samsung, a LG, a Sony, a Motorola e a HTC.

O mercado de dispositivos Android cresceu muito e Deitel (2015, p. 3) afirma que:

> "A primeira geração de telefones Android foi lançada em outubro de 2008. Em outubro de 2013, um relatório da *Strategy Analytics* mostrou que o Android tinha 81,3% da fatia de mercado global de *smartphones*, comparados com 13,4% da Apple, 4,1% da Microsoft e 1% do Blackberry. De acordo com um relatório do IDC, no final do primeiro trimestre de 2013, o Android tinha 56,5% de participação no mercado global de *tablets*, comparados com 39,6% do iPad da Apple e 3,7% dos *tablets* Microsoft Windows."

O Android é desenvolvido em quatro camadas, cada uma contendo funcionalidades distintas, sendo a zero a de nível mais baixo, e a três a de nível mais alto.

Observe a seguir as descrições das camadas e componentes da plataforma Android que podem ser visualizadas na Figura 1.

- **Camada 0 (Kernel)** – É onde estão as funções de gestão básica da máquina, ou seja, controle da memória, gerenciamento dos componentes de *hardware* e segurança. O Kernel do GNU/Linux é a base da plataforma Android. O Android *Runtime* (ART) utiliza o kernel para realizar as funções de gerenciamento e encadeamento de memória de baixo nível;

- **Camada 1 (*Hardware Abstraction Layer* - HAL)** - Essa camada provê interfaces padronizadas que disponibilizam as capacidades de hardware do dispositivo para a estrutura da Java API de maior nível. A HAL consiste em módulos de biblioteca, que implementam uma interface para um tipo específico de componente de *hardware*, como o módulo de câmera ou *bluetooth*. Quando um *Framework* API faz uma chamada para acessar o *hardware* do dispositivo, o sistema Android carrega o módulo da biblioteca para este componente de *hardware*;

- **Camada 2 (Bibliotecas do sistema operacional)** – É responsável pelas funcionalidades básicas do dispositivo. As bibliotecas incluem um conjunto de programas em linguagem Java que podem ser desenvolvidos pelos desenvolvedores de aplicativos. Vários componentes e serviços principais do sistema Android, como ART e HAL, são implementados por código nativo que exige bibliotecas nativas programadas em C e C++. A plataforma Android fornece a Java *Framework* APIs para expor a funcionalidade de algumas dessas bibliotecas nativas aos aplicativos.

Por exemplo, é possível acessar *OpenGL* ES pela Java *OpenGL* API da estrutura do Android para adicionar a capacidade de desenhar e manipular gráficos 2D e 3D no seu aplicativo. Para dispositivos com Android versão 5.0 (API nível 21) ou mais recente, cada aplicativo executa o próprio processo com uma instância própria do Android *Runtime* (ART).

- **Camada 3 (Java API Framework) –** É totalmente acessível aos programadores e nela estão as funcionalidades de gestão das informações dos aplicativos, como notificações e demais recursos. O conjunto completo de recursos do SO Android está disponível pelas APIs programadas na linguagem Java. Essas APIs formam os blocos de programação que você precisa para criar os aplicativos Android simplificando a reutilização de componentes e serviços de sistema modulares e principais, inclusive: um sistema de visualização rico e extensivo útil para programar a interface de usuário (IU) de um aplicativo, com listas, grades, caixas de texto, botões e, até mesmo, um navegador da web incorporado; um gerenciador de recursos, fornecendo acesso a recursos sem código como strings localizados, gráficos e arquivos de *layout*; um gerenciador de notificação que permite que todos os aplicativos exibam alertas personalizados na barra de *status*; um gerenciador de atividade que gerencia o ciclo de vida dos aplicativos e fornece uma pilha de navegação inversa; provedores de conteúdo que permitem que aplicativos acessem dados de outros aplicativos, como o aplicativo Contatos, ou compartilhem os próprios dados. Os desenvolvedores têm acesso completo aos mesmos *Frameworks* APIs que os aplicativos do sistema Android usam;

- **Camada 4 (Aplicativos do sistema)** - É a camada que se comunica com o usuário. Nela estão os aplicativos como os de mensagens, e-mails, mapas e navegadores. O Android vem com um conjunto de aplicativos principais para e-mail, envio de SMS, calendários, navegador de internet, contatos etc. Os aplicativos inclusos na plataforma não têm status especial entre os aplicativos que o usuário opta por instalar. Portanto, um aplicativo terceirizado pode se tornar o navegador da Web, o aplicativo de envio de SMS ou até mesmo o teclado padrão do usuário (existem algumas exceções, como o aplicativo Configurações do sistema). Os aplicativos do sistema funcionam como aplicativos para os usuários e fornecem capacidades principais que os desenvolvedores podem acessar pelos próprios aplicativos. Por exemplo, se o seu aplicativo quiser enviar uma mensagem SMS, não é necessário programar essa funcionalidade — é possível invocar o aplicativo de SMS que já está instalado para enviar uma mensagem ao destinatário que você especificar.

Para desenvolver aplicativos para a plataforma Android, é necessária a utilização do Android SDK (*Software Development Kit*). O Android SDK constitui um conjunto de ferramentas necessárias ao desenvolvedor, incluindo um emulador, que permite testar as aplicações em um PC, antes de enviá-las ao dispositivo móvel.

Além do emulador, o Android SDK inclui as bibliotecas requeridas para o desenvolvimento, uma ferramenta de depuração de erros, exemplos de códigos, documentação e tutoriais para o sistema operacional Android.

Toda vez que uma nova versão do sistema Android é liberada, o correspondente kit Android SDK também é disponibilizado.

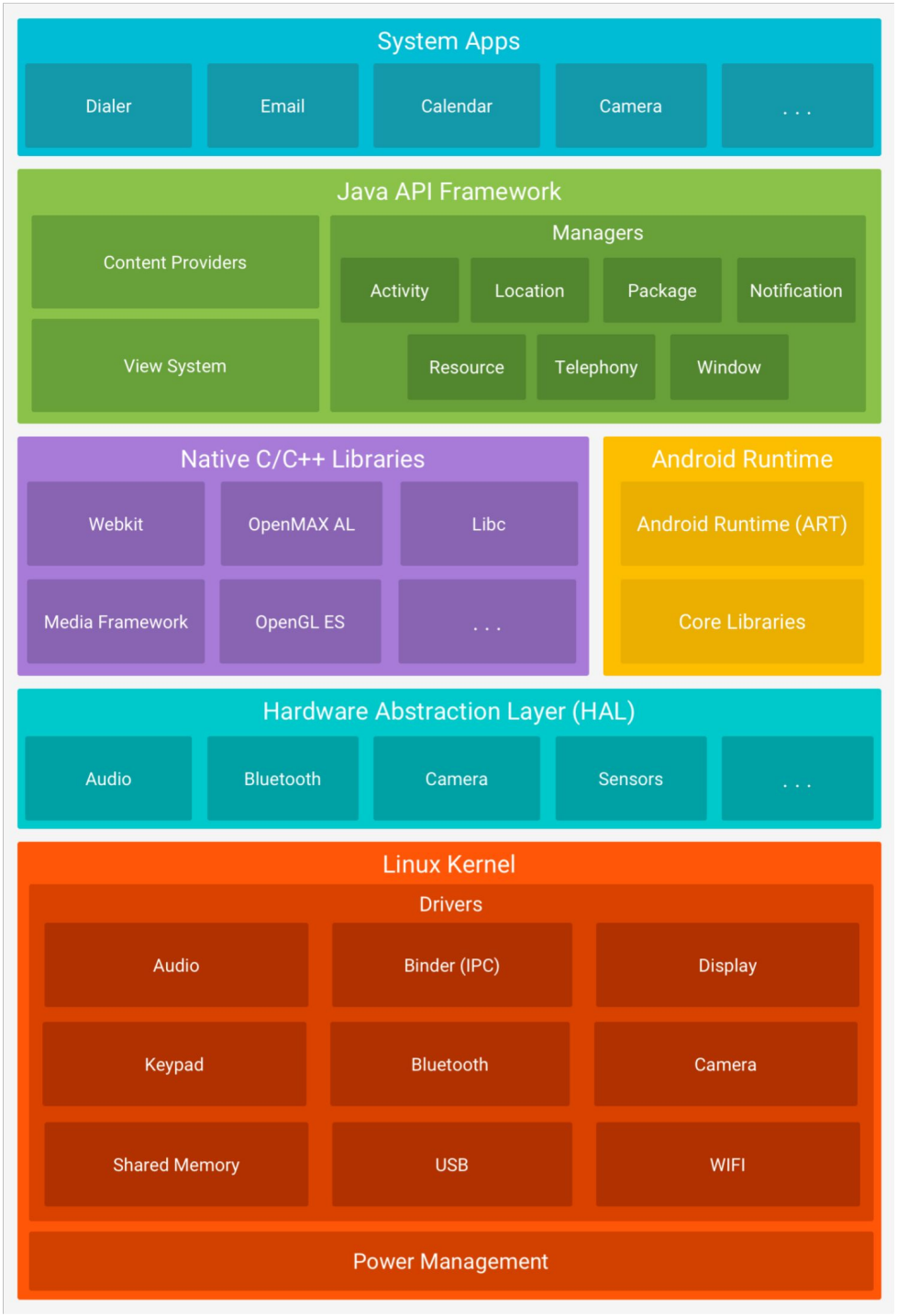


Figura 1 - Camadas e componentes da plataforma Android.
Fonte: Adaptado de *Android Developers* - Disponível em:
https://developer.android.com/guide/platform/images/android-stack_2x.png?hl=pt-br

A responsabilidade por gerenciar os processos e *threads*, além da memória, arquivos, pastas, redes, drivers dos dispositivos e energia é da camada mais baixa da arquitetura Android, o kernel Linux.

A arquitetura do sistema Android executa todos os componentes de uma aplicação em um mesmo processo e *thread*. Quando outro processo faz requisição de memória, ou, quando a memória fica sobrecarregada, por ter maior nível de importância, o processo em execução é parado e o processo que tem maior importância de acordo com o usuário é carregado. Dentro da arquitetura Android há cinco níveis de importância de processos:

1. *Foreground Process*;
2. Processo visível;
3. Processo de serviço;
4. Serviços em *Background*;
5. Processos vazios.

Para realizar a gerência de processos o Android utiliza o ***binder***, um mecanismo que faz a comunicação entre os processos. Qualquer comunicação entre os processos passa sempre pelo *binder* (uma classe do Java). O escalonamento de CPU é feito com a criação de filas com os componentes: processos interativos, processos em *batch*, processos em tempo real. Dentro da arquitetura Android o escalonador emprega o conceito de *time-sharing*, e é do tipo preemptivo.

A arquitetura do sistema Android não implementa nenhuma proteção para travamento de ***Deadlock***, partindo do princípio que eles não irão ocorrer.



***Deadlock*:** no contexto de sistemas operacionais, refere-se a uma situação em que ocorre um impasse, e dois ou mais processos ficam impedidos de continuar suas execuções - ou seja, ficam bloqueados, esperando uns pelos outros.

O gerenciamento de memória do Android toma por base o gerenciamento de memória do GNU/Linux. Todas as operações em níveis mais baixos, como I/O e gerência de memória são tratadas pelo LINUX.

Assim, da mesma forma que o Linux, para o gerenciamento de memória é utilizada a técnica de memória virtual por dois motivos básicos:

- e forma segura e eficiente em permitir o compartilhamento da memória entre diversos programas;
- remover de uma quantidade pequena e limitada da memória principal os transtornos de programação;

Assim, como a maioria dos sistemas operacionais por meio da gerência de arquivos facilita o acesso dos usuários ao seu conteúdo. O sistema de arquivos é responsável, dentro do sistema operacional, por realizar essa gerência.

De forma resumida os principais diretórios do Android são:

- ***data*** - armazena os dados das aplicações;
- ***system*** - armazena informações do sistema;
- ***system/lib*** - armazena as bibliotecas;
- ***system/bin*** **e** ***system/xbin*** - armazena os serviços;
- ***system/app*** - armazena as aplicações java.

A biblioteca denominada Bionic foi implementada pelo Android para ser utilizada como biblioteca do sistema. Essa biblioteca contém os seguintes diretórios:

- **/** – diretório raiz (Android e Linux);
- **/Cache** – armazenamento de dados para execuções rápidas (Android);
- **/Data** – Dados (Android). Esse contém dados do usuário armazenado sem uma partição separada de MTD;
- **/Default.prop** - (Android) definições de propriedade-padrão e valores restaurados a partir dos arquivos em cada reinicialização;
- **/Dev** – arquivos de dispositivos (Linux e Android).

A maioria dos diretórios do sistema operacional Android segue o padrão do sistema operacional GNU/Linux. Apenas alguns poucos diretórios foram adicionados. Vejamos:

- **/etc** – arquivos de configuração (Linux e Android);
- **/init** – inicialização (Android);
- **/lib** - Bibliotecas essenciais partilhadas (Linux e Android);
- **/lost+found** - arquivos recuperados (Linux e Android);
- **/media** - mídias removíveis (Linux e Android);
- **/proc** – kernel e arquivos de processo (Linux e Android);
- **/root** - Diretório home para o super usuário (Linux e Android);
- **/sbin** – arquivos binários de administração (Linux e Android);
- **/sdcard** – Cartão SD (Android);
- **/system** – Sistema (Android);
- **/tmp** - arquivos temporários (Linux e Android).



Como posso saber mais a respeito do desenvolvimento Android?

Na página oficial do Android, é possível encontrar as ferramentas de desenvolvimento atualizadas, incluindo o Android Studio, que reúne as ferramentas do SDK em um ambiente de trabalho para o desenvolvedor. Se estiver desenvolvendo um aplicativo que exige código C ou C++, você pode usar o Android NDK para acessar algumas dessas bibliotecas de plataforma nativa diretamente do seu código nativo.

Para obter mais informações, acesse a página oficial do Android: https://developer.Android.com

3.2 Arquitetura do Sistema Apple iOS

O sistema iOS tem seu desenvolvimento sob responsabilidade da Apple e é bastante protegido, isto é, raramente algum aplicativo se comunica diretamente com o *hardware* do dispositivo. Os aplicativos devem ser desenvolvidos para se comunicar por meio de interfaces de sistema. Essa é uma das razões pelas quais o sistema iOS é considerado um dos mais seguros, e também, imune a ataques e vírus. Assim como o Android, o iOS está organizado em quatro camadas, sendo a Core OS a de nível mais baixo, e a *Cocoa Touch* a de nível mais alto.

Camada *Core OS* – É a camada de comunicação direta com a máquina, com acesso aos componentes de *hardware*, aos acessórios conectados e à execução de cálculos fundamentais.

Camada *Core Services* – Nesta camada, estão os serviços básicos do sistema para uso das aplicações. No entanto, a maioria dessas aplicações não usa esses serviços diretamente, mas sim partes do sistema construídas “em cima” desses serviços.

Camada Média – Nesta camada estão os recursos e as tecnologias de áudio, vídeo e gráficos, todos projetados para tratar aplicativos multimídia e tornar sua programação mais fácil.

Camada *Cocoa Touch* – Nesta camada encontram-se os recursos para a elaboração das aplicações. Esse nível dispõe de tecnologias como a de notificação e a de multitarefa.

O desenvolvedor de aplicativos para iOS necessita trabalhar em um ambiente de desenvolvimento próprio, o XCode IDE (*Integrated Development Environment*), parte do iOS SDK (*Software Development Kit*).

Para realizar o desenvolvimento, são utilizadas linguagens próprias como o *Swift*, além do *Objective-C*.

Opcionalmente, o desenvolvedor pode utilizar outras plataformas, como a versão iOS do Xamarin, que também oferece os recursos necessários ao desenvolvimento e aos testes de aplicativos.



Como desenvolver para iOS?

O desenvolvimento para iOS segue ainda requisitos rigorosos da loja de aplicativos, que determinam funcionalidades que podem ou não existir, bem como diretrizes de desenho da interface.

Para obter mais informações, acesse a página oficial de desenvolvimento da Apple. Acesse:
https://developer.apple.com/

3.3 Arquitetura do Windows Phone

Ao contrário do que muitos pensam o *Windows Phone* não é uma continuação do *Windows Mobile*. A Microsoft, em completa ruptura com as antigas versões, busca agora focar diretamente no mercado consumidor, apresentando uma nova interface gráfica.

O objetivo da Microsoft é manter, em todos os dispositivos, a estrutura de menu chamada Metro, que foi lançada na versão 8 do Windows e está presente no Windows 10 com pequenas modificações.

Assim como o Android e o iOS, a estrutura interna do Sistema operacional Windows é baseada em camadas, cada uma com funções específicas.

As camadas do Windows são quatro, sendo a de mais baixo nível conhecida como camada *Hardware*, e a de mais alto nível como plataforma de aplicações.

Camada *Hardware* – Esta é a camada que se comunica com o dispositivo diretamente, seja com componentes internos da máquina, seja com acessórios externos.

Camada HAL – Com nível mais alto que a camada *Hardware*, o objetivo da camada HAL (*Hardware Abstraction Layer*) é possibilitar o acesso indireto dos aplicativos aos componentes da máquina. Nesta camada, estão as funcionalidades como a gestão da inicialização, além de recursos básicos de entrada e saída de informações.

Camada Kernel – Esta camada contém funcionalidades de gestão de memória, rede, segurança e os controladores dos dispositivos de *hardware*.

Camada *Application Plataforms* – Esta camada é onde estão as interfaces com o usuário e nela é executada a maior parte das funcionalidades das aplicações.

O desenvolvimento para a plataforma Windows é facilitado pela utilização do *Visual Studio Community* e do SDK (*Software Development Kit*) do Windows, que apoiam a elaboração dos chamados aplicativos universais para a plataforma.

Em tese, em um mesmo ambiente, é possível desenvolver aplicativos para computadores, *tablets* e *smartphones* equipados com Windows.



Como desenvolver para o Windows Phone?

É importante ter algum conhecimento em C# ou VB, que são as duas possíveis linguagens de desenvolvimento, para iniciar o desenvolvimento para Windows Phone.

Para obter mais informações, acesse a página oficial de desenvolvimento da Microsoft. Acesse:
https://developer.microsoft.com

3.4 Desenvolvimento de Aplicativos para Diferentes Sistemas Operacionais

Os sistemas iOS, Android e Windows têm características diferenciadas que, somadas aos diversos dispositivos existentes, produzem uma enormidade de variações com que os desenvolvedores têm de lidar. Os programadores que conseguem merecem um brinde!

Essa enorme diversidade de dispositivos móveis no mercado impõe grandes desafios aos programadores, que têm de desenvolver e testar seus aplicativos em diferentes equipamentos, com:

- telas de tamanhos variados;
- processadores de capacidades diversas;
- dispositivos de *hardware* específicos;
- quantidades imprevisíveis de memória disponível;

O desenvolvimento de aplicações para dispositivos *mobile* tem características diferentes do desenvolvimento tradicional, devido à variedade de equipamentos disponíveis e às limitações da capacidade de processamento, do tamanho da tela e da área de trabalho.

Além disso, esses dispositivos estão sempre sujeitos a configurações diferenciadas tanto de *hardware* quanto de *software* por parte dos fabricantes, principalmente quando estes fazem modificações no sistema operacional, o que é muito comum nos dispositivos Android.



A capacidade de processamento depende, fundamentalmente, de detalhes construtivos do dispositivo, como modelo do processador, velocidade das vias de comunicação entre os componentes de *hardware*, memória disponível para cálculos e outras especificações que estão totalmente fora do controle do desenvolvedor.

Com tantos desafios a serem vencidos, será que é possível atender todas as plataformas?

Essa é uma questão muito importante para o desenvolvedor, que deve entender e aceitar o seguinte: muito dificilmente, conseguiremos atender todas as plataformas.

Dessa forma, o desenvolvedor terá de escolher para quais sistemas operacionais irá desenvolver. Lembrando que, não basta desenvolver um aplicativo, é necessário cuidar de seu suporte e de suas atualizações, e isso pode ser muito trabalhoso se houver a intenção de atingir todos os dispositivos do mercado.

Mesmo grandes empresas, que têm centenas de desenvolvedores trabalhando para elas, fazem escolhas. São raros os aplicativos disponíveis para todas as plataformas, mesmo considerando as três dominantes: Android, iOS e Windows.

Os recursos de desenvolvimento utilizados nas aplicações móveis devem assegurar certo nível de suporte a essa diversidade. Dessa forma, devem garantir a execução sem erros e o controle de comportamentos indesejáveis do aplicativo em diferentes aparelhos.

Um dos focos do desenvolvedor deve ser a usabilidade, que é a facilidade, ou simplicidade, com que ocorre a interação de um aplicativo ou *website* com o usuário.

No contexto do desenvolvimento de aplicações móveis, a usabilidade envolve desde a apresentação gráfica do aplicativo, suas respostas aos comandos do usuário, seu nível de interatividade e em que grau sua utilização pode ser considerada intuitiva.

Telas simples e com poucas opções facilitam o entendimento e a utilização dos aplicativos.

Já vimos algumas das características dos três sistemas operacionais mais utilizados no mundo dos dispositivos móveis, certo?

Falamos também, um pouco a respeito dos ambientes de desenvolvimento. Mas, agora, vamos detalhar um pouco mais as funcionalidades desses ambientes, também chamados de plataformas.

Para facilitar a vida do desenvolvedor, os próprios fornecedores de sistemas operacionais disponibilizam ambientes integrados de desenvolvimento ou IDE (*Integrated Development Environment*).





Onde obter informações sobre os ambientes de desenvolvimento para dispositivos móveis?

Os ambientes oficiais do Android, iOS e Windows podem ser obtidos, gratuitamente, nos seguintes sites:

https://developer.Android.com

https://developer.apple.com/

https://developer.microsoft.com/

Funcionalidades dos Ambientes de Desenvolvimento

Os ambientes reúnem ferramentas para apoiar o desenvolvedor em seu trabalho.

Vejamos a seguir as ferramentas típicas em desenvolvimento:

a) Editor

Programa para se trabalhar os códigos fonte escritos nas linguagens de programação aceitas pelo sistema. É possível editar os códigos fonte em editores padrões, como o editor de textos do Windows. No entanto, os editores integrados aos ambientes de desenvolvimento, usualmente, dispõem de mais recursos para organização do programa fonte e sua documentação, facilitando o trabalho do desenvolvedor;

b) Compilador

Transforma o código fonte do programa escrito pelo desenvolvedor em um programa em linguagem de máquina, isto é, em um formato que será entendido pelo dispositivo;

c) Gerador de Testes

Possibilita realizar testes automáticos de entrada e saída de dados no aplicativo, simulando a operação realizada pelos usuários. É um importante recurso quando se deseja testar diversas possibilidades de inserção de dados, buscando eventuais erros ou grandes volumes de acesso;

d) Depurador (ou debugger, em inglês)

Ajuda a encontrar erros e a aprimorar o programa escrito pelo desenvolvedor;

e) Emulador

Ferramenta que permite simular a execução do aplicativo em dispositivos móveis dentro do ambiente de desenvolvimento, normalmente em um PC ou Mac. Por meio dos emuladores, é possível ver e testar o aplicativo como se ele estivesse em um dispositivo móvel, sendo possível realizar ajustes e correções sem a necessidade de publicá-lo para carga nos dispositivos;

f) Ferramentas para distribuição

Tem por objetivo formatar o "pacote" de informações, documentos e executáveis, que será direcionado à loja do fornecedor (*AppStore* da Apple, Google Play ou Windows *Store*) para a aprovação e publicação do aplicativo.

Além dos ambientes de desenvolvimento oficiais, existem outros no mercado, que podem ser utilizados pelos desenvolvedores de acordo com a disponibilidade e facilidade de uso.

Na verdade, a escolha do ambiente de desenvolvimento é, muitas vezes, uma questão de gosto pessoal ou de familiaridade do desenvolvedor, exceto quando o desenvolvedor atua em uma empresa ou um projeto em que determinado ambiente é mandatório, para fins de padronização e documentação.

O desenvolvedor tem disponíveis diversos ambientes de desenvolvimento para cada plataforma.

Para o Android, além do IDE oficial Android *Studio*, podem ser usados os seguintes ambientes:

- Eclipse (https://eclipse.org/ide/)
- NetBeans (http://plugins.netbeans.org/plugin)
- Xamarin (https://www.xamarin.com/)

A linguagem de programação para desenvolvimento de aplicativos para Android é o Java, mas alguns códigos em C e C++ podem ser utilizados.

A plataforma Android dispõe de muitos recursos. Dessa forma, desenvolvedores experientes podem utilizar mais de um ambiente, aproveitando os recursos que consideram melhores de um ou de outro.

O Apple iOS dispõe do ambiente oficial *Xcode*, mas o desenvolvedor também pode optar por utilizar um ambiente de desenvolvimento alternativo, como um dos seguintes:

- Jetbrains (https://www.jetbrains.com/objc/)
- Xamarin (https://www.xamarin.com/studio)

A linguagem de programação para desenvolvimento de aplicativos para o iOS é o *Swift*, uma linguagem de programação criada pela Apple para as plataformas Mac e iOS.

A interface gráfica dos aplicativos é criada pelo *Xcode*. No entanto, o desenvolvedor pode utilizar outras linguagens, como o *Objective-C*, por exemplo.

O Windows dispõe do ambiente oficial *Visual Studio*, mas também podem ser usados os seguintes ambientes, por exemplo:

- Eclipse (https://eclipse.org/ide/)
- Xamarin (https://www.xamarin.com/studio

As linguagens de programação para o desenvolvimento de aplicativos para Windows são o C, C#, C++ e *Visual Basic*.

A estratégia de desenvolvimento da Microsoft conta com a utilização de um único conjunto de ferramentas para os programadores de aplicativos para PC, *tablet* e *smartphone*.

Bora rever!

Essa unidade apresentou a arquitetura dos principais Sistemas Operacionais para computadores e dispositivos móveis. Foi possível compreender que os sistemas operacionais móveis normalmente combinam recursos úteis para uso móvel ou portátil (tela sensível ao toque, celular, Bluetooth, Wi-Fi, GPS de navegação móvel, câmera fotográfica, câmera de vídeo, reconhecimento de voz e leitor de música) com as características dos sistemas operacionais de um computador pessoal.

O funcionamento das aplicações é influenciado pela arquitetura do sistema operacional a qual, também, interfere no processo de desenvolvimento dessas aplicações.

O sistema Android foi desenvolvido com o conceito de plataforma aberta, baseada no sistema operacional GNU/Linux, o que confere aos fabricantes de dispositivos a liberdade de alterar seu código-fonte e, dessa forma, criar versões que melhor se adaptem a seus equipamentos. O sistema Linux possui diversos componentes, com diversas interfaces gráficas e bibliotecas, assim como disponibiliza muitas ferramentas para a criação de aplicativos.

O sistema iOS tem seu desenvolvimento sob responsabilidade da Apple e é bastante protegido, isto é, raramente algum aplicativo se comunica diretamente com o hardware do dispositivo. Os aplicativos devem ser desenvolvidos para se comunicar por meio de interfaces de sistema. Essa é uma das razões pelas quais o sistema iOS é considerado um dos mais seguros, e, também, imune a ataques e vírus.

Bora rever!

Assim como o Android e o iOS, a estrutura interna do Sistema operacional Windows é baseada em camadas, cada uma com funções específicas. As camadas do Windows são quatro, sendo a de mais baixo nível conhecida como camada Hardware, e a de mais alto nível como plataforma de aplicações.

Os sistemas iOS, Android e Windows têm características diferenciadas que, somadas aos diversos dispositivos existentes, produzem uma enormidade de variações com que os desenvolvedores têm de lidar.

O desenvolvimento de aplicações para dispositivos mobile tem características diferentes do desenvolvimento tradicional, devido à variedade de equipamentos disponíveis e às limitações da capacidade de processamento, do tamanho da tela e da área de trabalho.

Na próxima unidade, veremos conceitos de virtualização, os tipos e como construir máquinas virtuais, além de aprofundar um pouco mais nos conceitos da máquina virtual Android.



Glossário

Touch: é uma tela sensível ao toque é um tipo de ecrã sensível à pressão, dispensando, assim, a necessidade de outro periférico de entrada de dados, como o teclado;

***Runtime*:** é o período em que um programa de computador permanece em execução;

API: Interface de Programação de Aplicação (português brasileiro) é um conjunto de rotinas e padrões estabelecidos por um *software* para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação do *software*, mas apenas usar seus serviços;

***Framework*:** é uma abstração que une códigos comuns entre vários projetos de *software* provendo uma funcionalidade genérica;

***OpenGL*:** o *OpenGL* é uma API livre utilizada na computação gráfica, para desenvolvimento de aplicativos gráficos, ambientes 3D, jogos, entre outros;

***Layout*:** é uma palavra inglesa, muitas vezes usada na forma portuguesa "leiaute", que significa plano, arranjo, esquema, design, projeto;

***Software Development Kit*:** é um conjunto de ferramentas de desenvolvimento de *software* que permite a criação de aplicativo para certo pacote de *software*, *hardware* ou sistema operacional;

***Ahead-of-time*:** adiantado, antecipado;

***Just-in-time*:** significa “na hora certa” ou "momento certo";

***Bytecodes*:** um formato de código intermediário entre o código fonte, o texto que o programador consegue manipular, e o código de máquina, que o computador consegue executar.

Unidade 4 VIRTUALIZAÇÃO

Virtualizar significa: simular, mascarar ambientes isolados, capazes de rodar diferentes sistemas operacionais (SO) dentro de uma mesma máquina. Este processo aproveita ao máximo a capacidade do *hardware*, que muitas vezes fica ociosa em determinados períodos do dia, da semana ou do mês.

A possibilidade de fornecer ambientes de execução independentes a diferentes usuários em um mesmo equipamento físico, concomitantemente, permite um aproveitamento maior. A virtualização ajuda, também, a resolver o problema da restrição ao uso do *hardware* para a utilização por determinados *softwares*, pois ela diminui o poder dos sistemas operacionais. Alguns *softwares* só rodam sobre o sistema operacional para o qual foram desenvolvidos.

Assim, a virtualização permite que diferentes sistemas rodem em um mesmo equipamento (máquina), ampliando, desta forma, a quantidade de *softwares* que podem ser empregados em um mesmo *hardware*.

Em servidores, essa abordagem é bastante empregada, com a vantagem adicional de fornecer uma interface (camada de abstração) dos verdadeiros recursos de uma máquina, já que provê um *hardware* virtual para cada sistema. No ambiente de servidores ela é uma excelente ferramenta para migração de sistemas.

Vamos refletir um pouco sobre o surgimento da virtualização. Considerando que o *hardware*, o sistema operacional e as aplicações foram desenvolvidos ao longo do tempo por empresas diferentes, e em momentos históricos diferentes.

É natural que com o passar do tempo os caminhos do desenvolvimento de tecnologias computacionais acabassem por se tornar heterogêneos (distintos) e por consequência, surgissem sistemas incompatíveis. Esse processo levou ao surgimento de plataformas operacionais diferenciadas, porém, incompatíveis entre si. Com o advento da virtualização surgiu a possibilidade da criação de uma máquina virtual (VM), uma camada de compatibilidade sobre o sistema operacional hospedeiro e hardware real, que poderia simular o ambiente de outra máquina real, onde, então, seria possível instalar outro sistema operacional convidado.

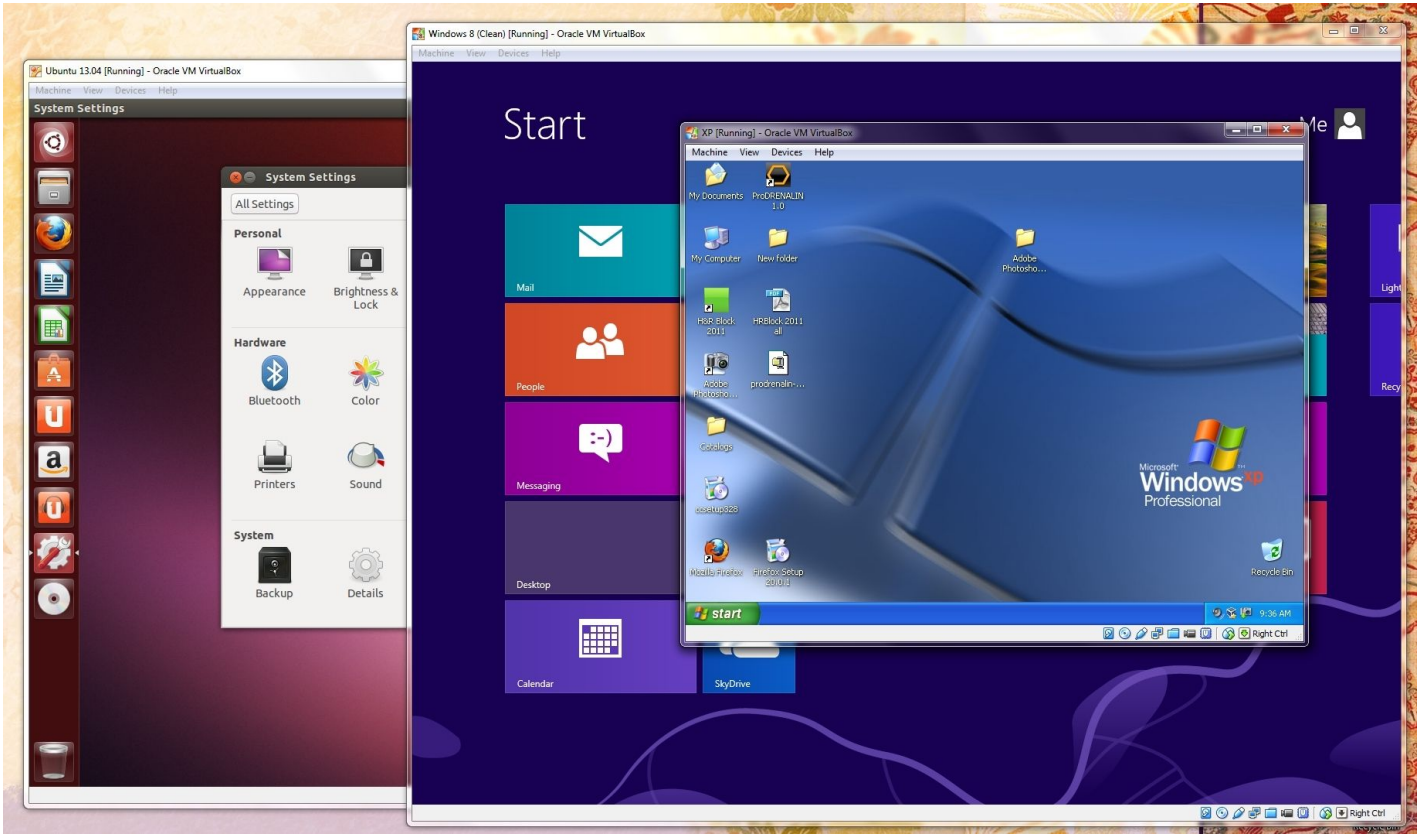


Figura 1 - Máquinas virtuais - Virtualbox.
Fonte: https://www.servti.com/blog/wp-content/uploads/2016/09/virtualbox.jpg

A Figura 1 exemplifica a execução de 3 (três) máquinas virtuais sobre um *hardware* x86 com Windows. O *Virtualbox* está executando três máquinas virtuais com *hardwares* independentes do *hardware* real, criados pela camada de virtualização, o que permite executar diversos sistemas operacionais diferentes.

Desta forma, com a virtualização, uma técnica que possibilita a criação de diversos ambientes virtuais (máquinas virtuais), pode instalar e executar diversos sistemas operacionais (SO) em uma mesma máquina real, permitindo executar um SO diferente dentro de cada máquina virtual criada. Para Deitel (2010, p.15), "uma máquina virtual (VM) é uma abstração em *software* de um computador executado frequentemente como uma aplicação de usuário sobre o sistema operacional nativo".

4.1 Fundamentos da virtualização

Uma máquina real (física) é composta por diversos dispositivos físicos que disponibilizam operações para as camadas de abstração acima. O núcleo desse sistema é o processador e o *chipset* da placa-mãe traz consigo todos os recursos como áudio, vídeo, memória, e portas de comunicação. Cada *hardware* pode ter as suas peculiaridades, mas de forma geral é assim que o *hardware* está organizado.

O emulador é um *software* que tem a função de enganar uma aplicação. A Figura 2 demonstra a atuação do emular. Imagine uma máquina real que executa certa aplicação (X) em um determinado sistema operacional (A). Se desejarmos executar essa aplicação em outra máquina, a qual não possui os mesmos recursos, com outro *hardware* e sistema operacional diferente (B). Desejamos obter o mesmo resultado da execução da aplicação X no sistema operacional A.

A Figura 2 demonstra a técnica para rodar uma aplicação projetada para um sistema operacional em outro incompatível.

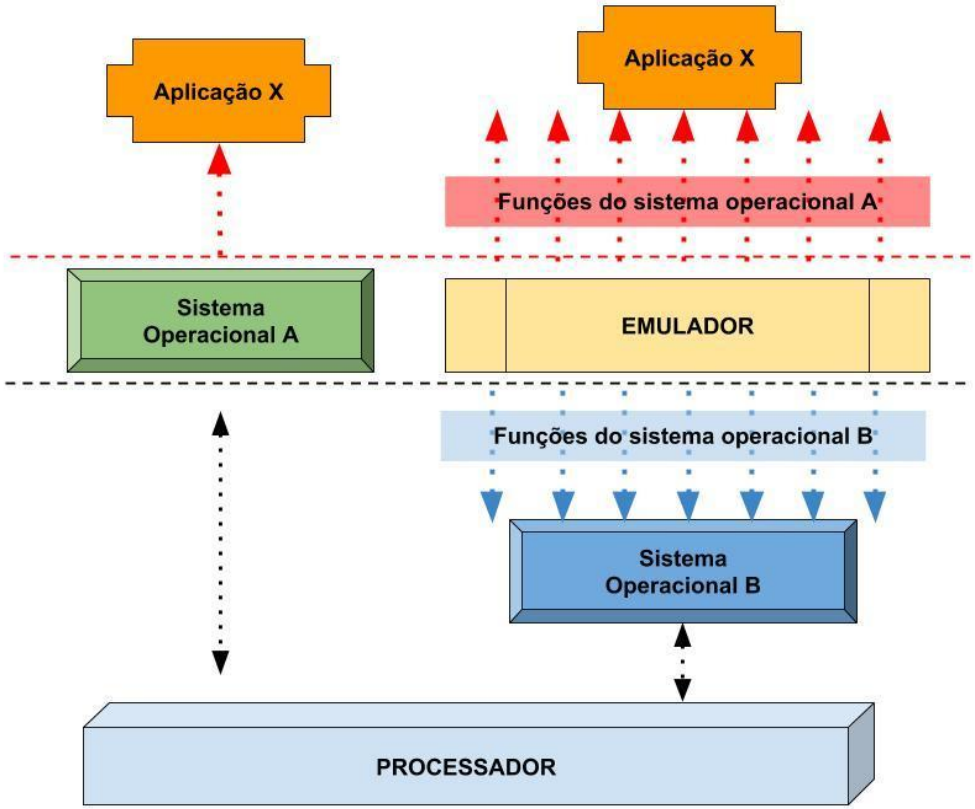
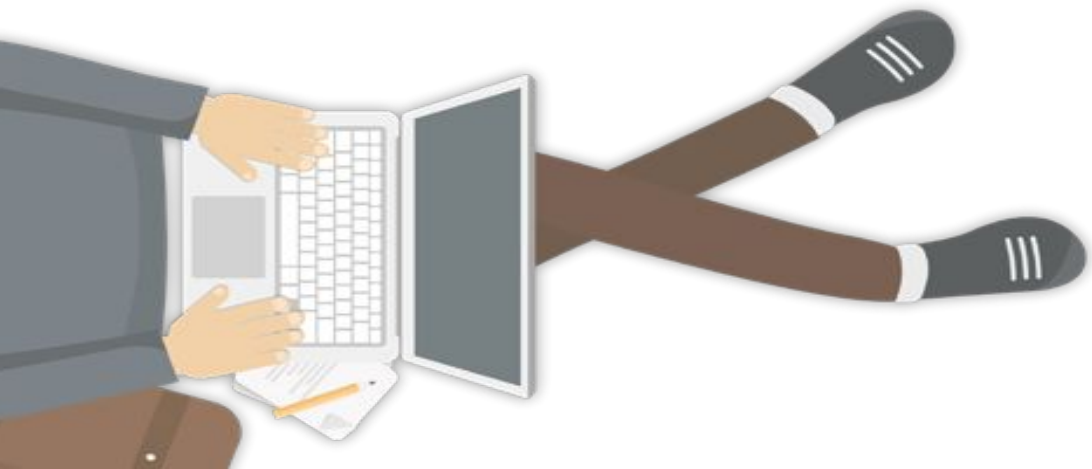


Figura 2 - Sistema de emulação.
Fonte: elaborado pelo autor.

Utilizando *software*, ou seja, uma máquina virtual (VM), podemos criar a imitação de uma máquina real. A simulação (imitação) criada por ela permite rodar sistemas operacionais que terão a ilusão de estar rodando na máquina real que a máquina virtual está simulando. Esse tipo de imitação pode ser realizada (criadas) em nível de sistema operacional, ou, até mesmo de aplicação. Vejamos:

a) **máquina virtual criada a nível de aplicação:** a máquina virtual é executada sobre o sistema operacional, denominado anfitrião. O nível acima da VM irá acreditar que abaixo dele há uma máquina real (imitação criada pela VM), e, assim, pode-se rodar outro sistema operacional diferente do que está abaixo da imitação. Esse é o processo empregado para executar uma aplicação que foi projetada para o sistema A em um sistema B (incompatível);



b) **máquina virtual criada a nível de sistema operacional:** neste caso, emprega-se um Monitor de Máquina Virtual (MMV), em inglês *Virtual Machine Monitor* (VMM). A função do MMV é controlar o *hardware* e permitir a criação de várias máquinas virtuais. O MMV permite assim executar vários sistemas operacionais, um em cada imitação criada. Cada sistema operacional novo, sendo executado, imaginará estar rodando sobre um *hardware* real, quando na verdade, está sobre uma interface de abstração criada pela VM para simular o *hardware* real, compatível com esse novo SO. A vantagem desse processo é permitir executar diferentes SO sobre um mesmo *hardware* (mesma máquina real), cada um com suas aplicações, de forma transparente e independente. O MMV, também, faz a gerência do uso de dispositivos físicos de entrada e saída, recursos de memória da máquina real, permitindo que os sistemas operacionais que estão sendo executados nas VMs acessem esses recursos de forma transparente e sem conflitos. É claro que o *hardware* real não será duplicado, mas sim multiplexado (compartilhado).

É desta maneira que os recursos da máquina são compartilhados entre os usuários. Eles permanecerão acreditando que estão executando suas aplicações diretamente na máquina real, quando na verdade estão compartilhando (dividindo) os recursos disponíveis.

Na figura a seguir temos a comparação de um sistema não virtualizado com os dois tipos possíveis de virtualizações.

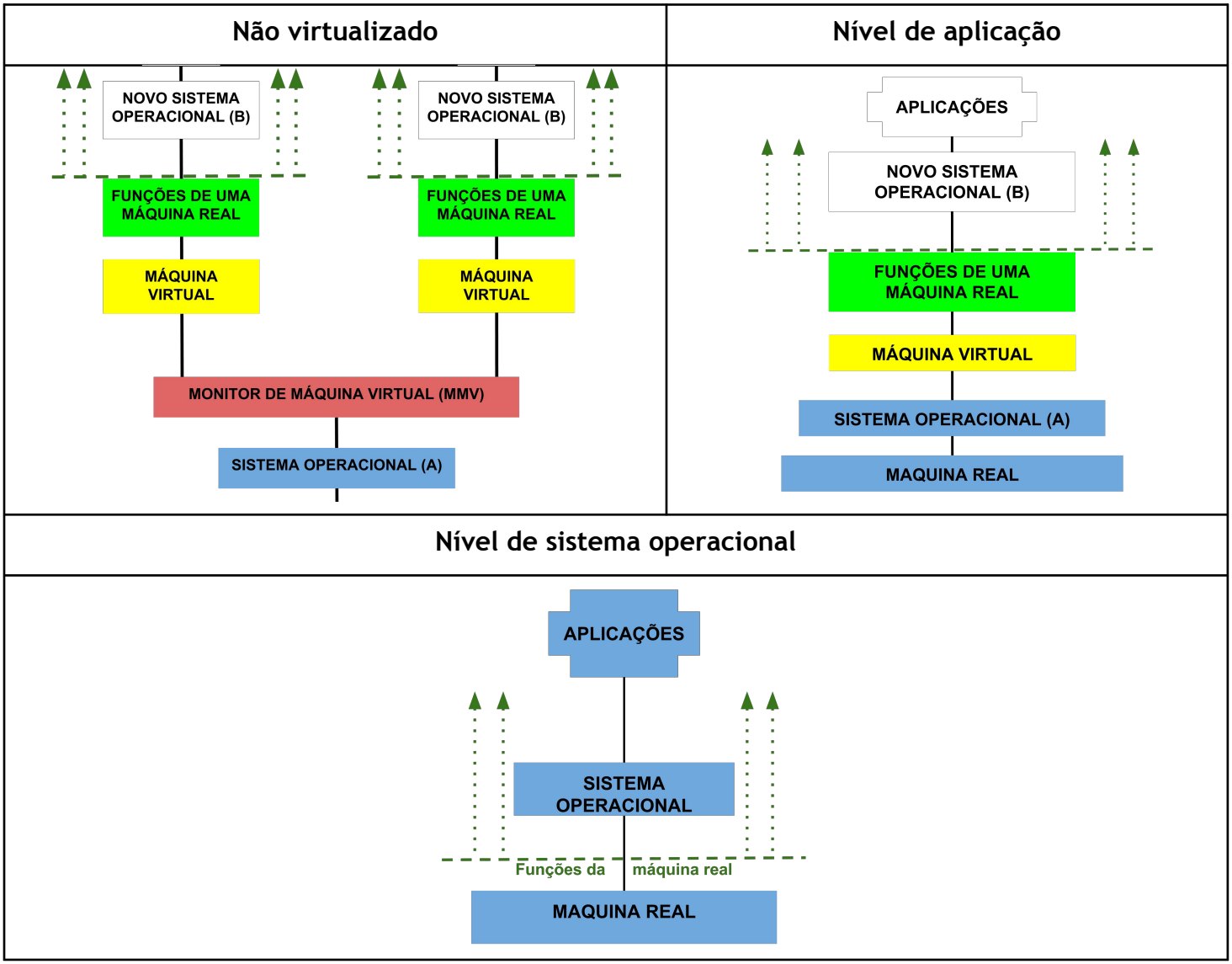


Figura 3 - Exemplo de sistema não virtualizado com possíveis virtualizações.
Fonte: elaborado pelo autor.

4.2 A construção de máquinas virtuais

A virtualização consiste na possibilidade de gerar abstrações de recursos (reais ou virtuais) empregando *softwares*, de forma que eles pareçam ser diferentes do que realmente são. Esse conceito pode ser estendido e empregado na execução de virtualização de três formas diferentes: virtualização de sistema operacional, virtualização de *hardware* e virtualização de linguagem de programação.

Vejamos:

a) **virtualização de sistema operacional:** é implantada sobre um sistema operacional existente e consiste em criar a simulação de um novo sistema operacional. Mesmo não havendo grandes ganhos significativos, permite resolver a necessidade de executar aplicações em um sistema operacional incompatível. Temos como exemplo, o *FreeBSD Jail* e o *User-mode Linux*;

b) **virtualização de *hardware*:** é utilizada para imitar um *hardware* real. A VM é executada sobre um SO e permite a execução de outros SO sobre ela. O sistema que está logo abaixo da VM pode ser um MMV ou um SO direto na máquina real. O *VMware* e o *Xen* na plataforma x86 são exemplos desse tipo de virtualização;

c) **virtualização de linguagens de programação:** é empregada para fazer o computador real se comportar de forma diferente, ou seja, com novas instruções. A VM irá executar o programa de acordo com o comportamento definido pelo usuário para a virtualização, e ficará encarregada de traduzir as ações em novas ações que o sistema operacional abaixo compreenda. Como exemplo desse tipo de virtualização, temos o Java e o *Smalltalk*.

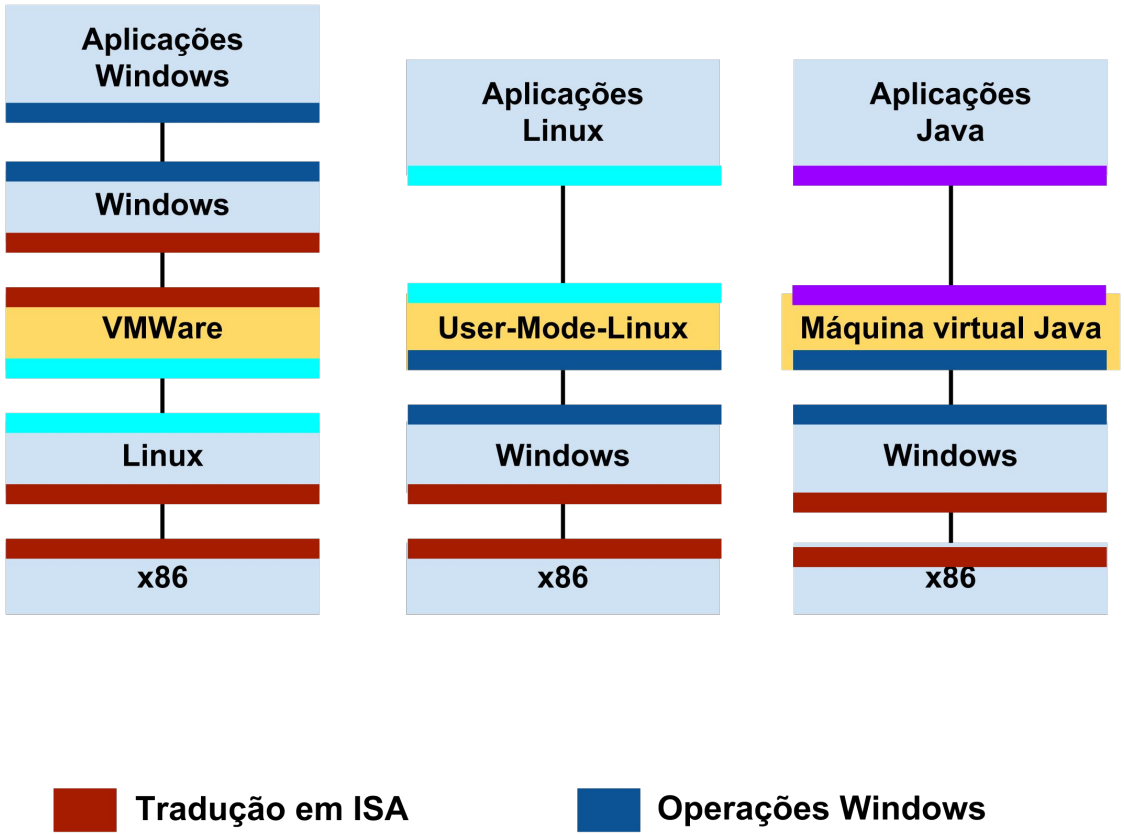


Figura 4 - Sistema de emulação.
Fonte: UFRJ - GTA - Virtualização. Disponível em:
https://www.gta.ufrj.br/grad/09_1/versao-final/virtualizacao/figuras/como_e_empregado.jpg

A virtualização pode ser empregada para facilitar o desenvolvimento de *softwares* e de sistemas operacionais. Podemos utilizar VMs para testar o SO em desenvolvimento, sem causar danos ou impactos. De igual forma, os *softwares* em desenvolvimento podem ser testados em sistemas virtuais.

A grande vantagem do emprego de máquinas virtuais, nesse contexto de desenvolvimento, é que podemos simular ambientes e testar sistemas operacionais e *softwares* que ainda estão em desenvolvimento, antes de serem comercializados. Outro aspecto é permitir a comparação de execução de um determinado *software* ou aplicação, em diversos sistemas operacionais, empregando a mesma máquina real.

É possível, ainda, executar aplicações diversas na mesma máquina, pois não é raro que alguém necessite rodar aplicações voltadas para sistemas operacionais diferentes. Neste caso, a virtualização pode ser empregada para permitir rodar vários sistemas diferentes na mesma máquina, e, assim, várias tarefas projetadas para sistemas incompatíveis podem ser executadas no mesmo ambiente de *hardware*.

Outra aplicação a considerar é empregar VMs para criar situações e ambientes simulados (criar a ilusão de recursos reais) que permitam avaliar o funcionamento de aplicações ou sistemas operacionais. Neste caso, a VM é empregada para simular, testar e avaliar resultados de forma simulada, atendendo a situações reais.

É possível empregar virtualização para manter sistemas legados (com funções muito específicas e de difícil adaptação) que necessitam versões antigas ou específicas de aplicações ou sistemas operacionais.

A virtualização pode ser empregada, também, para a consolidação de servidores, uma vez que a rápida evolução do poder computacional não foi acompanhada pela demanda por capacidade. Esse processo permitiu recursos ociosos, como servidores utilizando uma pequena parte do seu poder de processamento. Desta forma, a virtualização atua para permitir que um servidor empregue toda a sua capacidade hospedando diversos sistemas operacionais e aplicações ao mesmo tempo. Então, usar o máximo da capacidade da máquina, significa na prática, diminuir custos com *hardware*.

Máquinas virtuais (virtualização) possibilitam o fornecimento de serviços dedicados a clientes específicos, traduzindo-se em processos mais seguros com confiabilidade e disponibilidade.

Na área de segurança podemos empregar máquinas virtuais para atuar como *honeypots* contra ataques de invasores na Internet ou Intranet. É possível manter de propósito várias instâncias de VMs para que os invasores possam atacá-las. Esse tipo de máquina de "isca" é chamado de *honeypots* (pote de mel) e tem a função de servir como sistema de monitoramento dos possíveis ataques, ajudando a desenvolver meios de prevenção contra eles.



Cinco motivos pelos quais você deve usar o recurso da virtualização de sistemas

Virtualizar sistemas operacionais é um recurso bastante utilizado por Administradores de Redes em *DataCenters*. Um motivo óbvio para tal uso é, sem dúvidas, ter a possibilidade de gerenciar diversos sistemas em apenas um computador hospedeiro. Mas existem cinco motivos pelos quais você também deveria usar esse recurso em sua máquina doméstica! Descubra por que!

Vamos ler um pouco mais sobre o tema?

Visite:
https://www.linuxdescomplicado.com.br/2011/06/5-motivos-pelos-quais-voce-deve-usar-o-2.html

4.3 Tipos de máquinas virtuais

A virtualização pode ser classificada quanto à arquitetura do processo. Temos três tipos possíveis. Vejamos:

a) Arquitetura Tipo I - Neste formato de arquitetura o Monitor de Máquina Virtual (MMV ou VMM) é implementado diretamente sobre o *hardware* hospedeiro. Assim, o monitor pode controlar todas as operações de acesso requisitadas pelos sistemas convidados, simulando máquinas físicas com propriedades distintas e trabalhando de forma isolada. Neste formato podemos ter diferentes sistemas operacionais virtuais operando sobre o mesmo *hardware*. São exemplos desse tipo: VMM's XEN e VMWARE ESX SERVER.



Figura 5 - Arquitetura Tipo I.
Fonte: Elaborado pelo autor.

b) Arquitetura Tipo II - caracteriza-se pela implementação do Monitor de Máquina Virtual (MMV ou VMM) sobre o sistema operacional instalado no *hardware* anfitrião, operando como um processo desse sistema operacional. As operações que seriam controladas pelo sistema operacional do hospedeiro são simuladas pelo monitor para as máquinas virtuais. Temos como exemplos desse tipo: *VMWARE SERVER* e *VIRTUALBOX*.

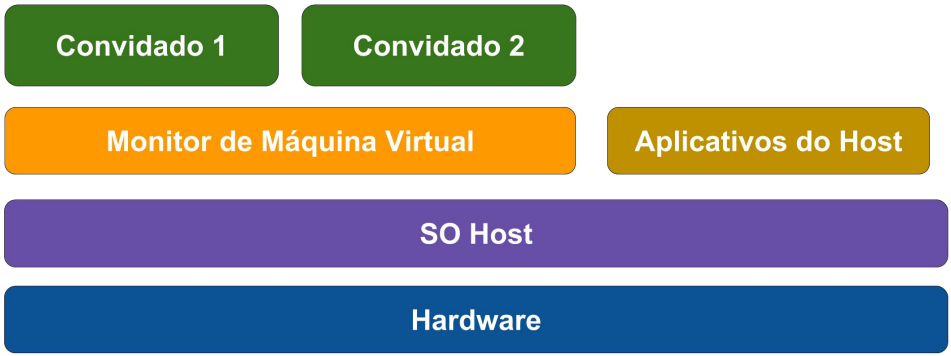


Figura 6 - Arquitetura Tipo II.
Fonte: Elaborado pelo autor.

c) Arquitetura Híbrida - este tipo engloba qualidades dos dois tipos anteriores. É possível agregar características do tipo I ao tipo II e vice-versa. O formato híbrido busca otimizar os tipos I e II.

Otimização para MMV de Tipo I: O sistema convidado acessa diretamente o *hardware*, através de modificações no sistema convidado e no monitor. Essa otimização é utilizada para algumas funcionalidades do Xen.

Otimização para MMV de Tipo II: o sistema convidado acessa diretamente o SO real da máquina, sobre o qual funciona o monitor. Dessa forma, alguns sistemas virtuais não precisam ser inteiramente providos pelo monitor. No *VMware*, o sistema de arquivos do SO real é utilizado pelo sistema convidado, poupando o monitor de gerar um sistema similar na aplicação virtual.

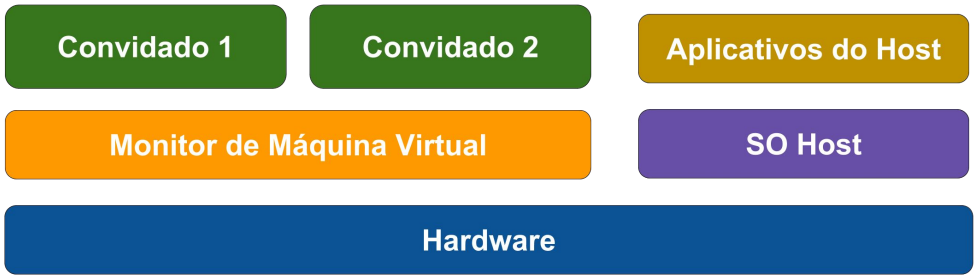


Figura 7 - Arquitetura Híbrida.
Fonte: Elaborado pelo autor.

4.4 Técnicas de virtualização

a) **Virtualização completa (total):** como o nome sugere, uma estrutura completa de *hardware* é virtualizada. O *hardware* hospedeiro é completamente abstraído e todas as características de um equipamento virtual são emuladas, ou seja, todas as instruções solicitadas pelo sistema convidado são interpretadas no Monitor de Máquina Virtual. O Sistema convidado não precisa sofrer qualquer tipo de alteração.

O sistema hospedado ignora a existência da máquina virtual e opera como se funcionasse diretamente sobre o sistema operacional para o qual foi projetado para funcionar. Há um grande nível de compatibilidade, porém há perda de velocidade.

A virtualização completa é mais flexível em termos de SO convidados, uma vez que este não precisa ser modificado para implementação dessa técnica. Todas as instruções são interpretadas pelo monitor de máquina virtual. Em compensação, essa interpretação de cada instrução provoca perda de desempenho de processamento, uma vez que o monitor de máquina virtual se utiliza de dispositivos de virtualização que atendem a uma gama de aplicativos e, por isso, possuem uma baixa especialização. Assim, não é possível ter o máximo desempenho desse aplicativo.

b) Paravirtualização: nessa técnica, a máquina virtual não é idêntica ao equipamento físico original, para que o sistema hospedado possa enviar as instruções mais simples diretamente para o *hardware*, restando apenas as instruções de nível mais alto para serem interpretadas pelo MMV. Há perda de compatibilidade, porém com ganho de velocidade.

Esse procedimento requer que o sistema operacional convidado seja modificado para interagir com o MMV e selecionar quais instruções devem ser interpretadas nele ou diretamente no hardware hospedeiro.

A paravirtualização é menos flexível, pois carece de modificações no sistema operacional convidado, para que este possa trabalhar perfeitamente. Porém, o fato de o sistema operacional convidado saber que opera sobre uma máquina virtual e, com isso, mandarem as instruções mais simples diretamente para o *hardware* diminui a sobrecarga no MMV e permite uma maior especialização dos dispositivos de virtualização. Dessa forma, os aplicativos operam mais próximos de sua capacidade máxima, melhorando seu desempenho em comparação à virtualização completa. Além disso, na paravirtualização, a complexidade das máquinas virtuais a serem desenvolvidas diminui consideravelmente.

c) Recompilação Dinâmica: na recompilação dinâmica, as instruções são traduzidas durante a execução do programa da seguinte forma: as instruções do programa são identificadas em forma de sequência de bits. Em seguida, as sequências são agrupadas em instruções mais próximas do sistema operacional hospedeiro. Por último, essas instruções são reagrupadas em um código de mais alto nível, que, por sua vez, é compilado na linguagem nativa do sistema hospedeiro. A recompilação dinâmica tem como principal vantagem a melhor adequação do código gerado ao ambiente de virtualização, que, com a compilação durante a execução, pode refletir melhor o ambiente original do aplicativo. Isso acontece porque durante a execução, há novas informações disponíveis, às quais um compilador estático não teria acesso. Dessa forma, o código gerado se torna mais eficiente. Em contrapartida, essa técnica exige maior capacidade de processamento, visto que a recompilação acontece em tempo real de execução do programa.

4.5 Ambientes de máquinas virtuais

Conheceremos alguns dos sistemas disponíveis para trabalhar com máquinas virtuais em sistemas operacionais convencionais (*desktops* e servidores). Apresentaremos os sistemas *VMWare*, *Virtualbox*, FreeBSD Jails, Xen, *User-Mode Linux*, QEMU, Valgrind e JVM.

Eles são exemplos de máquinas virtuais de aplicação e de sistema, empregando virtualização total ou paravirtualização, além de abordagens híbridas. A escolha foi feita considerando que eles estão entre os mais representativos de suas respectivas classes.

Sistema	Descrição
VMWare	(Virtualização total) - Atualmente, o *VMware* é a máquina virtual para a plataforma x86 de uso mais difundido, provendo uma implementação completa da interface x86 ao sistema convidado. Embora essa interface seja extremamente genérica para o sistema convidado, acaba conduzindo a um hipervisor mais complexo. Como podem existir vários sistemas operacionais em execução sobre mesmo *hardware*, o hipervisor tem que emular certas instruções para representar corretamente um processador virtual em cada máquina virtual, fazendo uso intensivo dos mecanismos de tradução dinâmica. **Site:** https://www.vmware.com/br.html
Virtualbox	(Virtualização total) - O *VirtualBox* é uma ferramenta de virtualização originalmente desenvolvida pela a empresa alemã Innotek. Posteriormente, foi adquirida pela Sun, que por sua vez em 2010 tornou-se aquisição da *Oracle Corporation*, recebendo assim, uma nova nomenclatura: *Oracle VM VirtualBox*. **Site:** https://www.virtualbox.org/
FreeBSD Jails	(Virtualização no nível do Sistema Operacional) - O sistema operacional *FreeBSD* oferece um mecanismo de confinamento de processos denominado *Jails*, criado para aumentar a segurança de serviços de rede. Esse mecanismo consiste em criar domínios de execução distintos (denominados *jails* ou celas). Cada cela contém um subconjunto de processos e recursos (arquivos, conexões de rede) que pode ser gerenciado de forma autônoma, como se fosse um sistema separado. Cada domínio é criado a partir de um diretório previamente preparado no sistema de arquivos. Um processo que executa a chamada de sistema *jail* cria uma nova cela e é colocado dentro dela, de onde não pode mais sair, nem seus filhos. **Site:** https://www.freebsd.org/doc/handbook/jails.html

Xen	(Paravirtualização) - O ambiente *Xen* é um hipervisor nativo para a plataforma x86 que implementa a paravirtualização. Ele permite executar sistemas operacionais como Linux e Windows especialmente modificados para executar sobre o hipervisor. Versões mais recentes do sistema Xen utilizam o suporte de virtualização disponível nos processadores atuais, o que torna possível a execução de sistemas operacionais convidados sem modificações, embora com um desempenho ligeiramente menor que no caso de sistemas paravirtualizados. **Site:** https://www.xenproject.org
User-Mode Linux	(Virtualização Total) - O *User-Mode Linux* foi proposto como uma alternativa de uso de máquinas virtuais no ambiente *Linux*. O núcleo do *Linux* foi portado de forma a poder executar sobre si mesmo, como um processo do próprio Linux. O resultado é um *user space* separado e isolado na forma de uma máquina virtual, que utiliza dispositivos de *hardware* virtualizados a partir dos serviços providos pelo sistema hospedeiro. Essa máquina virtual é capaz de executar todos os serviços e aplicações disponíveis para o sistema hospedeiro. Além disso, o custo de processamento e de memória das máquinas virtuais *user-Mode Linux* é geralmente menor que aquele imposto por outros *hipervisores* mais complexos. **Site:** http://user-mode-linux.sourceforge.net/
QEMU	(Virtualização completa) - O QEMU é um hipervisor que não requer alterações ou otimizações no sistema hospedeiro, pois utiliza intensivamente a tradução dinâmica como técnica para prover a virtualização. É um dos poucos hipervisores recursivos, ou seja, é possível chamar o QEMU a partir do próprio QEMU. **Site:** https://www.qemu.org/

Valgrind	(Virtualização de Aplicações - compilação *just-in-time* (JIT)) - O *Valgrind* é uma ferramenta de depuração de uso da memória RAM e problemas correlatos. Ele permite investigar vazamentos de memória (*memory leaks*), acessos a endereços inválidos, padrões de uso dos caches e outras operações envolvendo o uso da memória RAM. O *Valgrind* foi desenvolvido para plataforma x86 Linux, mas existem versões experimentais para outras plataformas. **Site:** http://www.valgrind.org/
JVM	(Virtualização de Aplicações) - É comum a implementação do suporte de execução de uma linguagem de programação usando uma máquina virtual. Um bom exemplo dessa abordagem ocorre na linguagem Java. Tendo sido originalmente concebida para o desenvolvimento de pequenos aplicativos e programas de controle de aparelhos eletroeletrônicos, a linguagem Java mostrou-se ideal para ser usada na Internet. O que a torna tão atraente é o fato de programas escritos nessa linguagem de programação poder ser executados em praticamente qualquer plataforma. A virtualização é o fator responsável pela independência dos programas Java do *hardware* e dos sistemas operacionais: um programa escrito em Java, ao ser compilado, gera um código binário específico para uma máquina abstrata denominada máquina virtual Java (JVM - Java Virtual Machine). A linguagem de máquina executada pela máquina virtual Java é denominada *bytecode* Java, e não corresponde a instruções de nenhum processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las. **Site:** https://www.oracle.com/technetwork/java/index.html

4.6 Máquina Virtual Android (Dalvik, ART (Android Runtime)).

Quem é o responsável pela execução de aplicativos instalados no sistema operacional do seu computador ou smartphone? A resposta será idêntica e simples se estivermos falando de um computador comum ou celular com iOS ou Windows Phone. Quem executa essa função de executar aplicativos instalados é o SO. Simples.

Porém, quando falamos de Android a questão muda um pouco. Devido a grande variedade de equipamentos e componentes diferentes que cada um possui o sistema operacional do Google (Android) precisa usar uma máquina virtual, para criar uma camada de compatibilidade (uma interface de abstração).

É necessário compreender, então, que o Android utilizou uma **máquina virtual** chamada **Dalvik** até a versão **KitKat**, e depois foi descontinuada, sendo substituída pela **máquina virtual ART**.



***KitKat*:** As versões do sistema operacional Android foram apresentadas na unidade 3 de nosso material didático. Para saber mais faça a releitura desse tópico.

Dentro das camadas da Arquitetura Android temos a camada de Runtime, onde está à máquina virtual Dalvik, criada para cada aplicação executada no Android. Esse novo modelo de Máquina Virtual possui melhor desempenho, maior integração com a nova geração de *hardware* e foi projetada para executar vários processos paralelamente.

Na máquina virtual Dalvik as aplicações escritas em Java são compiladas em *bytecodes* Dalvik e executadas usando a Máquina Virtual Dalvik (JIT (*Just-In-Time*)), que é uma máquina virtual especializada desenvolvida para uso em dispositivos móveis, o que permite que programas sejam distribuídos em formato binário (*bytecode*) e possam ser executados em qualquer dispositivo Android, independentemente do processador utilizado. Apesar das aplicações Android serem escritas na linguagem Java, ela não é uma máquina virtual Java, já que não executa *bytecode* JVM.

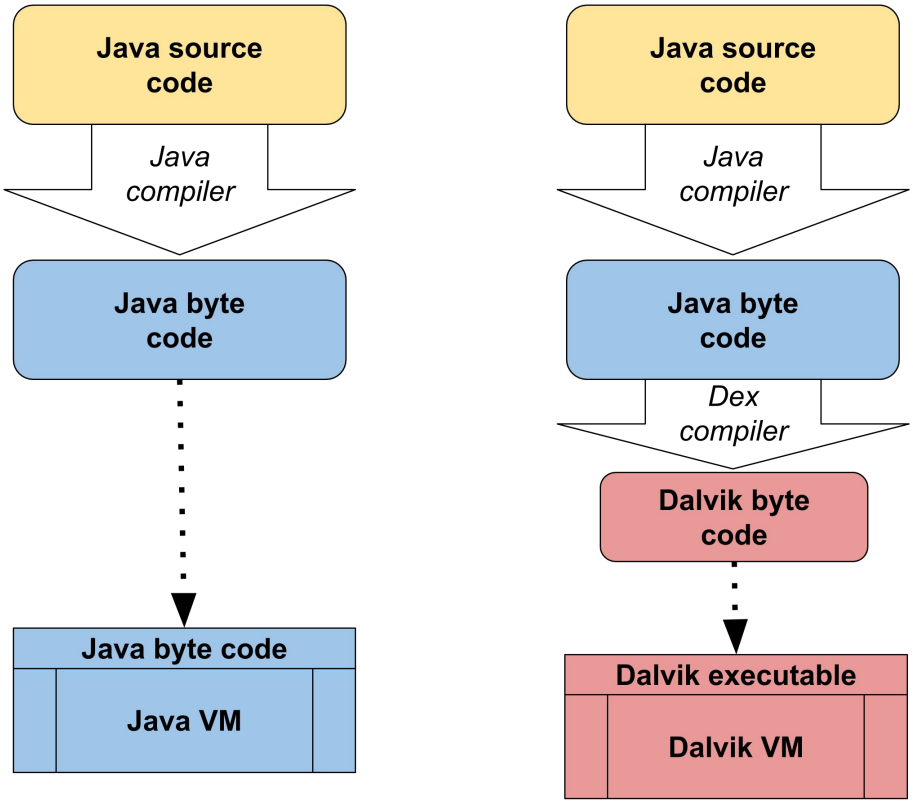


Figura 8 - Comparação entre JVM tradicional e VM Dalvik.
Fonte: Site Luiztools - Acesso em 01 Dez 2018. Disponível em:
http://www.luiztools.com.br/wp-content/uploads/2016/07/android-dalvik-300x223.png

Mas, afinal, o que é o Android *RunTime* (ART) então? O ART (*Ahead-of-Time Compilation*) é projetado para executar várias máquinas virtuais em dispositivos de baixa memória executando arquivos DEX, um formato de *bytecode* projetado especialmente para Android, otimizado para oferecer consumo mínimo de memória. É possível construir cadeias de ferramentas, como Jack, e compile fontes Java em *bytecodes* DEX, que podem ser executadas na plataforma Android. Alguns dos recursos principais de ART são:

- Compilação "*ahead-of-time*" (AOT) e "j*ust-in-time*" (JIT);
- Coleta de lixo (GC) otimizada;
- Melhor compatibilidade de depuração, inclusive um gerador de perfil de exemplo, exceções de diagnóstico detalhadas e geração de relatórios de erros, além da capacidade de definir pontos de controle para monitorar campos específicos.

Antes do Android versão 5.0 (API nível 21), o Dalvik era o tempo de execução do Android. Se o seu aplicativo executa o ART bem, deve funcionar no Dalvik também, mas talvez não vice-versa.

O Android também contém um conjunto das principais bibliotecas de tempo de execução que fornecem a maioria da funcionalidade da linguagem de programação Java, inclusive alguns recursos de linguagem Java 8 que a estrutura da Java API usa.



Palestra - *Campus Startup School*: Desvendando o ecossistema e a arquitetura Android

Nesta palestra, Neto Marin ensina os conceitos básicos de desenvolvimento para Android, incluindo a evolução da plataforma, como otimizá-la em todas as telas e dominar o processo de arquitetura do aplicativo.

Visite:
https://www.youtube.com/watch?v=hiJrgrujWeA



Bora rever!

Essa unidade, apresentou os fundamentos da virtualização. Foi possível perceber a importância de das técnicas de virtualização e como são organizados os ambientes de máquinas virtuais. Vimos, ainda, o funcionamento da máquina virtual Android (Dalvik e ART).

Virtualizar significa: simular, mascarar ambientes isolados, capazes de rodar diferentes sistemas operacionais (SO) dentro de uma mesma máquina. Este processo aproveita ao máximo a capacidade do *hardware*, que muitas vezes fica ociosa em determinados períodos do dia, da semana ou do mês.

A virtualização permite a criação de uma máquina virtual (VM), funcionando como uma camada de compatibilidade sobre o sistema operacional hospedeiro e *hardware* real, que pode simular o ambiente de outra máquina real, onde então é possível instalar outro sistema operacional convidado.

Utilizando uma máquina virtual (VM) podemos criar a imitação de uma máquina real. A simulação (imitação) criada por ela permite rodar sistemas operacionais que terão a ilusão de estar rodando na máquina real que a máquina virtual está simulando. Esse tipo de imitação pode ser realizado (criadas) como virtualização de sistema operacional, virtualização de *hardware* ou virtualização de linguagens de programação.



Bora rever!

A virtualização pode ser empregada para facilitar o desenvolvimento de *softwares* e sistemas operacionais. Podemos utilizar VMs para testar o SO em desenvolvimento, sem causar danos ou impactos. De igual forma os *softwares* em desenvolvimento podem ser testados em sistemas virtuais.

Podemos realizar virtualização empregando alguns sistemas disponíveis para trabalhar com máquinas virtuais em sistemas operacionais convencionais (*desktops* e servidores) como *VMWare*, *Virtualbox*, *FreeBSD Jails*, *Xen*, *User-Mode Linux*, *QEMU*, *Valgrind* e *JVM*. Eles são exemplos de máquinas virtuais de aplicação e de sistema, empregando virtualização total ou paravirtualização, além de abordagens híbridas.

E assim, chegamos ao fim do conteúdo desse capítulo. Revise os conteúdos sempre que tiver dúvidas sobre o tema.

Glossário

Chipset: é um conjunto de componentes eletrônicos de baixa capacidade, em um circuito integrado, que gerencia o fluxo de dados entre o processador, memória e periféricos. É normalmente encontrado na placa mãe.

***Hipervisor*:** um *hipervisor*, ou monitor de máquina virtual, é um *software*, *firmware* ou *hardware* que cria e roda máquinas virtuais (VMs). O computador no qual o *hipervisor* roda uma ou mais VMs é chamado de máquina hospedeira (*host*), e cada VM é chamada de máquina convidada (*guest*).

***Ahead-of-time*:** o ato de transformar o código-fonte em código de máquina é chamado de "compilação". Quando todo o código é transformado de uma só vez antes de atingir as plataformas que o executam, o processo é chamado de Compilação AOT (*Ahead-Of-Time*).

***Just-in-time*:** em Ciência da Computação, compilação *just-in-time* (JIT), também conhecida como tradução dinâmica, é a compilação de um programa em tempo de execução , usando uma abordagem diferente da compilação anterior à execução.

JVM: (do inglês *Java Virtual Machine* - JVM) é um programa que carrega e executa os aplicativos Java, convertendo os *bytecodes* em código executável de máquina. A JVM é responsável pelo gerenciamento dos aplicativos, à medida que são executados.

Interface: uma interface, em ciência da computação, é a fronteira que define a forma de comunicação entre duas entidades.

Abstração: é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais.

Referências

LECHETA, R. R. **Android Essencial**. São Paulo: Novatec, 2016.

LECHETA, R. R. **Google Android**. São Paulo: Novatec, 2016.

DEITEL, P.; DEITEL, H.; DEITEL, A. **Android para programadores**. 2a ed. Porto Alegre: Bookman, 2015.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 3a. ed. São Paulo: Pearson Pretice Hall, 2010.

MACHADO, Francis B. **Arquitetura de Sistemas Operacionais.** 4a. ed. Rio de Janeiro: LTC, 2011.

SILBERSCHATZ, Abraham. **Fundamentos de Sistemas Operacionais**. Rio de Janeiro: LTC, 2011.

OLSEN, Diogo Roberto. **Sistemas Operacionais**. Curitiba: Editora Livro Técnico, 2010.

MÓDULO III

ANÁLISE E PROJETO DE SISTEMAS

Angélica Toffano Sedel Calazans

Apresentação

O Módulo III desta obra nos mostra a importância em adotarmos todos os elementos que compõem a Análise e Projeto de Sistemas, ao iniciarmos um novo projeto. Vamos aprender, como é importante passarmos pelas etapas da análise e ter a visão de todo o projeto, antes de colocarmos a “mão” no código.

O módulo está estruturado em 4 unidades temáticas com tópicos específicos, conforme descrito abaixo.

Unidade 1: o que é e o que compõe um projeto de software.

Unidade 2: importância da modelagem de requisitos e suas ferramentas.

Unidade 3: conceito de modelagem de sistemas, modelos e suas perspectivas em relação a modelagem de sistemas orientados a objetos.

Unidade 4: UML e seus diagramas. Introdução a design patterns ou padrões de projeto.

Vamos começar?

Unidade 1 Levantamento, análise e negociação de requisitos

1.1 Projeto de software

Neste tópico, estudaremos o que é e o que compõe um projeto de software. Esse é um conceito-chave, pois é base de tudo que veremos nesta unidade.

Contudo, talvez seja mais fácil ir por partes.

O que é um projeto?

Segundo o PMI (2018), “projeto é o esforço temporário empreendido para criar um produto, serviço (...)”. Quando vamos construir uma casa temos um projeto. Temos o esforço de várias pessoas – arquiteto, engenheiro, pedreiro, servente – que vão executar, durante um tempo, uma série de atividades para construir o produto que é a casa.



Fonte: Disponível em: < https://pixabay.com/pt/software-cd-dvd-digital-disco-871026/>.



Bora Refletir!

E quando decidimos estudar informática, temos também um projeto? Quem está envolvido nesse projeto? Quais as suas atividades? Qual o produto final?

O que é um software?

Para Sommerville (2011), um software são "programas de computador e a documentação associada". No seu celular, existem uma série de softwares ou app. Você consegue identificar alguns deles?

Agora ficou fácil definir projeto de software.

Um projeto de software é um conjunto de atividades e resultados associados que produz um produto de software. (SOMMERVILLE, 2011)

Esclarecendo: quando falamos em projeto de software ou sistema estamos nos referindo também ao processo de desenvolvimento de software adotado para construção daquele software. Existem muitos processos e abordagens que se propõem à construção de um software.

Vamos ver um exemplo? Estamos desenvolvendo um Sistema de Biblioteca, ou seja, um projeto de software de biblioteca. Esse sistema, para ser desenvolvido, necessita seguir algum processo de desenvolvimento de software. Posso desenvolver esse sistema seguindo a Abordagem de Orientação a Objeto ou posso desenvolver esse sistema seguindo uma Abordagem de Métodos ágeis.



Figura 1 - Atividades comuns ao processo de desenvolvimento de sistema
Fonte: adaptada de Castro et al, 2014.

Não existe um processo ideal ou perfeito para a construção de um determinado software. Os processos de desenvolvimento evoluíram e estão constantemente evoluindo por vários motivos: o avanço da tecnologia, a capacidade das pessoas, o tipo de sistema a ser desenVolvido, tipo de organização que vai desenvolver ou utilizar o produto (Medeiros, 2004). Assim, embora existam muitos processos de sistema, algumas atividades são comuns a todos eles, conforme você pode visualizar na Figura 1.Essas atividades podem ser executadas em diferentes sequências e agrupadas em diferentes etapas de acordo com a metodologia adotada. A literatura apresenta uma série de modelos de processo de desenvolvimento de sistema, tais como: Cascata, Prototipação, Processo Unificado (Orientação a objetos), Métodos Ágeis (XP, SCRUM) e outros. Detalharemos esses modelos na Unidade 3.

Vamos agora detalhar melhor o que acontece na atividade relacionada aos requisitos.

1.2 A importância da Engenharia de Requisitos no projeto de software

Antes de verificarmos a importância da Engenharia de requisitos, vamos compreender o que é requisito de software.

Requisitos são descrições dos serviços oferecidos pelo software (SOMMERVILLE, 2011).

Ou

Requisito de software é uma AÇÃO que o software deve executar e que possui características e condições próprias para automatizar um processo ou necessidade do cliente (CASTRO et al, 2014).



Bora Refletir!

Gosto sempre de fazer a comparação com a construção de uma casa. Quando vamos construir, temos que definir a metragem da casa, a quantidade de quartos, portas, janelas etc. Com relação ao desenvolvimento de software, temos que identificar o que o cliente quer. Por exemplo: se fôssemos desenvolver um aplicativo ou software de comércio eletrônico de roupas, ele precisaria de uma listagem de todos os produtos? Se sim, essa listagem deve ter um filtro para que o cliente possa filtrar o tipo de roupa que está procurando? Um filtro por preço? Tudo isso são os requisitos do sistema. Você consegue imaginar mais alguns requisitos com relação a este contexto? Procure na internet, entre em sites de comércio eletrônico e veja que tipo de consultas eles proporcionam. Cada consulta identificada é um requisito que foi definido.

Veja a tirinha do Dilbert sobre requisitos. Retrata bem a dificuldade de elicitar requisitos do cliente.



Figura 2: Tirinha sobre Requisitos
Fonte: Disponível em: <https://esdepre.files.wordpress.com/2012/03/dilbert-requisitos-de-sistema.jpg>.

Assim, podemos afirmar que Engenharia de Requisitos (ER) visa definir as formas para elucidar, organizar e documentar os requisitos de maneira sistêmica. Para tanto, a compreensão completa do problema, a definição dos requisitos do software e sua especificação detalhada é fundamental para a obtenção de um software com alta qualidade.

Não importa quão bem projetado ou codificado esteja um programa, se os requisitos forem mal analisados e especificados, futuramente poderá trazer problemas para o cliente e o desenvolvedor. Por isso, a Engenharia de Requisitos (ER) é importante para a compreensão da Engenharia de Software (ES).

O processo de Engenharia de requisitos é composto de várias etapas conforme a Figura 3. A seguir, vamos detalhá-las para que você compreenda e consiga trabalhar com requisitos de forma eficaz.



Figura 3 – Principais etapas do processo de Engenharia de requisitos
Fonte: (IEEE, 2014)

Na fase de Elicitação se tenta obter o máximo de informações possíveis para o conhecimento do sistema. Busca-se, nesta fase, responder questões do tipo: quais os problemas do cliente? Quem são os atores envolvidos? Quais as necessidades com relação ao sistema? Qual o escopo do sistema? Estaremos detalhando melhor essa fase no próximo tópico.

Nessa etapa, se você estiver utilizando a abordagem Orientada a Objeto, você provavelmente fará um documento de visão ou mesmo um caso de uso relativo ao documento de visão detalhando essas informações; se estiver utilizando uma abordagem ágil, poderá ter que criar um cartão de visão (no caso da abordagem extreme programming ou XP).

Na fase de análise, aprofunda-se sobre o conhecimento obtido na fase Elicitação e começa-se a identificar as funcionalidades do software a ser desenvolvido. Para atender às necessidades do cliente, quais as funções ou módulos o software deve ter?

Na fase de especificação, já conhecendo as funções ou módulos, iniciamos a escrita dos requisitos. Para isso, temos que conhecer os principais tipos de requisitos. No tópico classificação de requisitos detalharemos melhor esse assunto.

Lembra que existem vários tipos de metodologias para desenvolver o produto de software?

A literatura apresenta uma série de modelos e/ou abordagens de processo de desenvolvimento de software, tais como Cascata, Espiral, Orientado a objeto, RUP, Métodos Ágeis.

Assim, se estivermos desenvolvendo seguindo a abordagem Orientada a objeto no momento da especificação, estaremos desenhando os diagramas de caso de uso e descrevendo os casos de uso. Se utilizarmos uma abordagem de Métodos ágeis, por exemplo o SCRUM, estaremos escrevendo história de usuários ou talvez casos de uso, dependendo de como a organização define os requisitos no modelo ágil. Detalharemos essas duas técnicas, casos de uso e história de usuário, na próxima unidade.

Na fase de validação se obtém o aceite do cliente sob determinado artefato que foi gerado. Esses artefatos podem ser os documentos de visão, diagramas, os casos de uso, as histórias de usuário. Esta atividade significa aprovar junto ao cliente as restrições, escopo, os diagramas de caso de uso e as descrições de caso de uso (se estiver utilizando a abordagem OO) ou história de uso (se estiver utilizando metodologia ágil) etc. Lembre-se de que o cliente deve registrar a validação realizada. Isso é realizado por meio de atas de reunião validadas, e-mail e outros documentos.

Todas essas tarefas são complexas, dependem do entendimento, da comunicação entre pessoas que têm visões diferentes (negócios vs técnica). São resultados de muitas horas de trabalho. Para entender melhor essas dificuldades vamos ler o texto a seguir.



Saiba mais!

Elicitar e analisar requisitos não são tarefas fáceis. Leia o texto "A difícil arte de Analisar requisitos" (Disponível em: https://esdepre.wordpress.com/2012/03/26/a-dificil-arte-de-analisar-requisitos/).
Ele demonstra as principais dificuldades para realizar essas atividades.

Vamos agora entender melhor o que é feito na etapa Elicitação de requisitos.

1.3 Principais conceitos: atores (organizações, sistemas, clientes), stakeholders, fronteiras, restrições

Na etapa de Elicitação, conforme já vimos, temos que identificar as principais necessidades do cliente com relação ao sistema, os atores envolvidos (cliente, organizações, equipamentos, sistemas etc.), as fronteiras e restrições.

Aqui é importante envolver e ouvir o cliente. O sistema deve atender às suas necessidades.

Antes de entendermos as perguntas que deverão ser respondidas, nesta etapa, é interessante entender o conceito de stakeholders, atores, fronteiras e restrições.

O que são atores (organizações, equipamentos, sistemas)?

Quando falamos em atores, na Engenharia de Requisitos, estamos identificando todos os personagens que vão interagir com projeto do sistema a ser construído. Bezerra (2006) acredita que o ator possui um papel no sistema e indica algumas categorias de atores: cargos (empregado, cliente, gerente etc.), organizações (Administradora de cartões, Correios etc.), outros sistemas (CADIN, SERASA, Receita Federal etc.), equipamentos (leitora de código de barras, sensor etc.). Atores representam o papel exercido por uma pessoa que interage com o sistema (Bezerra, 2006).

Mas o que são stakeholders?

Segundo Sommerville (2011), o termo stakeholder é utilizado para se referir a qualquer pessoa ou grupo afetado pelo sistema, direta ou indiretamente. Podem ser os usuários finais, engenheiros que estão desenvolvendo sistemas relacionados, gerentes de negócio, especialistas daquele domínio. Podem ou não ser atores do sistema. Se interagirem com o sistema serão atores com a denominação correta: cliente, fornecedor, gerente etc.

Para identificar os stakeholders algumas perguntas podem ser feitas: Quem definirá as necessidades, o escopo do sistema? Quem vai utilizar o sistema? Quem será afetado pelas saídas do sistema? Quem vai avaliar e aprovar o sistema quando ele for entregue?

Assim, é muito importante identificar corretamente os stakeholders do seu projeto. Já pensou se você se esquecer de identificar algum stakeholder? Isso pode impactar no seu projeto, você não acha? Podem ser esquecidos alguns requisitos importantes.



Fique Ligado!

E os desenvolvedores, são atores? Não. Os desenvolvedores ou analistas participam da criação do sistema. Participam do processo de desenvolvimento do sistema. Se eles depois vierem a utilizar o sistema com o papel de cliente, aí sim eles terão esse papel, pois estarão interagindo com o sistema na condição de cliente.

Ficou claro?

Outro conceito importante são as fronteiras do sistema. A fronteira é o limite de atuação do sistema. Na definição da fronteira estamos identificando o que estará dentro e o que estará fora dos objetivos do sistema.

Imagine que vamos desenvolver um sistema de biblioteca. O nosso sistema inicialmente só terá como objetivos a manutenção do acervo (livros) e dos clientes (empréstimo, devolução e reserva de livros). Poderíamos identificar então como atores: o cliente e a bibliotecária.

Não estão nessa fronteira a compra de livros (departamento financeiro), a contabilidade da biblioteca (departamento contábil) etc. E, por isso, não são demonstrados como atores. E se mais adiante se identificar a necessidade de uma ligação com o sistema financeiro (para a compra de novos livros)? Então teremos mais um ator representado pelo sistema financeiro.A Figura 4 apresenta como poderíamos representar a fronteira desse sistema na versão inicial.

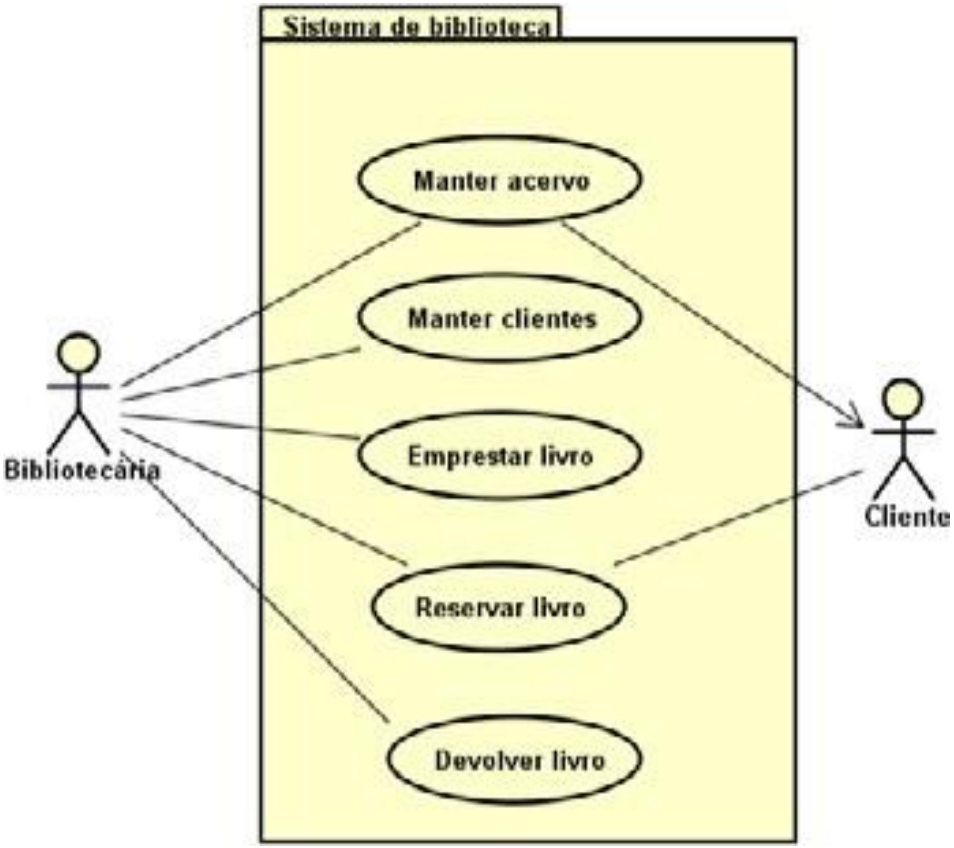


Figura 4 – Exemplo de Caso de uso identificando a fronteira do Sistema de biblioteca
Fonte: CASTRO et al, 2014.

Finalmente, vamos para o último conceito importante que são as restrições.

O que são restrições?

Restrições são as limitações internas ou externas ao projeto de sistema. Elas existem para definir os limites do projeto e impactam o projeto. Algumas delas podem ser constatações de que ações podem ser realizadas para diminuir ou eliminar essas restrições.

Alguns exemplos:

Vamos pensar que estamos construindo um sistema para o lançamento de um e-commerce para a venda de produtos de natal (árvores de natal, bolas, enfeites). Concorda que a restrição seria o sistema estar pronto até antes do Natal? Essa seria uma restrição com relação ao prazo.

Os computadores de determinada empresa recebem manutenção todos os finais de semana. Você concorda que uma restrição é o sistema a ser desenvolvido não estar disponível todos os dias? Um sistema disponível todos os dias é identificado como 7/24 (sete dias com 24 horas). Assim, o sistema a ser desenvolvido seria 5/24 (cinco dias com 24 horas? Esse é um exemplo de restrição tecnológica.

E com relação a custo? Você consegue pensar em algum tipo de restrição possível?

Se o seu stakeholder que define o sistema junto à equipe de desenvolvimento só pode estar disponível 1 dia por semana, isso pode ser considerado uma restrição? Por quê?

Se o sistema precisa de 50 GB no servidor, que atualmente só tem 30 GB, você considera isso um exemplo de restrição? Por quê?



Saiba mais!

Se quiser entender melhor as restrições, esse texto da equipe AEVO detalha mais esse assunto. Ele também aborda o conceito de premissas, que é muito interessante.
http://blog.aevo.com.br/premissas-e-restricoes-em-projetos-o-que-eu-preciso-saber/

Vamos voltar agora às atividades executadas durante a Elicitação. Agora que já entendemos alguns conceitos como stakeholders, atores, fronteiras, restrições é importante lembrar de algumas questões que devem ser respondidas nessa etapa, tais como (SBROCCO, 2012):

- Quais os problemas que estamos tentando resolver?
- Quais os objetivos do sistema que vamos desenvolver?
- Quem são os stakeholders?
- Quais são as fronteiras?
- Quais são as restrições?

Como já dissemos, dependendo a metodologia adotada no projeto de desenvolvimento de sistemas, essas informações comporão um documento de visão (Orientação a objeto), um cartão de visão (Metodologia ágil). O mais importante aqui não é o documento em que as informações são inseridas, mas o conteúdo dessas informações ser o mais aderente às necessidades dos stakeholders, de forma a garantir a completa elicitação das informações iniciais necessárias para a construção de um sistema.

Alguns autores citam o Método 5W2H para facilitar e complementar esse trabalho de elicitação. É o que veremos a seguir.

1.4 Método 5W2H (What, Who, Why, When, Where, How, How much)

O método 5W2H facilita a identificação de forma organizada das informações em qualquer projeto, inclusive projetos de TI. Representa as palavras em inglês *What* (O quê), *Who* (Quem), *Why* (Por quê), *When* (Quando), *Where* (onde), *How* (como) e *How much* (Quanto). Essas perguntas correspondem às perguntas básicas relacionadas a requisitos (SBROCCO, 2012).

Vamos detalhar?

What – O que o sistema tem que fazer? Quais os seus os problemas? Quais os objetivos? Quais as funções ou módulos? Quais as entradas? Quais as saídas?

Who – Quem são os atores envolvidos? Quem são os stakeholders? Quem vai validar as entregas do sistema?

Why – Por que esse processo existe? Por que estamos desenvolvendo esse produto?

hen – Quando será executado? Quando será avaliado?

Where – Onde será executado? Onde será avaliado?

How – Como será executado?

How much – Quanto irá custar?



Bora Refletir!

Vamos voltar ao nosso Sistema de biblioteca:
Você consegue responder a algumas dessas perguntas, outras você necessitaria discutir mais com o stakeholder. Você consegue identificar quais as perguntas você responderia e quais seriam necessários mais detalhamento?



Saiba mais!

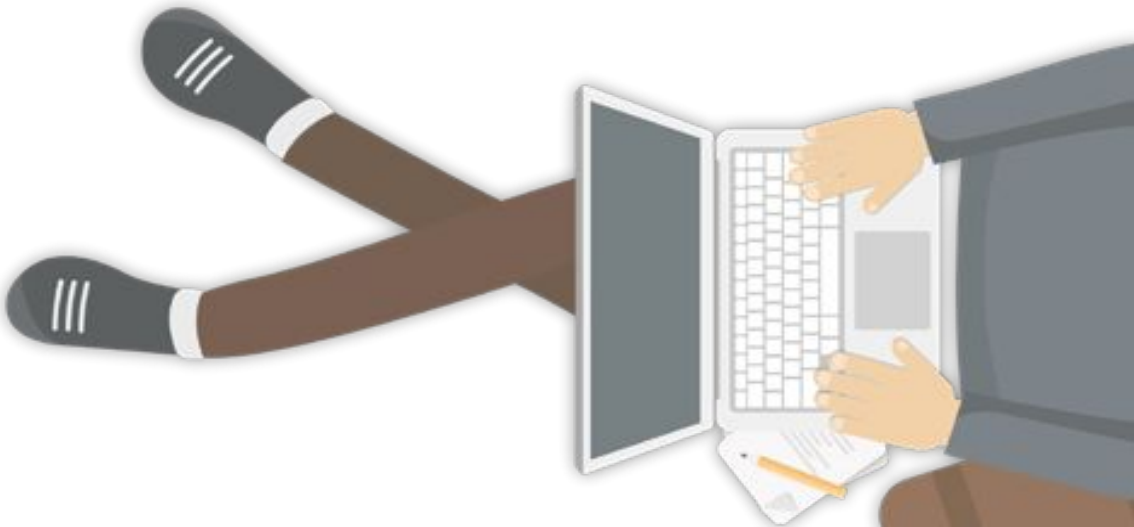
Veja o vídeo de Suedilson Filho sobre o 5W2H; ele complementa as informações sobre o método.
https://www.youtube.com/watch?v=o1nqbxTzqMU

A seguir, veremos alguns conceitos sobre a classificação dos requisitos. Lembrando: após a Elicitação, temos a Análise e a Especificação dos requisitos. E para isso, precisamos entender que existem tipos de requisitos diferentes. Esse assunto será muito importante para a próxima unidade, na qual exercitaremos a escrita de requisitos.

1.5 Classificação de requisitos

Os requisitos são normalmente classificados em requisitos funcionais, não funcionais e de domínio (SOMMERVILLE, 2011).

Requisitos funcionais definem o que o sistema deve fazer, como o próprio nome diz representam as funcionalidades do sistema. Descrevem que necessidades de negócio devem atender, os cadastros, os relatórios, as funcionalidades tangíveis ao usuário. No caso de utilizar a abordagem Ágil (SCRUM), os requisitos funcionais são cadastrados no “Product Backlog” e posteriormente se transformam em “estórias”. No caso da abordagem Orientada a Objeto são definidos por meio de casos de uso.



Alguns exemplos de requisitos funcionais em descrição de caso de uso considerando o exemplo do sistema de biblioteca:

. este caso de uso destina-se a possibilitar uma bibliotecária realizar a manutenção do cadastro dos clientes através das operações de inclusão, alteração, exclusão e consulta;

. este caso de uso destina-se a possibilitar uma bibliotecária realizar a manutenção do acervo dos livros através das operações de inclusão, alteração, exclusão e consulta.

Os requisitos não funcionais são aqueles não relacionados diretamente ao objetivo do sistema, mas que determinam “como” e “onde” o sistema deve funcionar. Por exemplo, critérios de segurança e criptografia, escalabilidade, disponibilidade, compatibilidade, portabilidade, etc. Além disso, os requisitos não funcionais também são cadastrados como itens do product backlog.

Alguns exemplos de requisitos não funcionais em linguagem natural considerando o exemplo do sistema de biblioteca:

. as funcionalidades de consulta ao acervo devem funcionar tanto em Firefox como Internet Explorer (característica de portabilidade);

. todas as funcionalidades do sistema devem seguir o padrão visual da instituição (cores, layouts, fontes) referente à Característica de conformidade.

Ambos os requisitos funcionais e não funcionais fazem parte do escopo do produto. E tanto um como outro são muito importantes para garantir a eficácia do sistema. Todos os requisitos devem ser testáveis, ou seja, devem ser escritos de forma a que seja possível testá-los.

Saiba mais!

https://www.youtube.com/watch?v=V74qIKo-OqI

Os requisitos de domínio são derivados do domínio do sistema. Podem ser funcionais ou não funcionais. Podem ser requisitos técnicos relacionados a padrões de criptografia, de acesso, velocidade, interfaces padrões etc. Podem também ser requisitos específicos de negócio, cálculos específicos do domínio negócio, necessários para que o sistema funcione. Podem ser restrições.

Alguns exemplos de requisitos de domínio em linguagem natural considerando o exemplo do sistema de biblioteca.

A interface do usuário com o banco de dados deve ser baseada no padrão Z39.50.

A codificação do acervo de livros deve obedecer ao padrão Classificação Decimal de Dewey (CDD).



Figura 5 -Tirinha
Fonte: imagem disponível em:
https://oblogdotarcisio.wordpress.com/tag/requisitos/



Bora rever!

Nesta unidade você estudou os principais conceitos relacionados ao Projeto de Sistema. Entendeu que quando falamos de projeto de sistema estamos nos referindo também ao processo de desenvolvimento de sistema. Que existem muitos processos/abordagens que se propõem a construção de um sistema. Não existe um processo ideal, é importante escolher, de acordo com o projeto, as necessidades, a arquitetura e o processo mais adequado ao seu sistema.

Independente do processo escolhido, a atividade de requisitos existe em todos os processos.

Estudamos os conceitos de requisitos e as etapas da Engenharia de Requisitos. Entendemos os principais conceitos de atores, stakeholders, fronteiras e restrições. Vimos ainda as informações essenciais para a elicitação das necessidades dos clientes para a construção do sistema. Estudamos que o método 5W2H pode ajudar a obter mais informações para a etapa de Elicitação. Por fim, estudamos os requisitos funcionais, não funcionais e de domínio.

Espero que tenha gostado e assimilado estes conhecimentos de forma a aplicá-los na sua vida profissional.

Nas próximas unidades detalharemos melhor esses conceitos e aprenderemos outros. Vamos lá?

Unidade 2

Modelagem, especificação, validação e verificação de requisitos

2.1 – A importância da modelagem e suas ferramentas

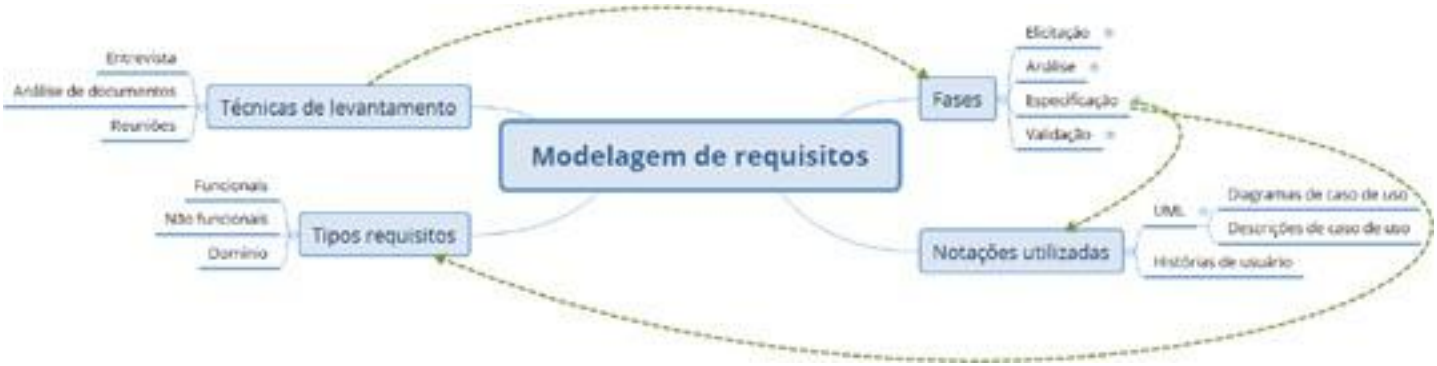


Figura 1 – Conceitos de modelagem de requisitos.
Fonte: elaborado pela autora.

Neste tópico, estudaremos a importância da modelagem de requisitos e suas ferramentas. A Figura 1 apresenta os principais conceitos relacionados à modelagem de requisitos. Alguns desses conceitos já estudamos na unidade anterior, como as fases da Engenharia de requisitos (Elicitação, Análise, Especificação e Validação). Também vimos classificação de requisitos em Requisitos funcionais, não funcionais e de domínio. Lembra? Todos esses conceitos estão bastante interligados.

Analisando a Figura 1, você pode verificar o link que existe entre a fase de especificação, as notações utilizadas para representar os requisitos e os tipos de requisitos. Lembre-se de que na fase de especificação você deve começar a escrever os requisitos (funcionais, não funcionais e de domínio). A forma dessa escrita vai depender da notação adotada pelo projeto de sistemas, UML, histórias do usuário etc.

Vamos nos aprofundar nessa discussão. Inicialmente, vamos ver o conceito de modelagem de requisitos.

O que é modelagem de requisitos?

A modelagem de requisitos tem como objetivo a construção de modelos que representem os requisitos do sistema. Normalmente, utiliza-se algum tipo de notação ou ferramenta para modelar os requisitos.

A modelagem de requisitos é realizada com o objetivo de simplificar a complexidade para representar as necessidades do sistema (MEDEIROS, 2004). Não temos a facilidade de entender e gravar todas as informações sobre tudo que o software precisa realizar, por isso utilizamos modelos. Normalmente é mais fácil compreender modelos desenhados do que textos escritos. Você concorda?

Veja os exemplos a seguir:



Figura 2. Emojis utilizados em aplicativos para conversas
Fonte: Whatsapp.



Figura 3. Imagem do Campus Brasília do IFB.
Fonte: retirada do Google Maps.

Mas, que notações existem para a modelagem de requisitos?

No caso de Sistemas Orientados a objetos, a UML – Linguagem de Modelagem Unificada (Unified Modeling Language) – é a linguagem utilizada para modelar esses sistemas. A UML auxilia os desenvolvedores a definir as características do software, seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos. A modelagem de requisitos nesse contexto utiliza diagramas de casos de uso e descrições de casos de uso, lista de requisitos não funcionais para representar os requisitos não funcionais e de domínio (se necessário).

No caso de sistemas construídos com metodologias ágeis, a notação utilizada para os requisitos são normalmente histórias de usuário, casos de uso, lista de requisitos não funcionais etc. Isso depende da metodologia adotada (XP, SCRUM ou outra) e dos padrões da organização.

Vamos conhecer um pouco mais de UML? Vamos ver um vídeo muito interessante sobre a UML: https://www.devmedia.com.br/uml/8579 .

Quer esclarecer melhor esse assunto? Leia o texto sobre o conceito de modelagem e diagramação.

Leitura complementar

http://spaceprogrammer.com/uml/introduzindo-o-conceito-de-modelagem-e-diagramacao/

Qualquer que seja a notação utilizada, UML ou outra, é importante entender que a modelagem de um sistema é uma forma de documentá-lo. Mas, para que documentar?

Os sistemas estão em constante mudança. Os clientes desejam melhorias, o mercado força a adoção de estratégias para que as empresas se tornem mais competitivas, as leis são modificadas com o passar dos anos, os sistemas podem ter falhas. Todos esses fatores e outros podem repercutir em mudanças no sistema que foi desenvolvido ou mesmo que está sendo desenvolvido. E uma documentação completa vai ajudar para que a manutenção do sistema ocorra de maneira correta, sem gerar novos erros ao corrigir os antigos.

Agora vamos conhecer algumas das técnicas de levantamento de requisitos. Qualquer que seja a notação utilizada, UML ou outra, é importante entender que a modelagem de um sistema é uma forma de documentá-lo. Mas, para que documentar?

Os sistemas estão em constante mudança. Os clientes desejam melhorias, o mercado força a adoção de estratégias para que as empresas se tornem mais competitivas, as leis são modificadas com o passar dos anos, os sistemas podem ter falhas. Todos esses fatores e outros podem repercutir em mudanças no sistema que foi desenvolvido ou mesmo que está sendo desenvolvido. E uma documentação completa vai ajudar para que a manutenção do sistema ocorra de maneira correta, sem gerar novos erros ao corrigir os antigos.

Agora vamos conhecer algumas das técnicas de levantamento de requisitos.

2.2 – Técnicas de levantamento de requisitos

São muitas as técnicas que podem ser utilizadas para o levantamento de requisitos. Agora, estudaremos as mais utilizadas: análise de documentos, entrevistas, workshop/reunião ou oficina, prototipação (CASTRO et al, 2014).

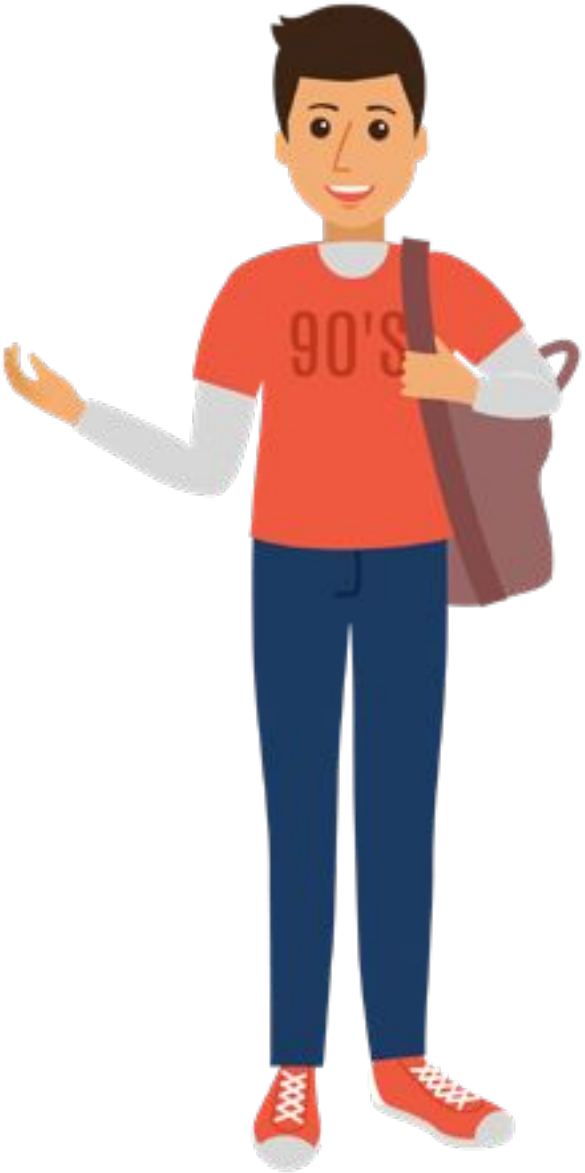
A análise de documentos é um passo muito importante na elicitação de requisitos. É necessário conhecer a organização, as áreas envolvidas, o sistema existente (se for uma manutenção) ou as metas e objetivos organizacionais com relação ao sistema a ser desenvolvido. É importante para o desenvolvedor ir à primeira entrevista já com um conhecimento prévio do ambiente, do sistema, da organização. Essas informações podem ser obtidas na internet (site da organização) ou mesmo em documentos organizacionais que sejam entregues pelos stakeholders para a equipe.



A entrevista é a técnica de elicitação mais utilizada e de simples implementação. Isso se deve ao fato de que a comunicação utilizada nas entrevistas é mais natural, simples e pode ser utilizada em várias situações.

O objetivo da entrevista é a obtenção de informações, fatos vivenciados, tendências e experiências sobre a empresa e suas atividades. Fornece uma visão concreta da realidade. Os requisitos são derivados das respostas a essas questões.

Segundo Sommerville (2011), as entrevistas podem ser fechadas e abertas. Fechadas quando o desenvolvedor leva uma série de questões pré-definidas a serem respondidas; e abertas quando não existe um roteiro pré-definido de questões e a equipe de desenvolvedores explora diversos assuntos objetivando o entendimento do sistema.





Fique Ligado!

Para realizar uma entrevista, é necessário que o desenvolvedor se prepare, considerando os seguintes pontos (Castro et al, 2014):

- Identificar ideias gerais sobre a organização, o tipo de produto, o domínio do problema;
- Estudar os documentos organizacionais;
- Identificar as pessoas a serem entrevistadas;
- Definir objetivos, escopo e duração da entrevista (previsão do tempo da entrevista);
- Preparar um roteiro;
- Providenciar material necessário para a entrevista.

Além disso, é importante que ao iniciar a entrevista, o entrevistador apresente a equipe, os objetivos da entrevista, a previsão do tempo a ser gasto. E, se for gravar, que peça autorização. Ao término desta, ele não pode se esquecer de agradecer, fazer um breve resumo do que foi discutido, informar como essas informações serão validadas pelo cliente (ata, e-mail etc.) e, se necessário, marcar a próxima entrevista.



Saiba mais!

Reforce essas etapas assistindo ao vídeo sobre essa técnica.
https://www.youtube.com/watch?v=XqnSRGGBDuU

A técnica de workshop, também chamada de oficina, tem por objetivo juntar os stakeholders de um projeto por um determinado período de tempo para discutirem os aspectos mais importantes para o desenvolvimento da aplicação. Trata-se de uma técnica de elicitação em grupo usada em uma reunião estruturada. Fazem parte do grupo os desenvolvedores e uma seleção dos stakeholders que melhor representam o contexto em que o sistema será usado, possibilitando a obtenção de um conjunto de requisitos bem definidos. O workshop tem o objetivo de incentivar o trabalho em equipe, o consenso e a concordância em um curto espaço de tempo (CASTRO et al, 2014).

O workshop traz alguns benefícios, tais como: envolvimento de todos os interessados, agiliza o entendimento do projeto do Sistema, facilita a obtenção de consenso, pois agrega as partes interessadas.



Fique Ligado!

O workshop necessita também de uma preparação detalhada que envolve:

definição de participantes, definição de agenda, logística, distribuição do material com antecedência para que todos os participantes cheguem à reunião informados sobre o que será discutido e o que se espera como resultado e, por fim, sensibilização envolvendo os benefícios da sessão.

Os workshops são, muitas vezes, realizados em uma série de sessões, com duração de 3 ou 4 horas cada. Estas sessões são realizadas duas ou três vezes por semana e, em média, possuem 4 a 12 participantes. Isto inclui tanto o domínio do negócio como da solução (CASTRO et al, 2014).

Antes dos workshops, normalmente, são realizadas entrevistas para que os desenvolvedores entendam o domínio do negócio, a política, o problema e os objetivos do projeto.

A técnica de Prototipação (BEZERRA, 2006) serve de complemento na análise de requisitos. Protótipos são esboços e podem ser criados para telas de entrada, de saída, consultas ou mesmo o sistema inteiro.

Na Engenharia de software, após a elicitação de requisitos, o protótipo é construído para ser validado junto aos usuários. Essa técnica assegura que os requisitos foram bem entendidos e que começaremos a produzir o produto correto.

As definições de requisitos são conceitos abstratos. Retratam o que o cliente pretende que o software faça. Retratam um pensamento. Os protótipos são coisas concretas, telas de entrada, de saída. Para o usuário trabalhar com coisas concretas é mais fácil; e para os desenvolvedores é a garantia de que o que foi entendido no processo de comunicação estava correto.

Vamos nos aprofundar nesse assunto? Existem vários tipos de prototipação...



Saiba mais!

Vamos assistir ao vídeo sobre a importância da prototipação:
https://www.youtube.com/watch?v=RI1XH64NC0Q

Vamos melhorar nossos conhecimentos? Leia sobre a importância e os tipos da prototipação e sobre as ferramentas que existem para construir protótipos.



Saiba mais!

Para complementar seus conhecimentos, acesse:
http://dextra.com.br/pt/blog/prototipacao-e-sua-importancia-no-desenvolvimento-de-software/

2.3 – Técnicas de especificação de requisitos - Orientação a objetos e métodos ágeis

Acabamos de estudar várias técnicas para levantar requisitos. O levantamento ou elicitação de requisitos não é uma tarefa fácil. E a utilização de técnicas ajuda a entender melhor as necessidades do cliente. A seguir, veremos técnicas de especificação de requisitos com foco na Orientação a objetos e nos Métodos ágeis.



Fonte: Imagem disponível em: <http://sosvip.blogspot.com/2010/05/fonte-dos-desejos-cuidado-com-o-que.html>.

Neste tópico vamos detalhar sobre como especificar requisitos considerando duas abordagens: a Orientação a objetos e os Métodos Ágeis.

Você recorda o que vimos na Unidade 1?



Fique Ligado!

A literatura apresenta uma série de modelos de processo de desenvolvimento de sistema, tais como: Cascata, Prototipação, Processo unificado (Orientação a objetos), Métodos ágeis (XP, SCRUM) e outros. Detalharemos esses modelos na Unidade III.

Na abordagem Orientação a objetos, as etapas de elicitação, análise e especificação, os requisitos podem utilizar uma ou mais das técnicas de levantamento de requisitos do tópico II. Muitas empresas utilizam entrevista e prototipação. Outras empresas, somente entrevistas. O mais importante é obter todas as informações sobre as necessidades do cliente com relação ao sistema. É identificar o escopo, as fronteiras, os atores e stakeholders.

Essas informações, na maior parte das empresas, são inseridas em um documento de visão. Sobre isso, Medeiros (2004) apresenta um modelo de documento de visão bem detalhado, mas sugere que cada organização customize o seu as suas necessidades. Nem todas as informações são necessárias para todos os sistemas. Veja no box abaixo:

Modelo de Documento Visão - Detalhado segundo Medeiros (2004, p.23-24)

1. **Introdução**

Reserve esta seção para uma descrição geral do trabalho esperado.

2. **Escopo**

Relacione os principais módulos que serão detalhados em perspectivas do produto (Veja os itens 8.1, 8.2 etc.).

3. **Definições Acrônimos e Abreviaturas**

Faça o possível para não utilizar termos relativos à área de informática, se isso for imprescindível, então forneça o significado desses termos nessa seção.

4. **Referências**

Relate a fonte na qual esse documento se baseou e os participantes desses eventos. Nomeie as pessoas, seus cargos e localidades.

5. **Oportunidades de Negócio**

Descreva qual é a oportunidade de negócio que está em apreciação como motivação para o surgimento desse software.

5.1 Problema a Ser Solucionado

Descreva qual é o problema que está sendo solucionado com esse software.

6. Descrições de Stakeholder e Usuários

Relacione os stakeholders com seus cargos, nomes, telefones, e-mails; e faça o mesmo com os usuários principais. Isso será uma boa referência para futuros participantes desse software.

1. **Stakeholder**
2. **Usuários**
3. **Ambiente Atual dos Clientes**

   Descreva o ambiente atual de informática relativo a esse software que será construído, como sistema operacional, tecnologias e outros que considerar importante.

7. Observação

Agregue qualquer informação que achar importante.

8. Módulos

1. Perspectiva do Produto: Nome do Módulo I
2. Perspectiva do Produto: Nome do Módulo II

...

8.N Perspectiva do Produto: Nome do Módulo N

...Descreva nesses tópicos os detalhes que cada módulo atenderá. Procure descrever o que o módulo faz, para que serve e como se relacionará com outros módulos, se isso for importante. Cada módulo será um componente no diagrama de "Caso de Uso" que representará o <sistema>>.

9. Precedência e Prioridades

Indique qual é a prioridade dos módulos e se ela será a mesma ordem de precedência para o desenvolvimento. Explique por que adotou essa estratégia e, principalmente, por que deixou de usar outras. Explicar isso pode ser importante para futuras reuniões, nas quais podem perguntar: "Mas, por que não fizemos isso dessa maneira?"

10. Requisitos Não Funcionais

Relate aqui as dependências de softwares externos, particularidades de performance que são esperadas, grandes necessidades de treinamento e outras que achar necessário.

11. Requisitos de Sistemas e Ambientes

Descreva o sistema operacional elegido, a linguagem de programação e os bancos de dados que serão utilizados. Nessa seção, informe se será utilizado um servidor de aplicação, qual é esse servidor, assim como o servidor de internet e outros softwares dos quais ele dependerá.

1. **Sistema Operacional**
2. **Linguagens de Desenvolvimento**
3. **Banco de Dados**

12. Requisitos de Documentação

Descreva em quais partes a documentação será dividida, quais são essas partes e o que o cliente pode esperar de cada uma.

13. Visão Geral UML do Sistema - Modelo Conceitual

Insira uma figura que represente o diagrama de Caso de Uso baseado neste documento.

Em relação aos requisitos funcionais, estes são especificados por meio de diagramas de casos de uso e descrições de caso de uso. Os requisitos não funcionais, normalmente, são definidos em listas de requisitos ou no documento de visão.

Um diagrama de caso de uso é um diagrama comportamental da UML que evidencia a visão externa do sistema a ser desenvolvido. Ele mostra quais são as funcionalidades desse sistema e quem interage com elas (BOOCH et al, 2004).

O diagrama de caso de uso mostra o que o sistema deve fazer sem se fixar no como, apresenta de forma gráfica as funcionalidades, possui convenções simples e de fácil entendimento para o usuário final; sua documentação pode ser detalhada em versões futuras.

Vamos voltar ao caso de uso do sistema de biblioteca? Vamos entender melhor a Figura 4, considerando as notações da Tabela 1?

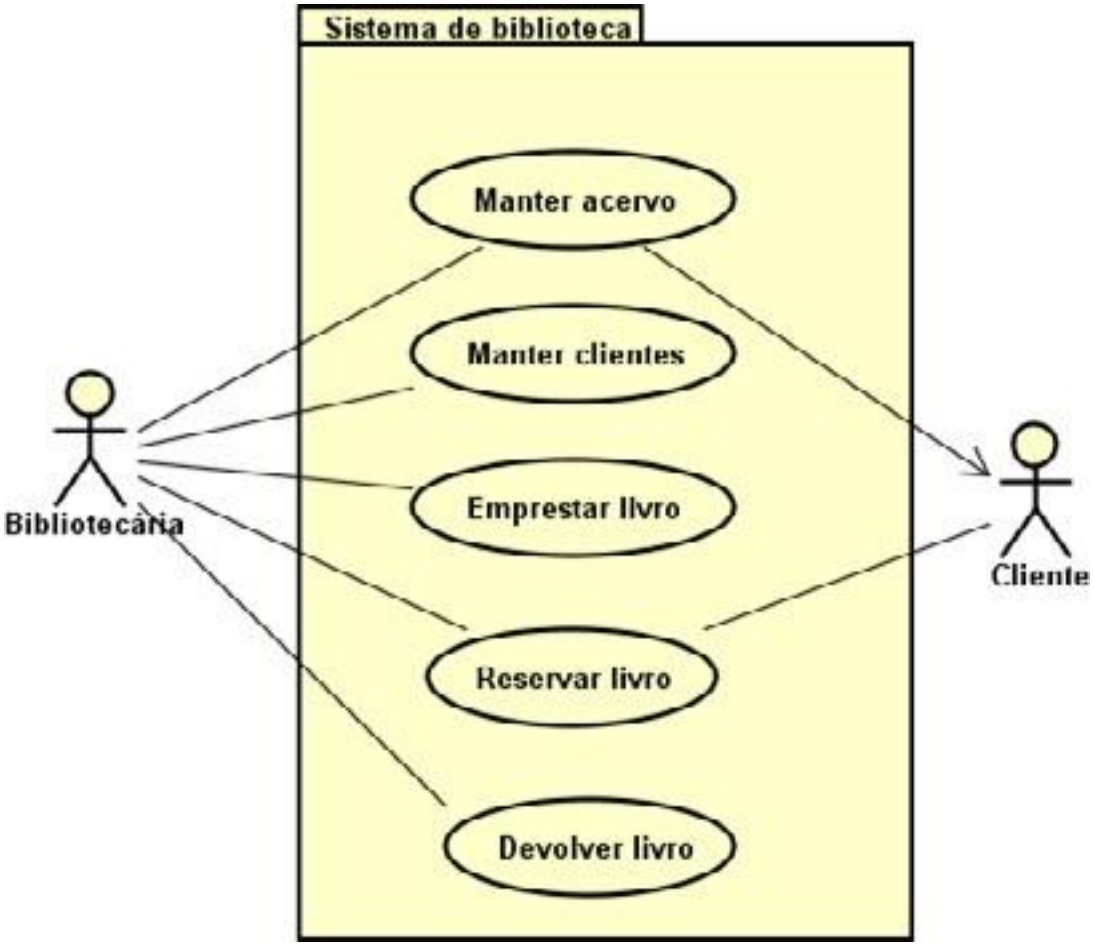


Figura 4 – Caso de uso Sistema de biblioteca
Fonte: CASTRO et al, 2014.

As principais notações estão apresentadas na Tabela 1 (adaptado de CASTRO et al, 2014):

ELEMENTO	SIMBOLOGIA	SIGNIFICADO
ATOR		É uma entidade externa que interage com o caso de uso. Representa um papel perante o sistema que pode ser de desempenhado por um grupo de pessoas, outro sistema, uma organização ou parte dela, um equipamento.
CASO DE USO		Representa uma funcionalidade que atende a um ou mais requisitos do cliente. Como nome, é suficiente usar um verbo no infinitivo + um complemento.
Associação ou (Relacionamento de Comunicação)		Representa um relacionamento entre o ator que utiliza o caso de uso. A ausência de uma seta indica que a comunicação se processa nos dois sentidos. Só use uma seta se a comunicação se processa em um sentido preferencial.

Tabela 1 – Principais notações do diagrama de caso de uso
Fonte: Adaptado Castro et al, 2014.



Fique Ligado!

Atores e os casos de uso são identificados a partir de informações coletadas no levantamento de requisitos (BEZERRA, 2006). Algumas perguntas são interessantes para identificar atores, tais como: que órgãos, empresas ou pessoas (cargos) irão utilizar o sistema? Que outros sistemas irão se comunicar com o software?

Outras perguntas ajudam a identificar os casos de uso, tais como: quais são as necessidades e objetivos de cada ator em relação ao sistema? Que informações o sistema deve produzir?



Além das notações citadas na Tabela 1, os seguintes tipos especiais de relacionamento podem ser usados destacando a dependência entre casos de uso: Generalização / Especialização; Extensão e Inclusão.

TIPO	SIMBOLOGIA	SIGNIFICADO	DIREÇÃO DA SETA
Generalização / Especialização		Depois de especificado um caso de uso, esse trabalho pode ser especializado. O caso de uso especializado (ou derivado) herda as características do caso de uso geral (ou base).	O caso de uso geral (ou base) recebe a ponta da seta.
EXTENSÃO		É a chamada opcional de um caso de uso por outro caso de uso. Ou seja, ao executar o primeiro caso uso, o segundo caso de uso pode ou não ser executado.	A ponta da seta vai para quem chama.
INCLUSÃO		É continuação obrigatória de um caso de uso em outro. Ou seja, ao se execu- tar o primeiro caso de uso, o segundo caso de uso será executado.	A ponta da seta vai na direção da continuação

Tabela 2 - Outras notações de dependência
Fonte: CASTRO et al, 2014.

Vamos entender melhor com um exemplo de cada.

Exemplo de Generalização/Especialização – Vamos imaginar um sistema de um barzinho. Este sistema possui um caso de uso geral que representa a ação de realizar pagamento. O ator cliente precisa realizar pagamento, mas este pagamento pode ser realizado de duas maneiras: com cartão de crédito e com dinheiro. A representação da Figura 5 apresenta os casos de usos especializados herdando as características do geral. A notação mostra o caso de uso geral "Realizar Pagamento" recebendo a ponta da seta.

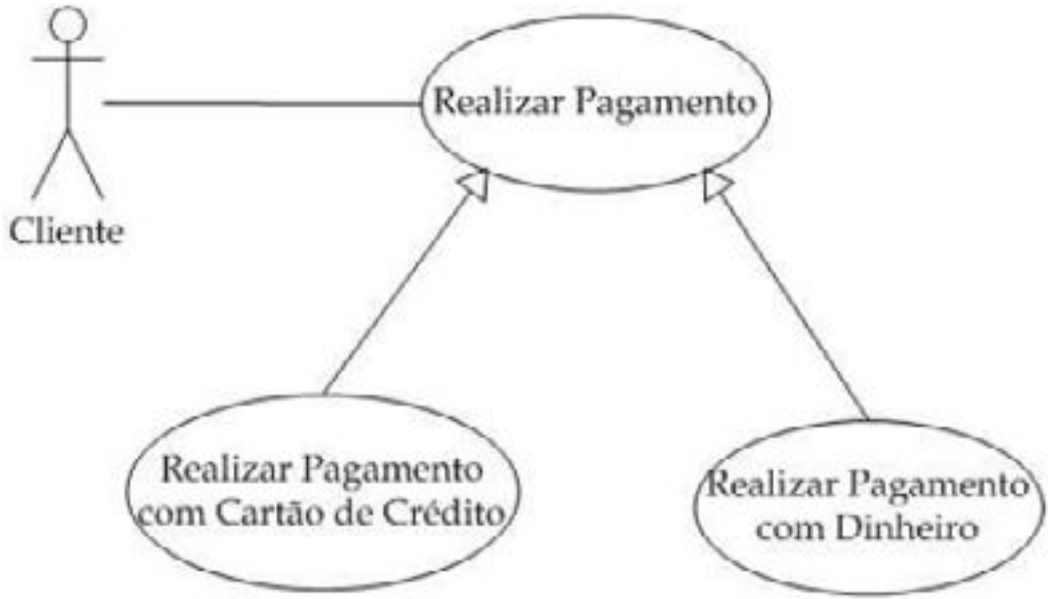


Figura 5 – Exemplo de Especialização
Fonte: BEZERRA, 2006.

Exemplo de Extensão - Vamos imaginar um sistema de edição de uma gráfica. Esse sistema possui como ator o escritor que se relaciona com o caso de uso editar documento. Em alguns momentos pode ser necessário o caso de uso substituir texto ou corrigir ortografia. A extensão nesse caso é representada pela chamada de um caso por outro. Lembrando que a ponta da seta vai para quem chama (Figura 6).

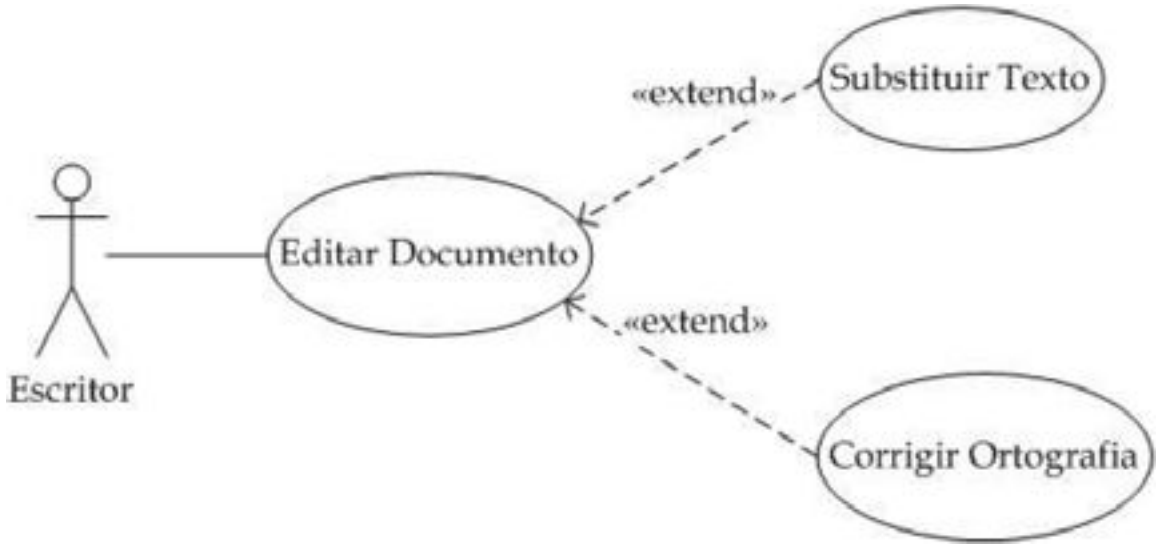


Figura 6 – Exemplo de Extensão
Fonte: BEZERRA, 2006.

Exemplo de Include – Vamos imaginar um sistema bancário, no qual o cliente se relaciona aos casos de uso obter extrato, realizar saque e realizar transferência. Todos esses casos de uso são continuação do caso de uso fornecer identificação. Assim, foi criado o include e a seta vai na direção da continuação (Figura 7).

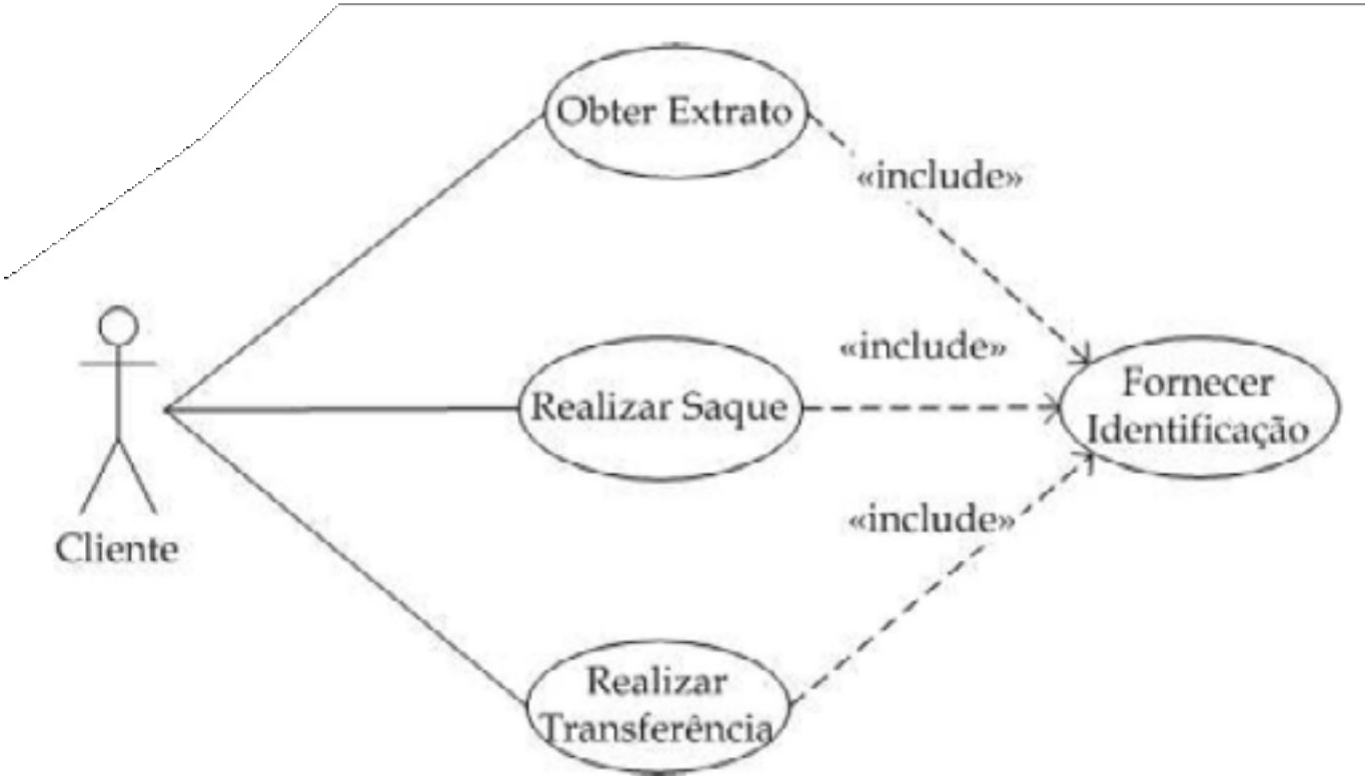


Figura 7: – Exemplo de include
Fonte: BEZERRA, 2006



Fique Ligado!

Uma descrição de cada ator deve ser incluída ao modelo de casos de uso, por exemplo, “Aluno – representa que fazem curso dentro da universidade”.

Com relação à descrição dos casos de Uso, a UML não define um padrão de descrição textual. A equipe e/ou a organização deve definir quais informações são importantes. Mas algumas informações são mais utilizadas: sumário, atores, fluxo principal, fluxos alternativos e referências cruzadas (BEZERRA, 2006).



Saiba mais!

Você sabia que a comunidade Astah disponibiliza ferramenta free para modelagem UML? Acesse o link e baixe a ferramenta. Ela possibilita construir casos de uso utilizando as notações da UML.
http://astah.net/editions/uml-new

E com relação aos Modelos ágeis? Como são tratados os requisitos?

Vamos entender um pouco o SCRUM como um todo?



Saiba mais!

Vamos assistir ao vídeo:
https://www.youtube.com/watch?v=vg1S1WYZa6o

O SCRUM é uma das metodologias ágeis mais utilizadas. Conforme vimos no vídeo, temos também uma visão (pode ser bem sucinta) com uma frase do objetivo do sistema. No levantamento são identificadas as incertezas técnicas, financeiras etc. São mitigadas as incertezas, definidas as estratégias, priorizadas as funcionalidades. E a partir daí são definidas as histórias de usuário. Para escrever uma história do usuário, é importante pensar em três questões, segundo Sutherland (2014):

Quem? Esta pergunta vai indicar o autor da ação;

O quê? Indica a funcionalidade desejada;

Por quê? Indica o valor agregado pela funcionalidade;

Veremos o exemplo dado por Sutherland (2014); vamos pensar em um sistema para um e-commerce de livros. No sistema, irão existir dois tipos de usuários: gerente e cliente. Algumas das histórias:

"Como cliente, eu quero colocar um livro em um carrinho de compras para que eu possa comprá-lo".

"Como gerente de administração de produtos, eu quero poder rastrear as compras de um cliente para que eu possa oferecer livros específicos para ele com base nas compras anteriores".

Podem ainda ser acrescentados critérios de aceite (condições, restrições, regras de negócio) como, por exemplo, restrições de acesso, mensagens de sucesso, erros. Assim, para o exemplo da história do gerente de administração, poderíamos ter como critérios de aceite:

O rastreamento deve considerar as últimas 10 compras.

Agora que já entendemos como especificar requisitos utilizando a abordagem Orientação a Objeto ou Modelos Ágeis, vamos entender como verificar ou validar esses requisitos. Para assegurar que o software adequado e correto seja desenvolvido, é muito importante analisar o que foi especificado. Está correto? Está consistente? No próximo tópico, estudaremos esse processo. Vamos lá?



2.4 - Processo de verificação e validação de requisitos

O objetivo da verificação e validação é identificar se o software possui defeitos e assegurar que o software funcione de acordo com o que foi especificado. Mas qual a diferença entre verificação e validação?

Para Rios e Moreira (2003):

Verificação: engloba as inspeções ou revisões sobre os produtos gerados pelas diversas etapas do processo de desenvolvimento.

Validação: avalia se o sistema atende aos requisitos do projeto (usuário). Os testes unitários, de integração, de sistema e de aceitação podem ser classificados como testes de validação.

E no contexto de requisitos, como fazemos a verificação e a validação dos requisitos representados por casos de usos ou histórias de usuários? Nesse momento, não temos ainda a possibilidade de realizar testes. Não é mesmo?

Na verificação de requisitos, tentamos determinar se estamos construindo o produto da maneira correta. Já na validação de requisitos, verificamos se a especificação de uma fase ou do sistema completo é apropriada e consistente com os requisitos dos stakeholders, ou seja, se é o produto correto.

E como fazemos isso? Por meio de inspeções. Mas, qual o objetivo de uma inspeção?



Fique Ligado!

O principal objetivo da técnica de inspeção é a descoberta antecipada de falhas para garantir que estas não se propaguem para o passo seguinte do processo de construção ou manutenção software.

A inspeção visa encontrar erros, analisando e entendendo o que o documento descreve e checando as propriedades de qualidade. Pode ser realizada de várias formas, a seguir, citamos a inspeção ad hoc e a técnica de inspeção baseada em checklist (CASTRO et al, 2014):

- A inspeção ad hoc não propõe nenhuma técnica formal de leitura; cada leitor (que não é o autor do documento) lê o documento do seu modo, identificando falhas. Por este motivo, essa técnica torna-se dependente da experiência do leitor.

- A técnica de leitura baseada em checklist propõe o uso de uma lista de verificação (checklist) que auxilia a encontrar os defeitos. O uso desse checklist permite ao inspetor seguir uma lista de defeitos com características a serem revisadas.

A seguir, listamos alguns dos defeitos mais citados com relação à verificação:

Defeito de ambiguidade - um requisito tem várias interpretações devido aos diferentes termos utilizados para uma mesma característica.

Clareza - os requisitos foram escritos em linguagem não-técnica permitindo um bom grau de entendimento? Foi elaborado um glossário de termos para auxiliar no entendimento?

Completeza - Todos os termos necessários foram definidos? Algum requisito deveria ter sido especificado em mais detalhes? Algum requisito deveria ter sido especificado em menos detalhes?

Consistência - Existe algum requisito que descreve duas ou mais ações que conflitam logicamente? Existe algum requisito impossível de ser implementado? Cada um dos requisitos é relevante para a solução do problema?



E com relação à validação? Vamos ver alguns defeitos?

Defeito de omissão - informações necessárias ao sistema são omitidas, faltam funções requeridas pelo cliente.

Defeito de fato incorreto - há informações nos artefatos do sistema que são contraditórias com o domínio da aplicação.



Fique Ligado!

No modelo ágil, a verificação e a validação, normalmente, são realizadas nas atividades de avaliação da documentação e da criação da documentação de testes.

Na atividade de avaliação da documentação, os profissionais de teste definem os cenários, fluxos e escrevem os casos de testes. Essa equipe deve ler, entender e analisar as histórias de usuário, validar esses documentos, participar do refinamento e eliminar possíveis brechas. Nessa função, deve-se ter sempre em mente a visão do usuário e quais são os possíveis casos reais que ele poderá executar.

Na criação da documentação de testes com os artefatos revisados, melhorados e atualizados, deve-se dar início à escrita de casos de testes. A documentação de testes deve ser simples, direta e clara na medida do possível e deve descrever o passo a passo para validação dos critérios de aceite. Fazem parte dessa atividade a criação, a manutenção e a atualização dos documentos, além da criação de scripts.



Saiba mais!

Vamos assistir ao vídeo abaixo? Ele descreve muito bem o processo de requisitos.

https://www.youtube.com/watch?v=C8heVblhmKo



Bora rever!

Caro estudante, nesta unidade você estudou a importância da modelagem e suas ferramentas. Entendeu os principais conceitos relacionados à UML, que é uma linguagem utilizada para modelar sistemas. Também aprendeu as técnicas de levantamento de requisitos e sua importância. Levantar requisitos não é uma tarefa fácil e as técnicas podem ajudar muito nosso trabalho. Na construção de um produto de software, estamos transformando uma abstração, que são as ideias do cliente, em um software. Esse processo envolve muita comunicação, ruídos, expectativas, necessidades etc. Qualquer ferramenta que ajude a elicitarmos com mais eficiência só pode agregar valor ao nosso trabalho.

Estudamos também técnicas de especificação de requisitos para a abordagem orientada a objeto e para os Métodos ágeis. Conhecemos o documento de visão, as principais notações do diagrama de caso de uso e o que deve conter a descrição do caso de uso. Com relação aos Modelos ágeis, entendemos como elaborar Histórias de usuário. Ao final, vimos o processo de Verificação e Validação de requisitos, sua importância e como pode ser implementado em uma organização para melhorar a qualidade das especificações. Espero que tenha gostado e assimilado estes conhecimentos de forma a aplicá-los na sua vida profissional.

Nas próximas unidades, detalharemos melhor esses conceitos e aprenderemos outros.

Vamos lá?



Questões de autoaprendizagem

1.Você faz parte de uma equipe de desenvolvimento de um sistema que possui vários stakeholders. Cada um desses stakeholders é responsável por uma parte do sistema e é necessário reuni-los para discutir as fronteiras e obter consenso sobre os requisitos a serem atendidos e sua ordem de prioridade. Qual a técnica de levantamento de requisitos seria mais adequada?

a.( ) entrevista
b.( ) análise de documentos
c.( ) workshop
d.( ) prototipação
e.( ) Nenhuma das alternativas anteriores

Solução – C. O workshop possibilita o envolvimento de todos os interessados, agiliza o entendimento do projeto do Sistema, facilita a obtenção de consenso, pois agrega as partes interessadas.

2. Um diagrama de caso de uso é um diagrama comportamental da UML que evidencia a visão externa do sistema a ser desenvolvido. Ele mostra quais são as funcionalidades desse sistema e quem interage com elas. São notações do diagrama de caso de uso: ator, caso de uso, associação, Generalização/especialização, Extensão, Inclusão. Acerca dessas asserções, assinale a opção correta.

a.( ) As duas asserções são proposições verdadeiras; a segunda é uma complementação da primeira.

b.( ) As duas asserções são proposições verdadeiras, mas a segunda não é uma complementação da primeira.

c.( ) A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.

d.( ) A primeira asserção é uma proposição falsa, e a segunda, uma proposição verdadeira.

e. ( ) Tanto a primeira quanto a segunda asserções são proposições falsas.

Solução – A. As duas proposições estão corretas, e a segunda é complementação da primeira. Um diagrama de caso de uso é um diagrama comportamental da UML que evidencia a visão externa do sistema a ser desenvolvido. Ele mostra quais são as funcionalidades desse sistema e quem interage com elas. São notações do diagrama de caso de uso: ator, caso de uso, associação, Generalização/especialização, Extensão, Inclusão.



Unidade 3

Caracterização e aplicação de metodologias e ferramentas de modelagem de sistemas orientados a objetos

3.1 Conceitos de modelagem de sistemas, modelos e suas perspectivas

Neste tópico, estudaremos os conceitos de modelagem de sistemas, modelos e suas perspectivas com relação a modelagem de sistemas orientados a objetos.

Já vimos anteriormente o termo Modelagem. Lembra? No tópico II, falamos de Modelagem de requisitos. Agora vamos entender um escopo maior que é a modelagem de sistemas.

O que é modelagem de sistemas?

Modelagem de sistemas é a atividade de construção de modelos que expliquem /ilustrem a forma de funcionamento de um software.

Modelar é desenhar ou projetar o software antes da codificação. Segundo Bezerra (2006), existem muitas razões para se modelar sistemas, tais como: facilitar a comunicação entre as pessoas envolvidas, prever o comportamento futuro do sistema, gerenciar a complexidade, reduzir os custos do desenvolvimento.

E modelagem de sistemas orientados a objetos?

O termo orientação a objetos relaciona-se a organizar o mundo real como um conjunto de objetos que incorporam:

estrutura de dados (propriedades ou atributos) e;

um conjunto de operações ou métodos (que manipulam esses dados).

Para entender melhor, veja o exemplo abaixo e a Figura 1. Todos os objetos têm um nome ou identificador, suas propriedades e operações.

Objeto: Pessoa

Propriedades ou atributos: nome, data de nascimento, peso, altura

Operações: andar, falar, dormir

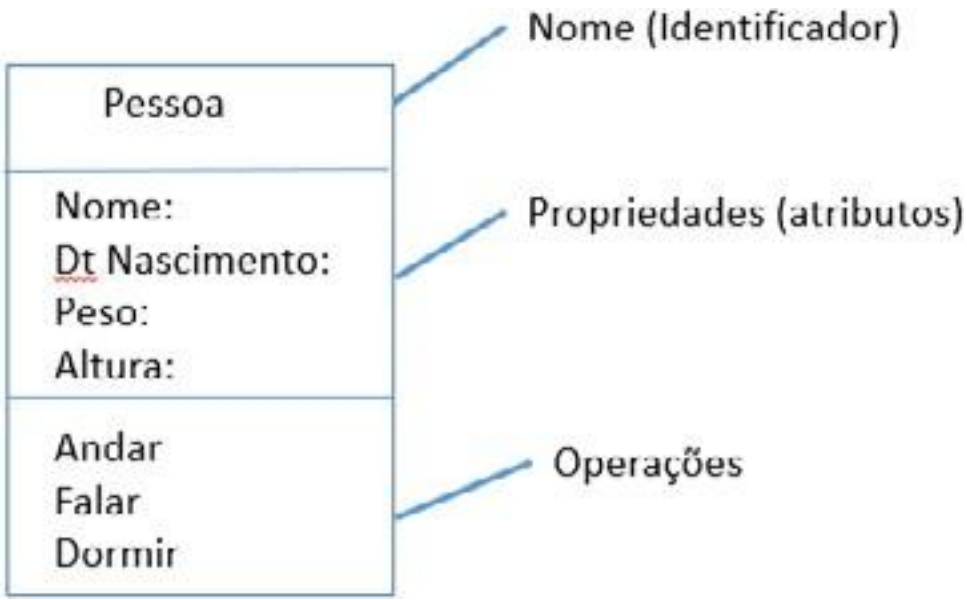


Figura 1 – Representação de Pessoa na orientação a objetos
Fonte: elaborado pela autora.



Questões de autoaprendizagem

E como seria um objeto veículo? Quais as propriedades ou atributos? Quais as operações ou métodos?

Dica:

Os nomes dos objetos geralmente são substantivos no singular, tais como: cliente, conta-corrente, pessoa, etc.

As propriedades ou atributos são substantivos, exemplo: cor, tamanho, peso, idade, número, etc.

Já as operações ou métodos, usualmente, são verbos, como: acelerar, validar, verificar, calcular, etc.

Então, a modelagem de sistemas com a abordagem orientada a objetos consiste na construção de objetos que colaboram para construir sistemas mais complexos.

São alguns dos princípios da Orientação a Objeto, segundo Bezerra (2006):

qualquer coisa é um objeto;

objetos realizam tarefas por meio da requisição de serviços a outros objetos;

cada objeto pertence a uma determinada classe. Uma classe agrupa objetos similares.

Mas o que é uma classe?

Uma classe descreve um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos (BEZERRA, 2006).

Vamos esclarecer: as classes são as partes mais importantes do modelo orientado a objetos. São utilizadas para capturar e apresentar o vocabulário do sistema que está em desenvolvimento.

Retomando o exercício proposto antes sobre o objeto veículo. Ao fazermos esse exercício, provavelmente, construiríamos algo similar a parte da Figura 2, a seguir:

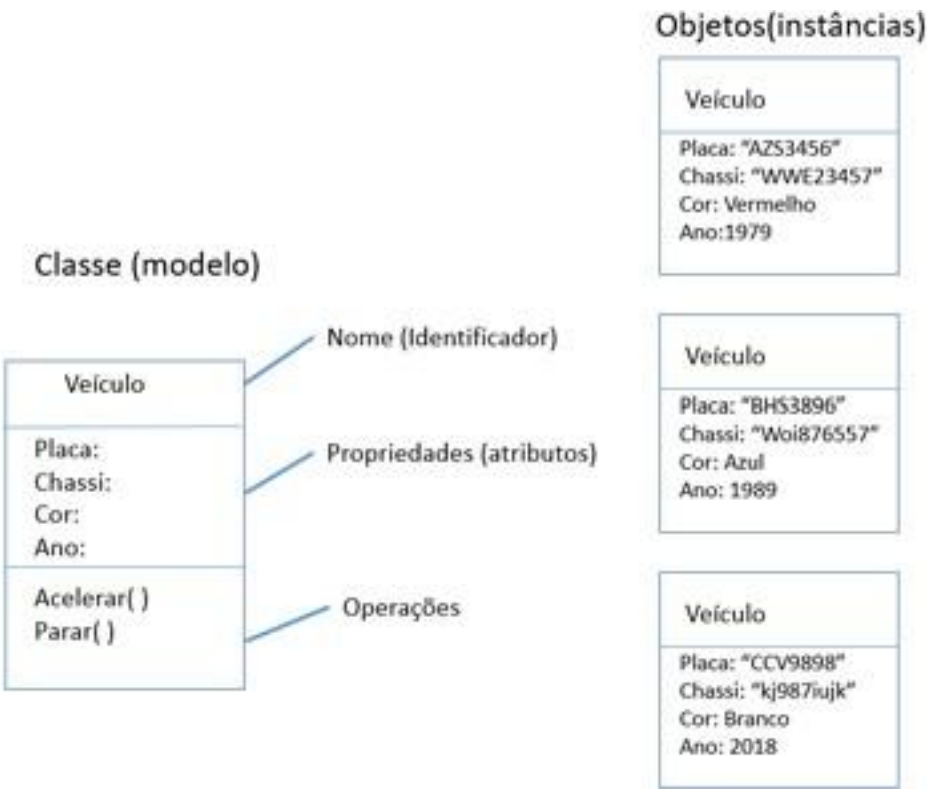


Figura 2 – Exemplo de Classe e objetos.
Fonte: elaborado pela autora.

O que definimos descreve um conjunto de objetos que compartilham os mesmos atributos, operações/métodos, etc. Todos os veículos têm placa, chassi, cor, ano. Todos os veículos têm operações de acelerar, parar. Então, definimos uma classe chamada veículo. Essa classe pode ter 1 ou mais objetos também chamados de instâncias. Elas utilizam os mesmos atributos e métodos, ou seja, todas as instâncias são veículos que possuem os mesmos atributos: placa, chassi, cor, ano. Está claro?

Vamos fazer uma analogia com a Engenharia Civil. Imagine um prédio de apartamentos. Todos os apartamentos são, rigorosamente, iguais e baseados em uma planta estabelecida (número de dormitórios, portas, fiação etc.). A planta dos apartamentos é a classe. Os apartamentos são objetos baseados naquela classe (MEDEIROS, 2004).

Ao criar uma classe, você está definindo que todos os objetos dessa classe têm os mesmos atributos e as mesmas operações/métodos.



Como definir os atributos e métodos?

Somente atributos que são de interesse do sistema devem ser descritos na classe. Isso quer dizer que se para o seu sistema a cor do veículo não é importante, a classe veículo não terá esse atributo. Claro?

E com relação aos métodos? Como criar os métodos?

Método é a implementação de uma operação. Os métodos são utilizados para realizar qualquer operação, por exemplo: cálculo salário, acender um aparelho, mostrar um saldo, manter um livro (incluir, alterar, excluir).

A Figura 3 apresenta um exemplo da classe livro.



Figura 3 – Exemplo de atributos e métodos para a classe livro
Fonte: elaborado pela autora.

Vamos entender melhor o conceito de Orientação a objetos. Assista ao vídeo, a seguir, tomando nota de todos os pontos relevantes ao que estamos discutindo aqui.



Saiba mais!

https://www.youtube.com/watch?v=7nmJ2oGRrbc



Saiba mais!

Você se lembra de que no módulo II, para modelar casos de uso, sugerimos a utilização do software Astah? Para as próximas figuras, estaremos utilizando esse software. Ok?

Você pode baixá-lo no link: http://astah.net/editions/uml-new

Para Bezerra (2006), a abordagem de orientação a objeto engloba, além de classes e objetos, os conceitos de herança, polimorfismo, encapsulamento e composição. Estudaremos esses conceitos a seguir.

Herança/Generalização

Mecanismo baseado em objetos que permite que as classes compartilhem atributos e operações baseados em um relacionamento, geralmente, generalização (RUMBAUGH et al, 1991).

Em uma Herança, uma classe “derivada” herda a estrutura de atributos e métodos de sua classe “base”, veja a Figura 4. A classe base é funcionário e a classe derivada é motorista. A Figura 4, a seguir, mostra a classe funcionário com 3 atributos. Ela proporciona os atributos e métodos para a classe derivada motorista.

Mas a classe derivada pode adicionar novos métodos, aumentar a estrutura de dados ou redefinir a implementação de métodos já existentes. No exemplo, a classe motorista adicionou mais um atributo. Qual?

Quais são as vantagens da herança? Você consegue identificar? Se tivéssemos mais uma classe, secretária, com atributos “Fluência língua inglesa” e “Fluência língua espanhola”, ela poderia ser outra classe derivada? A resposta é sim. Ela herdaria também a estrutura e os métodos da classe funcionário. Veja a Figura 4. Ficou claro?

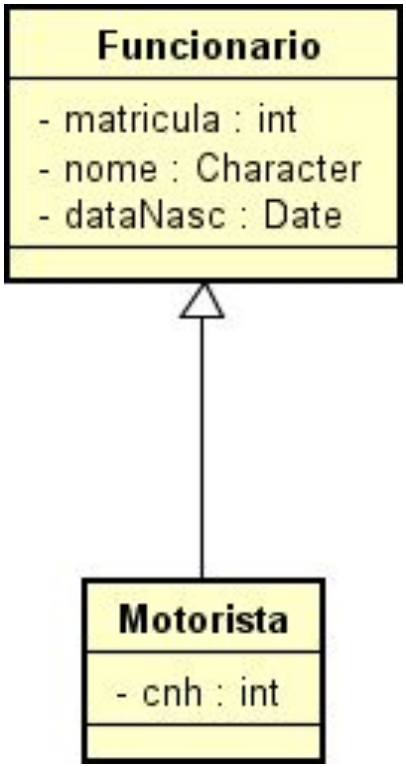


Figura 4 - Exemplo de herança
Fonte: elaborado pela autora.



Saiba mais!

Veja outros vídeos que abordam sobre herança.

https://www.youtube.com/watch?v=_PZldwo0vVo
https://www.youtube.com/watch?v=He887D2WGVw

Polimorfismo

Polimorfismo é uma operação que pode assumir múltiplas formas; a propriedade segundo o qual uma operação ou método pode comportar-se diferentemente em classes diferentes (RUMBAUGH et al, 1991).

O polimorfismo possibilita o reuso. Vamos ver o exemplo anterior, mas agora na Figura 6. O método CalSalario é herdado da superclasse (Funcionário), entretanto, para motorista e para a secretária, ele tem uma implementação diferente. Por exemplo:

Para apurar o salário do motorista = (salário + adicional direção (%20 sobre horas dirigidas))

Para apurar o salário da secretária = (salário + adicional fluência (+%10 sobre salário para cada língua com fluência))

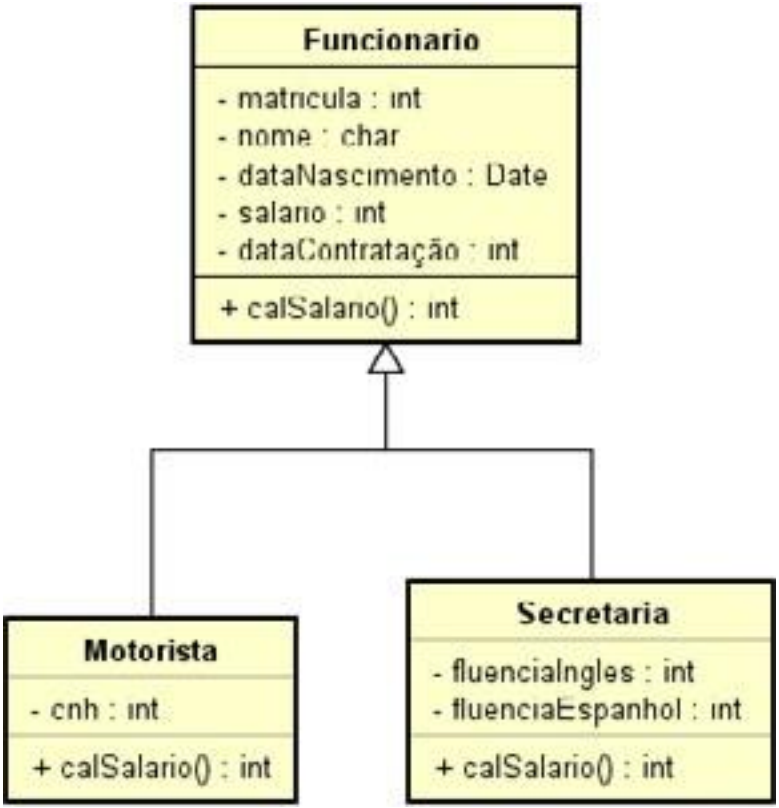


Figura 5 - Exemplo de polimorfismo.
Fonte: elaborado pela autora.



Saiba mais!

Vamos para um vídeo sobre polimorfismo?

https://www.youtube.com/watch?v=9-3-RMEMcq4

Encapsulamento

"O encapsulamento refere-se à visibilidade de atributos, métodos e classes e ao empacotamento das classes" (MEDEIROS, 2004, pág. 9). O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. Protege os dados manipulados dentro da classe e determina onde esta classe pode ser manipulada.



Por exemplo: você entra no carro, dirige e pisa no freio. O que acontece com o mecanismo do freio e com os pneus, você não tem a menor ideia. Só espera que o carro pare, não é? Para o motorista não é necessário entender o mecanismo interno do veículo, contudo, quem projetou e construiu o veículo se preocupou com isso e implementou este mecanismo de forma correta. Isso é o que chamamos de encapsulamento e ocultamento de informações.

Na orientação a objeto, para possibilitar o encapsulamento, define-se o nível de acesso. O encapsulamento é dividido em dois níveis:

Nível de classe: quando se determina o acesso relacionado a uma classe inteira: public ou Package-Private (padrão);

Nível de membro: quando se determina o acesso de atributos ou métodos de uma classe: public, private, protected ou Package-Private (padrão).



Saiba mais!

Imagine uma quadra de esportes... Tem aquelas que ficam localizadas em parques e podem ser usadas por todos (*public*) – todos podem utilizar seus atributos e usar de acordo com seu desejo. Tem aquelas quadras que ficam em clubes e só podem ser usadas por seus membros (*protected*) – somente as classes do mesmo pacote têm acesso. Por fim, tem aquelas que ficam em casas e somente o dono da casa pode utilizar (*private*) – somente a classe em que está o atributo tem acesso.

No encapsulamento, um objeto que precise da colaboração de outro objeto para realizar uma operação, simplesmente, envia uma mensagem a este último (BEZERRA, 2006). Entretanto, o remetente da mensagem precisa saber quais operações o receptor pode realizar ou fornecer. "Na terminologia da orientação a objetos, diz-se que o objeto possui uma interface. Em termos simples, a interface do objeto corresponde ao que ele conhece e ao que sabe fazer, sem descrever como conhece ou faz" (Bezerra, 2006, p. 9). Ou seja, a interface é o contrato entre a classe e o mundo exterior.

Vamos fazer uma analogia?

Quando o usuário faz uma escolha na tela do computador, ele está utilizando uma interface gráfica. Correto? O que possibilita ao mundo externo colaborar com uma classe é sua interface.



Saiba mais!

Vamos complementar esse assunto com um vídeo?

https://www.youtube.com/watch?v=1wYRGFXpVlg

Composição

Quando analisamos as coisas do mundo real, podemos visualizar o conceito de composição. Por exemplo: uma televisão é composta por um painel de controle e uma tela. Esse material que você está lendo é composto por páginas. Na orientação a objeto,os objetos também podem ser compostos por outros objetos. Ou seja, podemos criar objetos a partir da reunião de outros objetos (BEZERRA, 2006).

Uma nota fiscal, por exemplo, normalmente é composta de um cabeçalho com os dados de data, CNPJ ou CPF de quem está comprando, número da nota fiscal e, também, tem uma relação dos produtos, seus preços unitários, quantidade por produto e valor total. Vamos pensar em uma classe nota fiscal com os atributos número, data e CNPJ. Também podemos ter uma classe Item NotaFiscal com os atributos número, quantidade do produto etc. É claro que deve existir, pelo menos, um item em uma nota fiscal para que ela exista. Concorda? Não existe nota fiscal sem pelo menos um produto vendido.

Esse é um exemplo de composição:NotaFiscal é composta de ItemNotaFiscal.

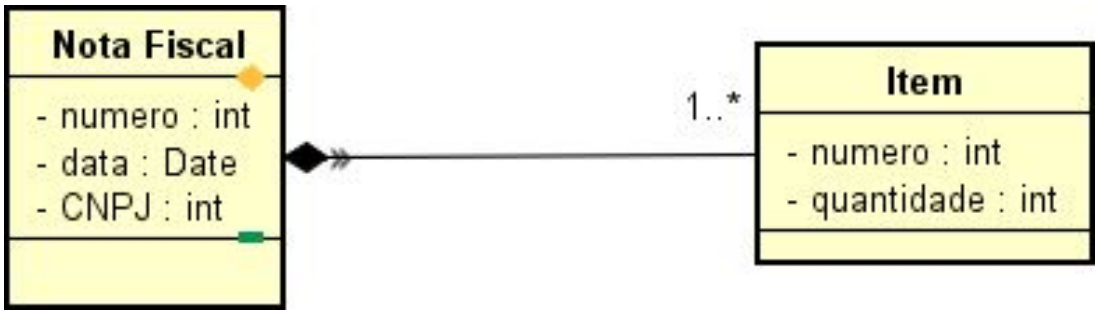


Figura 6: Exemplo de composição.
Fonte: elaborado pela autora.

Agora, estudaremos alguns modelos de processo de desenvolvimento de software que utilizam os conceitos da orientação a objetos. Vamos lá?



3.2 Etapas do processo de desenvolvimento de software e metodologias

Neste tópico, detalharemos mais os processos de desenvolvimento de software. Você deve se lembrar, pois citamos o processo de desenvolvimento de software na Unidade I.



Fique Ligado!

A literatura apresenta uma série de modelos de processo de desenvolvimento de sistema, tais como: Cascata, Prototipação, Processo Unificado (Orientação a objetos), Métodos Ágeis (eXtremeProgramming - XP, SCRUM) e outros.



Fique Ligado!

Uma observação importante: existem alguns modelos prontos os quais já foram escolhidos os melhores métodos, procedimentos e ferramentas. O Processo unificado (Orientação a objetos) e os Métodos Ágeis (SCRUM, XP e outros) são exemplos desse tipo de modelos (CASTRO et al, 2014).

Não existe o modelo ideal, cada um tem vantagens e desvantagens. O conhecimento dos modelos é importante, pois no mercado algumas organizações trabalham com Métodos Ágeis, outras com Processo Unificado, outras com um modelo próprio, adaptado de algum outro. Assim, conhecer os principais modelos nos dará uma visão abrangente das variadas formas que podemos utilizar para construir um produto de software. Nesse tópico, estudaremos o modelo unificado e os modelos ágeis. Por quê?

Porque o objetivo desta unidade é estudar modelos com foco em orientação a objetos. Porque modelos como o Cascata e a Prototipação, além de não terem foco na orientação a objeto, são modelos mais antigos, apesar de algumas empresas ainda utilizarem esses modelos para construir software.

Processo Unificado

O Processo Unificado (PU) surgiu como um processo para o desenvolvimento de software visando à construção de sistemas orientados a objetos. Os precursores desse modelo foram os trabalhos de Rumbaugh et al (1991), Jacobson et al (1992) e Booch (1994).



Saiba mais!

O modelo Processo Unificado tem como principais características: ser baseado em componentes que realizam interfaces, utilizar a UML, ser dirigido a casos de uso, ter o desenvolvimento iterativo e incremental e ser focado em arquitetura.

Já sabemos o que é UML e casos de uso (vimos os conceitos na Unidade II). Mas o que é ter o desenvolvimento iterativo e incremental? O que é "ser focado em arquitetura?"

Desenvolvimento iterativo é uma estratégia de planejamento em que se definem as entregas das partes do sistema.

Desenvolvimento Incremental é uma estratégia de planejamento em estágios em que várias partes do sistema são desenvolvidas em paralelo e integradas quando completas. Nesse modelo, o sistema é dividido em subsistemas por funcionalidades. Essas versões são definidas começando com um pequeno subsistema funcional, adicionando-se mais funcionalidades a cada versão (CASTRO et al, 2014).

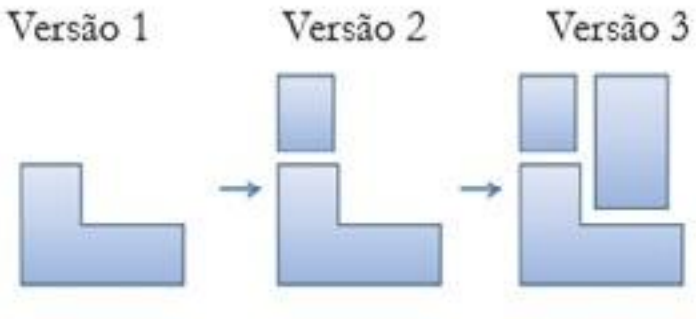


Figura 7 – Exemplo de desenvolvimento incremental.
Fonte: PFLEEGER, 2004

Ser focado em arquitetura engloba ter a visão do software como um todo. Vamos para um exemplo. Lembra do sistema de biblioteca da Unidade I?

Quando escrevemos sobre as questões da biblioteca, as tecnologias e ambientes computacionais que serão utilizados pelo novo software, estamos falando de sua arquitetura. Mais adiante retomaremos esse assunto.



Fique Ligado!

Vamos procurar na internet, no Google. Procure pelo termo “processo unificado de desenvolvimento de software”. Você vai encontrar várias referências ao RUP, não é? Mas o que é RUP?

O RUP – RationalUnifiedProcess é um refinamento do Processo Unificado. Quando falamos do RUP, é necessário identificar três dimensões (CASTRO et al, 2014):

RUP: como um processo de desenvolvimento de sistemas;

RUP: como o produto comercializado pela IBM/Rational, que contém uma Base de Conhecimento do RUP (disponível no site da IBM), chamado de IBM RationalMethod Composer;

RUP: conjunto de ferramentas comercializadas pela IBM/Rational, empregadas para apoiar o uso produtivo do RUP.

O RUP trabalha com os conceitos de disciplinas e fases, conforme a Figura 8. Na fase de Iniciação, o foco está no entendimento do negócio, no escopo (análise do problema e criação de uma “visão” da solução). Também são identificadas estimativas preliminares de prazo, custo e riscos do projeto. Na fase de Elaboração, refinam-se os requisitos, define-se a arquitetura e constrói-se um protótipo (se necessário).

A Construção tem foco na implementação do sistema, onde a maior parte do código é desenvolvido. Nesta fase, o projeto é totalmente desenvolvido e a transição corresponde às atividades de teste, treinamento e implantação do sistema em ambiente de produção.

Em cada fase, o projeto percorre várias iterações. Cada iteração é uma sequência de atividades planejadas e avaliadas no final. Cada iteração gera um executável e se baseia na iteração anterior. Ou seja, o projeto no RUP é desenvolvido de modo "iterativo e incremental".

A Figura 8 mostra o modelo do RUP; cada cor representa o tempo gasto ou o esforço para aquela disciplina naquela fase.

Vamos ver se você entendeu? Identifique quais disciplinas têm maior esforço na fase de iniciação. Conseguiu? São a Modelagem de Negócios e Requisitos. Lembre-se de que, nesta fase, estamos identificando o escopo, os problemas, os objetivos, os requisitos do sistema. Tem sentido estarmos gastando mais esforço nessas disciplinas.

Quais disciplinas não têm nenhum esforço gasto nessa etapa? É a implantação não é? É óbvio que na iniciação não tem esforço para implantação.

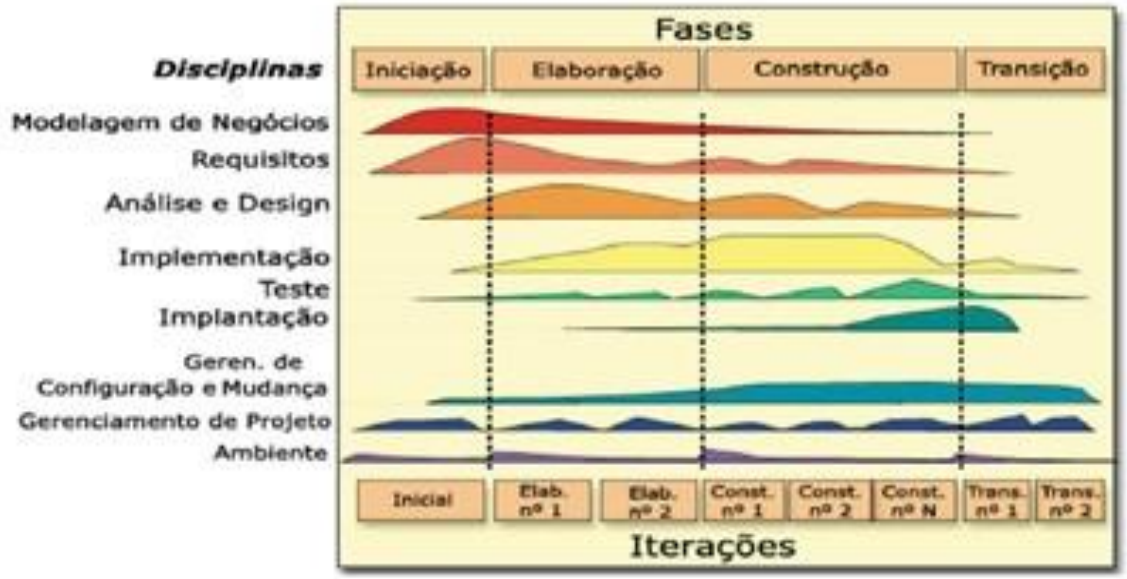


Figura 8 – Modelo RUP.
Fonte: Rational, 2002.

Além das fases, o RUP trabalha com o conceito de disciplinas. Uma disciplina consiste em conjunto de atividades sequenciadas para produzir um artefato. Em cada iteração, a equipe gasta quanto tempo for apropriado para cada disciplina, conforme Figura 8.

Métodos Ágeis

A expressão "Metodologias Ágeis" ou Métodos Ágeis tornou-se conhecida em 2001, quando especialistas em processos de desenvolvimento (XP, Scrum e outros) estabeleceram princípios e características comuns destes métodos. Assim, foi criada a "Aliança Ágil" e definiu-se os 4 valores do manifesto ágil:

Indivíduos e interações mais que processos e ferramentas. Trabalhar no software mais que documentação abrangente. Colaboração do cliente mais que negociação contratual.

Responder às mudanças mais que seguir um plano

Assim, o grupo das Metodologias Ágeis engloba vários métodos, modelos ou frameworks, tais como: ExtremingProgramming – XP, SCRUM, LeanDevelopment e outros. A seguir, apresentaremos os principais conceitos do SCRUM.

Scrum

O SCRUM é um framework iterativo que possibilita aos desenvolvedores tratar e resolver problemas complexos e adaptativos e entregar produtos com o mais alto valor possível. É um framework dentro do qual você pode empregar vários processos ou técnicas.

Vamos ver um vídeo sobre o SCRUM?

https://www.youtube.com/watch?v=7lhnYbmovb4

O Scrum é baseado em papéis (SCHWABER ET SUTHERLAND, 2017):

-ProductOwner (PO) que é dono do produto, responsável por maximizar o valor do produto;

-Scrum Master (SM) que é o responsável por promover e suportar o Scrum como definido no Guia Scrum;

-Time de Desenvolvimento consiste de profissionais que realizam o trabalho de entregar um incremento potencialmente liberável do produto "Pronto" ao final de cada Sprint.

O Scrum possui os seguintes eventos (SCHWABER ET SUTHERLAND, 2017):

-Sprint - é um time-boxed (evento) de um mês ou menos, durante o qual um incremento do produto é criado e liberado.

-Planejamento da Sprint - um time-boxed (evento) com no máximo oito horas para uma Sprint de um mês de duração. O planejamento define a meta da Sprint e responde às seguintes questões:

O que pode ser entregue como resultado do incremento da próxima Sprint?

Como o trabalho necessário para entregar o incremento será realizado?

-Reunião Diária - é um time-boxed (evento) de 15 minutos para o Time de Desenvolvimento. A reunião diária é realizada todos os dias da Sprint. Nela, o Time de Desenvolvimento planeja o trabalho para as próximas 24 horas.

-Revisão da Sprint - é realizada no final da Sprint para inspecionar o incremento do produto e adaptar o Backlog do produto, se necessário.

-Retrospectiva da Sprint - é uma oportunidade para o Time Scrum inspecionar a si próprio e criar um plano para melhorias a serem aplicadas na próxima Sprint.

São artefatos do Scrum (SCHWABER ET SUTHERLAND, 2017):

-Backlog do Produto - é uma lista ordenada de tudo que é conhecido ser necessário no produto. É a única origem dos requisitos para qualquer mudança a ser feita no produto.

-Backlog da Sprint - é um subconjunto de itens do Backlog do Produto selecionados para aquela Sprint, juntamente com o plano para entregar o incremento do produto e atingir o objetivo da Sprint.

-Incremento - O incremento é a soma de todos os itens do Backlog do Produto completados durante a Sprint e o valor dos incrementos de todas as Sprints anteriores.

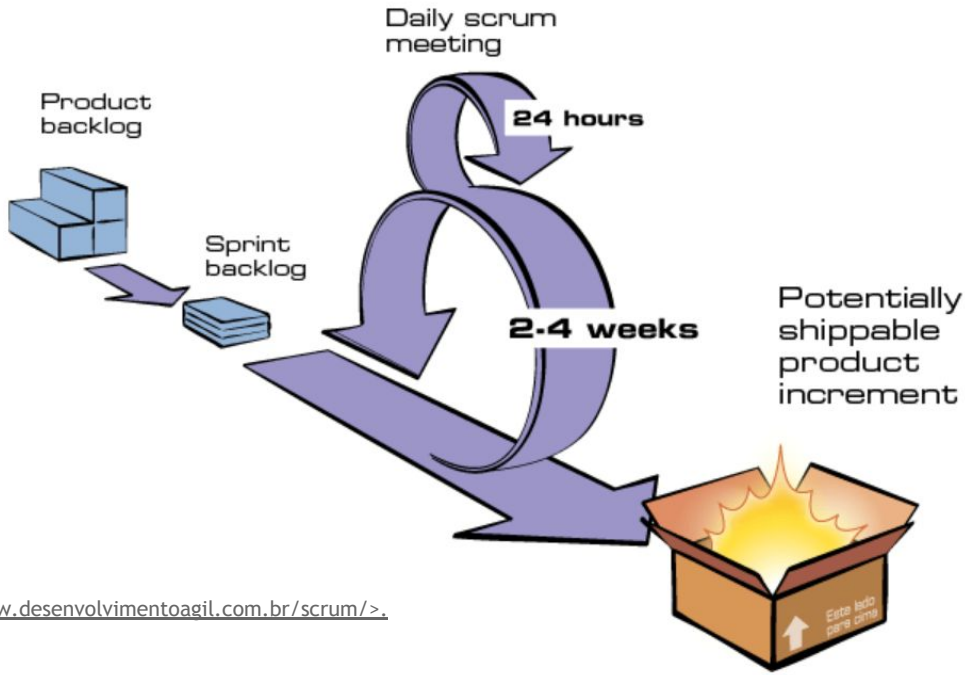


Figura 9 - Processo SCRUM
Imagem disponível em: <https://www.desenvolvimentoagil.com.br/scrum/>.

A Figura 9 mostra o processo do SCRUM. Normalmente, o SCRUM trabalha com histórias de usuário que devem ser curtas e responder aquelas três questões citadas na Unidade II, lembra?



Fique Ligado!

Vamos ver o vídeo sobre histórias e modelos ágeis?

https://www.youtube.com/watch?v=3Smbhnmue7Y



Saiba mais!

Você quer saber mais sobre o SCRUM? Acesse o link: https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-Portuguese-Brazilian.pdf

Bora rever!

Caro estudante, nesta unidade você estudou as metodologias e ferramentas de modelagem de sistemas orientados a objetos. Entendeu os principais conceitos relacionados à Orientação a objetos. Entendemos o que é um objeto, o que são classes, propriedades e operações. Entendemos, também, os principais conceitos relacionados a orientação a objetos que são: herança, polimorfismo, encapsulamento e composição. Estes conceitos serão revisados no módulo 4 quando estaremos detalhando melhor a linguagem UML.

Estudamos também as etapas do processo de desenvolvimento de software e metodologias. Conhecemos o processo unificado, discutimos o porquê de ser chamado algumas vezes de RUP. Ele é um modelo interativo e incremental.

Vimos, também, os modelos ágeis e detalhamos o SCRUM. Todos esses processos trabalham com a abordagem orientada a objetos. Espero que tenha gostado e assimilado estes conhecimentos de forma a aplicá-los na sua vida profissional.

Na próxima unidade, detalharemos melhor esses conceitos e aprenderemos outros. Vamos lá?



Figura 10. Tirinha
Fonte:
Imagem disponível em:
https://vidadeprogramador.com.br/2011/11/16/orientado-a-objetos/



Questões de autoaprendizagem

1. Qualquer coisa é um objeto e cada objeto pertence a determinada classe. Uma classe descreve um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos.

Acerca dessas asserções, assinale a opção correta.

a. As duas asserções são proposições verdadeiras, e a segunda é uma complementação da primeira.

b. As duas asserções são proposições verdadeiras, mas a segunda não é uma complementação da primeira.

c. A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.

d. A primeira asserção é uma proposição falsa, e a segunda, uma proposição verdadeira.

e. Tanto a primeira quanto a segunda asserções são proposições falsas.

Solução – A. As duas proposições estão corretas, e a segunda é complementação da primeira. Qualquer coisa é um objeto e cada objeto pertence a uma determinada classe. Uma classe descreve um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos.

2. Observe a figura a seguir e identifique a questão correta.

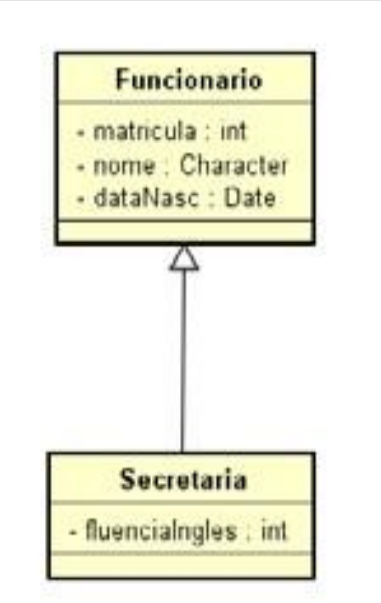


Figura 11
Fonte: elaborado pela autora.

a. É um exemplo de herança na qual a classe base é secretária e a classe derivada é funcionário. A classe secretária proporciona os atributos e métodos para a classe derivada funcionário.

b. É um exemplo de herança na qual classe base é funcionário e a classe derivada é secretária. A classe funcionário proporciona os atributos e métodos para a classe derivada secretária.

c. É um exemplo de polimorfismo, pois possibilita que o método se comporte diferente em diferentes classes.

d. É um exemplo de composição, pois mostra que os objetos podem ser compostos por outros objetos.

Solução - B. É um exemplo de herança na qual a classe base é funcionário e a classe derivada é secretária. A classe funcionário proporciona os atributos e métodos para a classe derivada secretária.

Unidade 4

UML e seus diagramas e Introdução a design patterns



4.1 Linguagem de Modelagem Unificada (UML)

Nesta Unidade, estudaremos a UML e seus diagramas. Também veremos uma introdução a design patterns ou padrões de projeto. Já vimos na Unidade II, a descrição da notação UML, lembram?

UML – Linguagem de Modelagem Unificada (*Unified Modeling Language*) é a linguagem utilizada para modelar os sistemas. A UML auxilia os desenvolvedores a definir as características do software, seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos.

A UML já está na versão 2.0. Ela define 13 tipos de diagramas, divididos em 3 categorias: diagramas estruturais, diagramas comportamentais e diagramas de interação. Cada categoria contém os seus diagramas relacionados conforme apresenta a Figura 1 (OMG, SD).

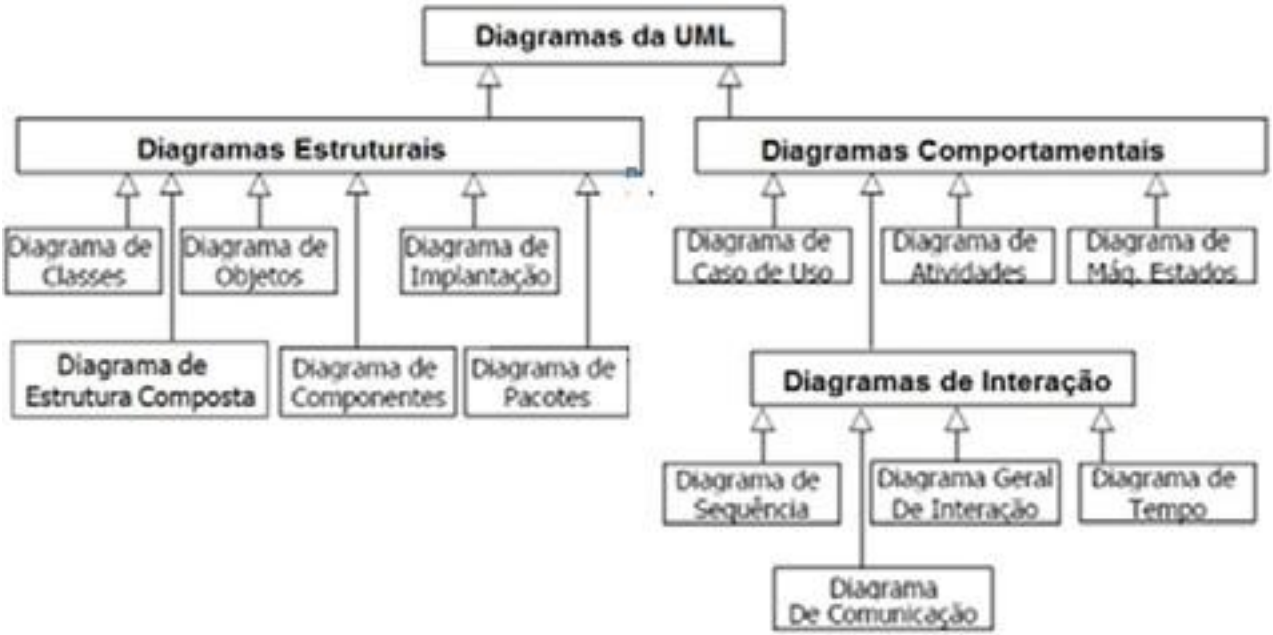


Figura 1 – Diagramas UML (Fonte: OMG, SD)

Fonte: elaborado pela autora.

Alguns autores separam os diagramas da UML, conforme o momento em que são criados dentro do processo de desenvolvimento de software. Por exemplo: Deboni (2003) classifica os diagramas em três grupos: contexto (casos de uso), conceitual (classes conceitual) e detalhado (classes completo, sequência, colaboração, atividades e outros). Estudaremos, detalhadamente, alguns diagramas nessa unidade.

Vamos entender melhor a UML?



Saiba mais!

https://www.youtube.com/watch?v=C3xYBT3o_5k

4.2 Diagrama de Caso de uso e Descrição de caso de uso

Já vimos, na Unidade II, o diagrama de caso de uso e as principais notações do caso de uso. Você se lembra? Vamos recordar?

Um diagrama de caso de uso é um diagrama comportamental da UML que evidencia a visão externa do sistema a ser desenvolvido. Ele mostra quais são as funcionalidades desse sistema e quem interage com elas (BOOCH et al, 2004).

Vamos esclarecer.

Normalmente, cria-se um diagrama de caso de uso chamado de nível 1 com a visão geral e outros casos de uso com visões mais detalhadas. Independente da forma utilizada para a construção dos diagramas de caso de uso, é importante entender que a construção de um diagrama (qualquer que seja ele) não é definitiva. Podem ocorrer e ocorrerão mudanças durante o processo de levantamento de requisitos e mesmo após a implantação do software.



Saiba mais!

Medeiros (2004) diz que um software pequeno pode ter aproximadamente dez diagramas de caso de uso e cada diagrama, em média, teria 8 a 10 casos de uso para serem trabalhados. Isso dá cerca de 80 casos de uso.

Muito trabalho para os desenvolvedores, não?

Uma sugestão de estrutura para um caso de uso segundo Medeiros (2004) Nome Referência: UC03298 - Nome do Caso de Uso

Verbo no infinitivo (informar, comprar, pagar...)

Breve Descritivo

Descrição que informa do que trata este Caso de Uso.

Pré-Condições

Descrição que informa o que é necessário acontecer para que este Caso de Uso se inicie.

Atores Envolvidos Cenário Principal

A descrição de uma tarefa que represente o mundo perfeito, sem exceções. Verbos no presente do indicativo ou substantivos, iniciando a frase com (Registra, Compra, Seleciona, Informa...)

Cenário Alternativo

Qualquer situação que represente uma exceção de um cenário principal. Veja o significado de exceção mais adiante neste capítulo.

Requisitos Especiais

Qualquer situação não contemplada anteriormente (adjetivos), veja exemplos mais adiante.

Dados

Tipos de dados que foram encontrados durante a descrição do Caso de Uso. Aqui informamos: texto, número, data etc., ou mesmo o tipo de dado e seu tamanho, conforme a linguagem a ser utilizada, caso você os conheça.

Sugestão de estrutura de Caso de Uso.

Observação:

Analista de Negócio:

_____

Entrevistado:

_____



Saiba mais!

Vamos ver o vídeo explicando como preencher cada um desses campos?

https://pt.slideshare.net/natanaelsimoes/descrio-formal-de-casos-de-uso

A seguir, estudaremos o diagrama de atividades.

4.3 Diagrama de Atividades

O diagrama de atividades tem como objetivo principal a especificação do comportamento do software, do ponto de vista funcional, ou seja, das suas funcionalidades (OMG, SD).

Esse diagrama tem vários objetivos:

Documentar o aspecto funcional do software. Ele representa o fluxo da informação que o software trabalhará e as condições ou decisões que precisam estar detalhadas ou descritas.

Mostrar aspectos específicos de alguma rotina que será automatizada pelo software. Deve-se evitar especificar toda uma funcionalidade, pois isto pode gerar diagramas de atividades complexos e difíceis de entender.

Mostrar como serão implementados os Requisitos Funcionais.

Documentar de forma macro como o sistema irá funcionar; mostrar como os módulos do sistema interagem entre si e as principais informações utilizadas referentes a entradas e saídas.

Lembra-se do caso de uso do sistema da biblioteca apresentado na Unidade II?

Para recordar:

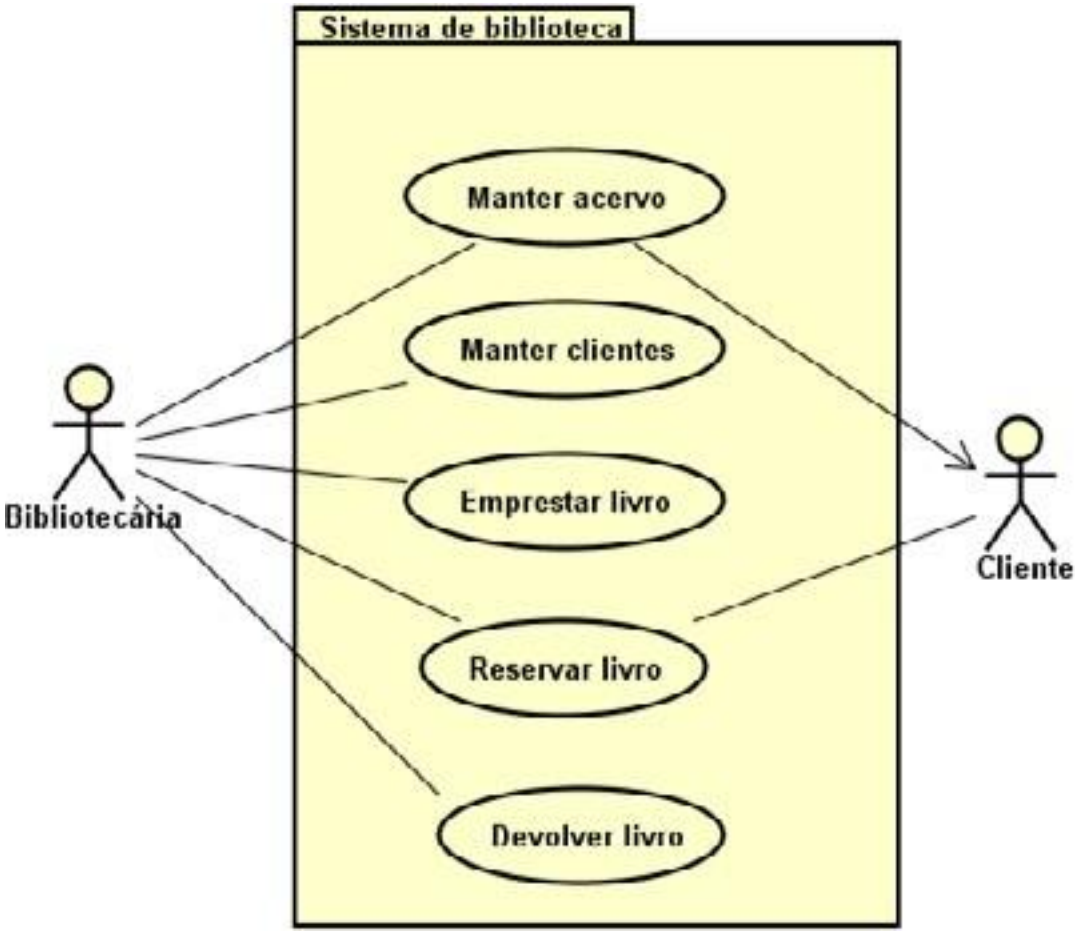


Figura 2 – Sistema de biblioteca.
Fonte: CASTRO et al, 2014.

Anteriormente, afirmamos que os diagramas podem ser modificados. Assim, em uma conversa posterior com o cliente do sistema de biblioteca identificamos que:

A biblioteca é vinculada a uma universidade e só podem ser clientes alunos desta universidade.

Dentro do caso de uso manter clientes, tem-se o envio de e-mail para confirmação de cadastro do cliente e o envio de informações (esclarecimentos) sobre como acessar a biblioteca digital.

Assim, teríamos o sistema de biblioteca, agora apresentado pela Figura 3.

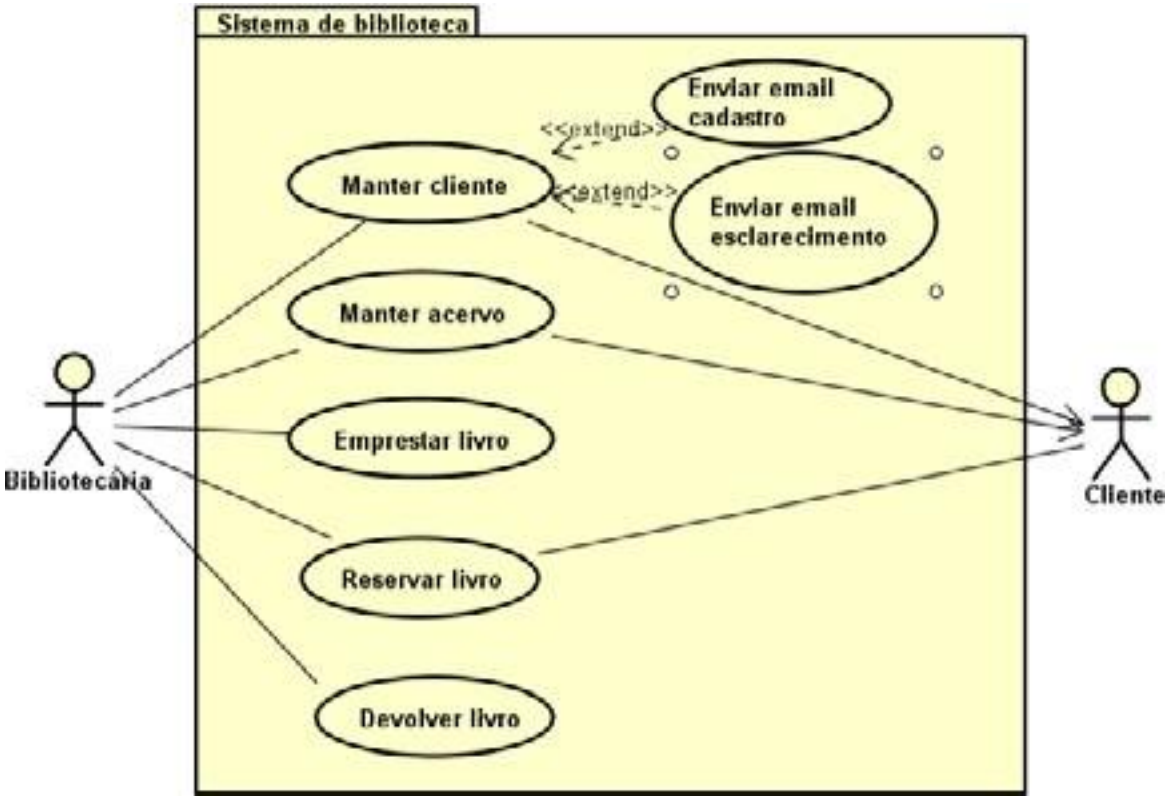


Figura 3 – Sistema de biblioteca representando as novas descobertas.
Fonte: adaptado de CASTRO et al, 2014.

A Figura 4 apresenta um exemplo bem simples para facilitar a compreensão do diagrama de atividade. Vamos entender as notações utilizadas nessa figura.

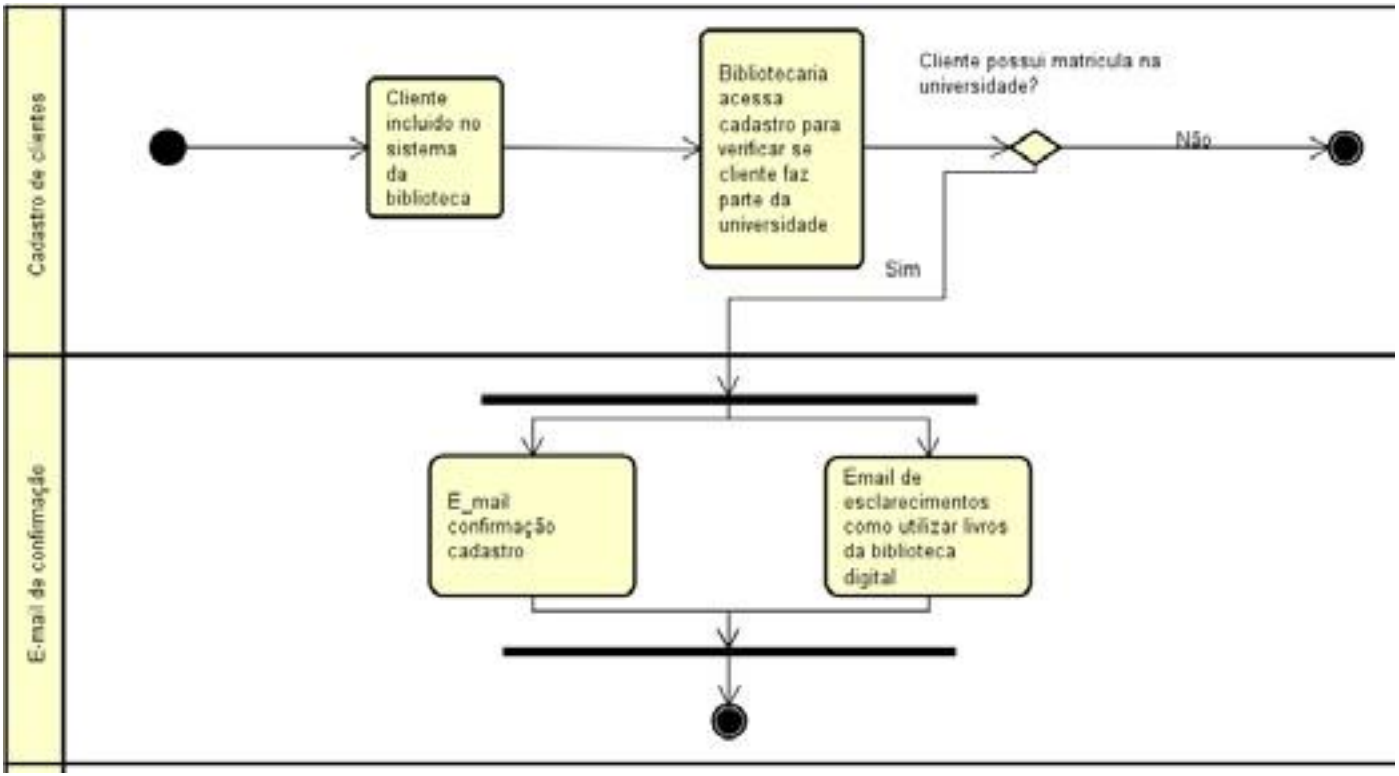


Figura 4 – Exemplo de um diagrama de atividades.
Fonte: elaborado pela autora.

Notações utilizadas em um diagrama de atividades:

- Partição ou raia: ilustra as fronteiras entre módulos, funcionalidades, sistemas ou subsistemas. Pode ser vertical ou horizontal. Uma curiosidade: o nome raia faz uma analogia com as raias das piscinas. A notação para raia é:

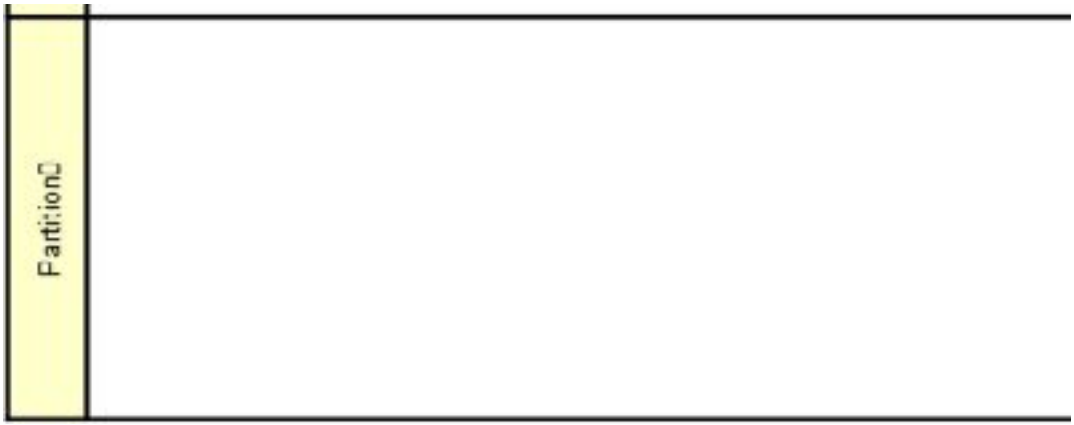


Figura 5 – Exemplo de um partição ou raias
Fonte: elaborado pela autora.

- A Iniciação ou Entrada: define o início do fluxo. Um diagrama de atividades pode ter mais de um elemento de iniciação, pois o seu início pode acontecer em mais de um "local". A notação para a iniciação é 

- A Atividade: representa a ação realizada. No exemplo da Figura 4 temos várias atividades. Você consegue identificar quais? A notação para atividade é:



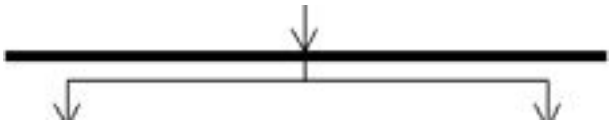
No exemplo acima temos quantas raias? Quais os nomes delas?

- A decisão representa uma condição que pode desviar o fluxo ilustrado no diagrama. Ela é utilizada quando lidamos com decisões. Por exemplo: “O cliente possui matrícula na universidade? Se sim, desvia para a atividade de enviar e-mail, se não, vai para o fim da atividade. A notação para decisão é:

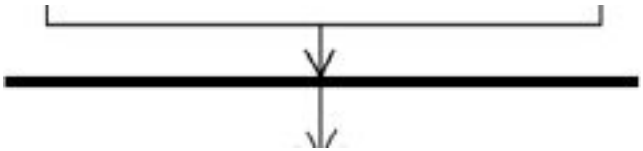


Você identifica quantas decisões na Figura 4?

- Fork/Join - o Fork tem como objetivo dividir o fluxo em mais de uma direção. Sua notação é:



O Join tem a finalidade inversa, faz a união de várias direções do fluxo em uma única direção. A notação é:



Na Figura 4 temos Fork e Join? Quais os fluxos que eles estão dividindo ou juntando?

A Finalização ou Saída define o fim do fluxo. Um diagrama de atividades pode ter mais de uma finalização, pois o diagrama pode ter diversos pontos de saída. Sua notação é:



Na Figura 4 temos quantas saídas? Em que momento elas ocorrem?

Para melhor ilustrar o assunto, vamos ver um vídeo sobre diagrama de atividades.



https://www.youtube.com/watch?v=vReuK7_tYWc

Estudaremos, a seguir, o diagrama de classes e suas principais características.

4.4 Diagrama de classes

O Diagrama de classes tem como objetivo descrever os vários tipos de objetos no sistema e o relacionamento entre eles. Já vimos na Unidade III o conceito de classes e objetos. Você se lembra?

Nós construímos diagramas de classes baseados em casos de uso. Isso facilita a identificar quais os objetos e classes são necessários para o sistema que estamos construindo.

Um diagrama de classes pode ser apresentado em três perspectivas (CASTRO et al, 2014), são elas:

1. Conceitual - apenas classes são utilizadas. Neste tipo de perspectiva, uma classe é interpretada como um conceito. Apenas atributos são utilizados e alguns tipos de relacionamentos.

2. Especificação - tanto classes como interfaces são utilizadas neste tipo de perspectiva. O foco consiste em mostrar as principais interfaces e classes juntamente com seus métodos. Não é necessário mostrar todos os métodos, pois o objetivo deste diagrama, nesta perspectiva, é prover um maior entendimento da arquitetura do software a nível de interfaces.

3. Implementação - nesta perspectiva, vários detalhes de implementação podem ser abordados, tais como: visibilidade de atributos e métodos, parâmetros de cada método, tipos dos atributos e dos valores de retorno de cada método etc.

Nessa unidade, vamos focar, inicialmente, na perspectiva conceitual, pois estudaremos o diagrama de classes como representação das entidades, atributos e relacionamentos. Um diagrama de classes contém entidades e relacionamentos. As entidades podem ser representadas pelas classes. As classes são representadas por um retângulo dividido em três compartimentos (OMG, SD), conforme Figura 6:

Nome - que conterá apenas o nome da classe modelada;

Atributos - que possuirá a relação de atributos que a classe possui em sua estrutura interna;

Operações - que serão os métodos de manipulação de dados e de comunicação de uma classe com outras do sistema.

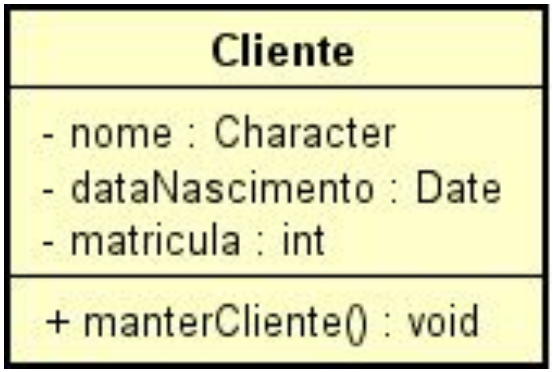


Figura 6 – Exemplo de diagrama de classes.
Fonte: elaborado pela autora.

Os relacionamentos ligam as classes/objetos entre si, criando relações lógicas entre estas entidades. O relacionamento tem:

Papel - descreve o relacionamento. A Figura 7 apresenta o papel “possui” que descreve o relacionamento ou a conexão entre as classes cliente e empréstimo.



Figura 7 – Exemplo de papel.
Fonte: elaborado pela autora.

O que a Figura 7 mostra? Que um objeto cliente pode se relacionar com muitos objetos empréstimo, ou seja, de uma forma bem simples: que um cliente pode ter 0 ou muitos empréstimos de livros.

Multiplicidade: para expressar a multiplicidade entre os relacionamentos, um intervalo indica quantos objetos estão relacionados no link. O intervalo pode ser de zero para um (0..1), zero para vários (0..* ou apenas *), um para vários (1..*). É também possível expressar uma série de números como (1, 4, 3..5). Se não for descrito nenhuma multiplicidade, então é considerado o padrão de um para um (1..1 ou apenas 1). A tabela 1 apresenta de forma sucinta as possibilidades, e a Figura 8, um exemplo.

0..1	No máximo um. Indica que os objetos da classe associada não precisam obrigatoriamente estar relacionados.
1..1	Um e somente um. Indica que apenas um objeto da classe se relaciona com os objetos da outra classe.
0..*	Muitos. Indica que podem haver muitos objetos da classe envolvidos no relacionamento
1..*	Um ou muitos. Indica que há pelo menos um objeto envolvido no relacionamento.
3..5	Valores específicos.

Tabela 1 – Tipos de multiplicidade.
Fonte: elaborado pela autora.



Saiba mais!

Saiba mais sobre o diagrama de classes lendo o artigo: Orientações básicas na elaboração de um diagrama de classes. O artigo é bem detalhado e dá um passo a passo para a elaboração de um diagrama de classes.

https://www.youtube.com/watch?v=rDidOn6KN9k

Os relacionamentos, considerando a perspectiva conceitual, podem ser dos seguintes tipos:

Associação: é uma conexão entre classes e também significa que é uma conexão entre objetos daquelas classes. É representada por uma linha sólida entre duas classes. Pode-se também colocar uma seta no final da associação indicando que esta só pode ser usada para o lado em que a seta aponta (OMG, SD). A Figura 8 apresenta esses dois tipos de representação. Esse diagrama de classes representa um modelo simples em que um cliente tem faturas e pode efetuar o pagamento. A classe pagamento é um exemplo de classe abstrata.

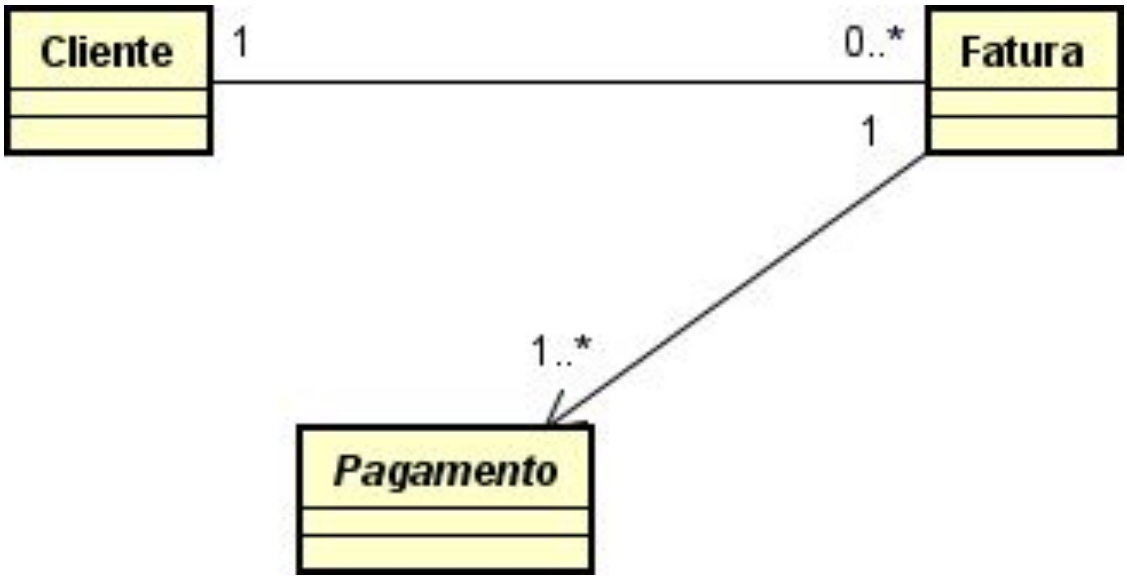


Figura 8 – Exemplo de associação.
Fonte: elaborado pela autora.



Para Refletir...

Pesquise a diferença entre classe concreta e abstrata. Analise e tente responder à pergunta:

- Por que pagamento é uma classe abstrata?

Herança/Generalização: já vimos o conceito de Herança/Generalização na Unidade III. Mecanismo baseado em objetos que permite que as classes compartilhem atributos e operações baseados em um relacionamento, geralmente, generalização.

A Representação Gráfica da Herança ou Generalização para a UML é 

A Figura 9 apresenta um exemplo de Herança/ Generalização, considerando uma modificação do diagrama de classes da Figura 6. Agora foram adicionadas 3 formas de pagamento. Quais seriam elas?

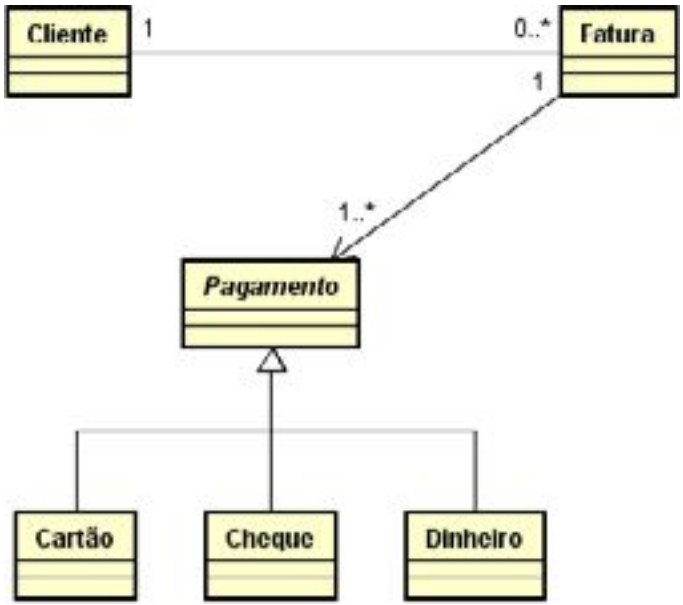


Figura 9 - Exemplo de Herança/Generalização.
Fonte: elaborado pela autora).



Saiba mais!

Veja o vídeo “UML - Diagrama de Classe - Parte 2 – Associação” para complementar o seu conhecimento.

https://www.youtube.com/watch?v=FL7zsHDqLXY

4.5 Outros diagramas

A UML possui uma série de outros diagramas conforme você pode constatar na Figura 1. A seguir, descreveremos o objetivo de alguns deles. É interessante que você veja os vídeos e leia os textos indicados para compreender melhor cada um deles. Ok?

Diagrama de Sequência - tem o objetivo de mostrar como as mensagens entre os objetos são trocadas no decorrer do tempo para a realização de uma operação. Geralmente, um diagrama de sequência é criado a partir de um diagrama de casos de uso, com a finalidade de descrever como serão as interações entre cada objeto/elemento do diagrama.

Vamos assistir ao vídeo abaixo? Ele mostra, passo a passo, a construção de um diagrama de sequência.



https://www.youtube.com/watch?v=ypP6HQdDxYM

Diagrama de objetos – o diagrama de objetos é uma instância do diagrama de classes (MEDEIROS, 2004). Na Unidade III, vimos a diferença entre classes e objetos. Você se lembra? Os diagramas de objetos são opcionais e não são tão importantes como os diagramas de classes. Porém, eles podem ser utilizados de forma complementar para esclarecer o entendimento de diagramas complexos ajudando na compreensão do sistema tanto para os desenvolvedores como para os clientes.



Saiba mais!

Leia um exemplo do diagrama de objetos no artigo a seguir.

http://micreiros.com/diagrama-de-objetos/

Ainda existem outros diagramas que não vimos nesta unidade. Você pode conhecê-los acessando os links a seguir ou procurando outros links na internet.

Diagrama de Implantação



https://estudonahttps://estudonaweb.com.br/artigo/entendendo-o-diagrama-de-implantacaoweb.com.br/artigo/entendendo-o-diagrama-de-implantacao

Diagrama de componentes



https://estudonaweb.com.br/artigo/entendendo-o-diagrama-de-componentes

4.6 Introdução a padrões do projeto

O padrão de projeto é um modelo de uma experiência amplamente testada e aprovada e que pode ser usado como um guia para a resolução de problemas específicos de arquiteturas orientadas a objetos (GAMMA et al, 2006).

Padrões e Linguagens de padrões são formas de descrever as melhores práticas em bons projetos e capturar a experiência de uma forma que torne possível a outros reusar essa experiência.

O conhecimento de padrões de projeto permite a criação de sistemas melhores, com manutenção menos custosa e reduzindo o retrabalho. Assim, o domínio sobre padrões de projeto é muito importante ao desenvolvedor de software.

Os padrões registram as experiências bem-sucedidas, tais como:

Livro “Design Patterns” - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four- GoF);

Padrões J2EE da Sun. Disponível em: http://java.sun.com/blueprints /patterns/.

GoF (Gang of four) é a referência clássica; muitos dos padrões conhecidos são baseados no catálogo GoF (GAMMA et al, 2006). Mas devemos lembrar que novos problemas incitam a criação de novos padrões de projeto, e padrões novos sempre estão surgindo.

Autores como Pressman e Maxim (2016), em uma de suas obras sobre engenharia de software, apresentam alguns pontos importantes que nos ajudam a compreender sobre padrões, tais como: elementos essenciais e passos de como utilizar um padrão. Veja a seguir.

Para esses autores, são elementos essenciais aos padrões:

- Nome;

- Descrição do problema e contexto para os quais o padrão se aplica;

- Descrição da solução genérica proposta;

- Consequências da aplicação do padrão (custos e benefícios).

Vamos ver um exemplo?

Vamos pensar que você quer desenvolver um software de e-comerce. Segundo Pressman e Maxim (2016, p.364):

> **E-commerce: específico para sites, estes padrões implementam elementos recorrentes e aplicações para comércio eletrônico.**
>
> **Padrão: shoppingcart**
>
> **Descrição: fornece uma lista de itens selecionados para compra.**
>
> **Detalhes:lista informações de itens, quantidade, código de produto, disponibilidade (disponível, esgotado), preço, informações para entrega, custos de remessa e outras informações de compra relevantes. Também oferece a habilidade de editar (por exemplo: remover, modificar quantidade).**
>
> **Elementos de navegação: Contém a capacidade de prosseguir com a compra ou sair para encerrar as compras.**

Segundo esses autores, os padrões são semiprontos. Isso quer dizer que sempre temos que revê-los, analisá-los e adaptá-los ao nosso ambiente.

Em sua obra, eles ainda nos ajudam com alguns passos para utilizar um padrão:

1. Leia o padrão todo uma vez;
2. Estude o código fonte de exemplo;

3. Escolha nomes para os participantes do padrão dentro do seu contexto;

4. Defina as novas classes e modifique classes existentes que são afetadas;

5. Defina nomes para as operações do padrão dentro do seu contexto;

6. Implemente as operações.

Alguns padrões do GoF: Abstract Factory, Prototype, Builder, Adapter, Composite, Chain of Responsibility, Observer, etc.

Acesse o Link e veja a definição do factory.

https://brizeno.wordpress.com/2011/09/17/mao-na-massa-factory-method/

Vamos complementar essas informações com o vídeo a seguir. Tente relacionar o conteúdo exibido com o que discutimos até agora.



https://www.youtube.com/watch?v=5CgwgRHhCRM



Bora rever!

Caro estudante, nesta unidade, você estudou a UML que é uma linguagem utilizada para modelar sistemas e os seus diagramas. A UML tem 13 diagramas e foram revistos alguns conceitos já estudados em unidades anteriores e detalhados outros conceitos com relação a essa linguagem. Entendemos que os diagramas de Casos de Uso necessitam ser descritos para facilitar o entendimento. Estudamos o objetivo e como construir um diagrama de atividades que especifica o comportamento do software.

Vimos também o diagrama de classes, sua importância e objetivo. O diagrama de classes descreve os objetos, relacionamentos e tipos de multiplicidade.

Estudamos, também, algumas características do diagrama de sequência e do diagrama de objeto. Foram disponibilizados links para você aprender os principais aspectos do diagrama de implantação e diagrama de componentes. Ao final, vimos uma introdução a padrões de projeto. Sua importância e como podem ser implementados em um sistema para melhorar a qualidade e promover o reuso das especificações. Espero que tenha gostado e assimilado estes conhecimentos de forma a aplicá-los em sua vida profissional.



Figura 10 - Tirinha
Fonte: Imagem disponível em: <https://vidadeprogramador.com.br/2011/11/23/sistema-inovador>



Questões de autoaprendizagem

Assinale a alternativa correta.

1. É motivo para se descrever um caso de uso:

a. ( ) Os diagramas de caso de uso não esclarecem totalmente as necessidades.
b. ( ) Os diagramas de caso de uso podem propiciar uma interpretação equivocada.
c. ( ) A descrição de caso de uso possibilita uma definição melhor do escopo.
d. ( ) A descrição de caso de uso facilita a programação.
e. ( ) Todas as alternativas anteriores

Solução – E. São vários os motivos para descrever um caso de uso: os diagramas não esclarecem totalmente as necessidades e podem propiciar uma interpretação equivocada. Além disso, descrever define melhor o escopo e facilita a programação.



2. O diagrama de atividades tem como objetivo principal a especificação do comportamento do software, do ponto de vista funcional, ou seja, das suas funcionalidades.

O Diagrama de classes tem como objetivo descrever os vários tipos de objetos no sistema e o relacionamento entre eles.

Acerca dessas asserções, assinale a opção correta.

a. ( ) As duas asserções são proposições verdadeiras, e a segunda é uma complementação da primeira.
b. ( ) As duas asserções são proposições verdadeiras, mas a segunda não é uma complementação da primeira.
c. ( ) A primeira asserção é uma proposição verdadeira, e a segunda, uma proposição falsa.
d. ( ) A primeira asserção é uma proposição falsa, e a segunda, uma proposição verdadeira.
e. ( ) Tanto a primeira quanto a segunda asserções são proposições falsas.

Solução. As duas proposições estão corretas, e a segunda não é complementação da primeira. O diagrama de atividades tem como objetivo principal a especificação do comportamento do software, do ponto de vista funcional, ou seja, das suas funcionalidades. O Diagrama de classes tem como objetivo descrever os vários tipos de objetos no sistema e o relacionamento entre eles.



Referências

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. Rio de Janeiro: Campus, 2006.

BOOCH,G.;RUMBAUGHJ.;JACOBSONI.**UML**:guiadousuário .2.ed. Rio de Janeiro: Campus, 2004.

BOOCH,G. **Object- oriented analysis and design with applications**. Califórnia: Benjamin/Cummin- gs Pub,1994.

CASTRO, E.R.C, CALAZANS, A.T.S, PALDES, R.A., GUIMARAES, F.A. **Engenharia de requisitos**: um enfoque prático na construção de software orientado a negócio. Florianopolis: Bookess, 2014.

INSTITUTE OF ELECTRICALAND ELECTRONICS ENGINEERS/COMPUTERSCIENCE. **Guide to the Software Engineering Body of Knowledge** v.3(SWEBOK). 2014. Disponível em:<https://www.computer.org/web/swebok/v3 >. Acesso em 13 de dez.2018.

JACOBSON, I; ERICSSON, M; JACOBSON,A.**The Object Advantage:** Business Process Reengineering With Object Technology. Addison-Wesley Professional, 1994.

MEDEIROS, E**. Desenvolvendo Software** Com Uml 2.0 Definitivo. SÃO PAULO: Pearson Makron Books, 2004.



Referências

MOREIRA T.R.F., RIOS, E. **Projeto e Engenharia de Software**: Teste de Software.. Editora: Alta Books, 2003.

PFLEEGER, S.L. **Engenharia de software**: teoria e prática. São Paulo: Pearson, 2004.

PMI. **Um guia de conhecimento de gerenciamento de projetos** – Guia PMbok.6ª. ed. PMI -Project. Management Institute: 2018 .

RUMBAUGH, J.; BLAHA, M.; EDDY, F.; LORENSEN, W. **Object Modeling Technique** - OMT, 1991.

SBROCCO, J.H.T.C. **Metodologias ágeis**: engenharia de software sob medida. São Paulo: Erica, 2012.

SCHWABER,K. SUTHERLAND, J. **Guia do Scrum Um guia definitivo para o Scrum:** as regras do Jogo .2017. Disponível em: <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-Portuguese-Brazilian.pdf.>. Acesso em: 24 de nov.2018.

SOMMERVILLE, I. **Engenharia de software**.9ª. edição. São Paulo: Pearson Prentice Hall, 2011.

SUTHERLAND, J. **Scrum: a arte de fazer o dobro do trabalho na metade do tempo** /; tradução de Natalie Gerhardt. -São Paulo: LeYa, 2014.



Currículo da Professora Autora

Possui doutorado em Ciência da Informação pela Universidade de Brasília (2008) e mestrado em Gestão do Conhecimento e TI pela Universidade Católica de Brasília (2003). Atuou, por 28 anos, como especialista em TI da Caixa Econômica Federal. Tem experiência na área de Ciência da Computação e Ciência da Informação. Professora universitária há mais de 10 anos, atuando na graduação e pós graduação de cursos de TI. Pesquisadora, atuando principalmente nos seguintes temas: análise por pontos de função, métricas, data mart, data warehouse e processo de desenvolvimento de software, qualidade de software, qualidade da informação, metodologia de pesquisa, contratação de serviços de TI, modelos ágeis, engenharia de requisitos, metodologias ativas de aprendizagem, tecnologia da informação e comunicação.

Angélica Toffano Seidel Calazans

ISBN: 978-85-64124-88-2

INSTITUTO FEDERAL
Brasília

Secretaria de
Educação Profissional
e Tecnológica

MINISTÉRIO DA
EDUCAÇÃO