COMPUTAÇÃO PARA DISPOSITIVOS MÓVEIS

CAPÍTULO 3 - COMO PODEMOS CONSTRUIR APLICATIVOS DE QUALIDADE?

Fernando Skackauskas Dias

INICIAR

Introdução

Com a tendência cada vez mais forte de desenvolver aplicativos para dispositivos móveis, para os mais diversos fins, o mercado passa a ser mais exigente com qualidade, o que exige critérios mais elevados de boa interação, funcionalidades inteligentes, alto desempenho, conectividade com as mais diversas plataformas e com outros aplicativos. Para atender a essas exigências, os sistemas operacionais são desenvolvidos com classes e componentes para serem utilizadas como recursos nos aplicativos.

Estes recursos concentram utilidades, como ferramentas para depuração e testes, que auxiliam os desenvolvedores a construírem aplicativos com maior agilidade e robustez. Outra função dos SOs dispositivos móveis tem como objetivo analisar o desempenho e a automatização na construção de aplicativos.

Neste capítulo, vamos entender o Databinding do SO Android, permitindo ao desenvolvedor escrever *layouts* declarativos. Vamos aprender, também, a usar os padrões MVP (Model-View-Presenter) e MVVM (Model-View-ViewModel) e o componente Dagger2, e demonstrar como utilizar as ferramentas de teste e depuração como o JUnit, Espresso e Robotium. Será mostrado como realizar a análise de desempenho por meio de ferramentas de análise. Por fim, serão mostradas as ferramentas de automatização de construção como o Gradle e Jenkins.

Acompanhe com atenção e bons estudos!

3.1 Padrões de projeto

Vamos começar entendendo que padrão de projeto é uma solução reutilizável, ou seja, um modelo e não uma solução implantada. Com o crescimento das aplicações móveis, e da concorrência entre eles, a busca por qualidade, manutenibilidade e clareza do código, se tornou essencial. Segundo Barbosa (2013, p. 11) “é preciso aplicar padrões de projeto para programação de aplicativos para esses dispositivos, seja eles: Symbian, iOS (sistema operacional da Apple), Android ou Windows Phone”.

Mas, como é possível adotar padrões de projeto para aplicativos móveis? Quais os padrões disponíveis? Como utilizar os padrões de projeto para dispositivos móveis?

Eles são utilizados por muitas empresas que buscam implantar novos projetos de *software* ou melhorar os antigos. Um padrão de projeto pode ser implantado em qualquer linguagem ou plataforma de desenvolvimento.

3.1.1 Databinding

O Android oferece suporte para escrever *layouts* declarativos, usando o Databinding, também, chamado de vinculação de dados. Databinding permite sincronizar a interface de usuário com o modelo de aplicativo e a lógica. Para Lee (*et al.*, 2005), isso minimiza o código necessário do aplicativo para se conectar aos elementos da interface do usuário.

O uso de Databinding requer alterações nos arquivos de *layout*. Eles iniciam com uma tarefa raiz, seguido de um elemento data e um elemento de visão. Os

elementos de dados descrevem os dados que estão disponíveis para ligação. O elemento de visão contém sua hierarquia semelhante aos arquivos de *layout*, que não são usados com Databinding.

As referências aos elementos de dados ou expressões dentro do *layout* são gravadas nas propriedades do atributo usando o @{}ou @={}. Na figura a seguir, é demonstrada a estrutura de vinculação de dados no Databinding.

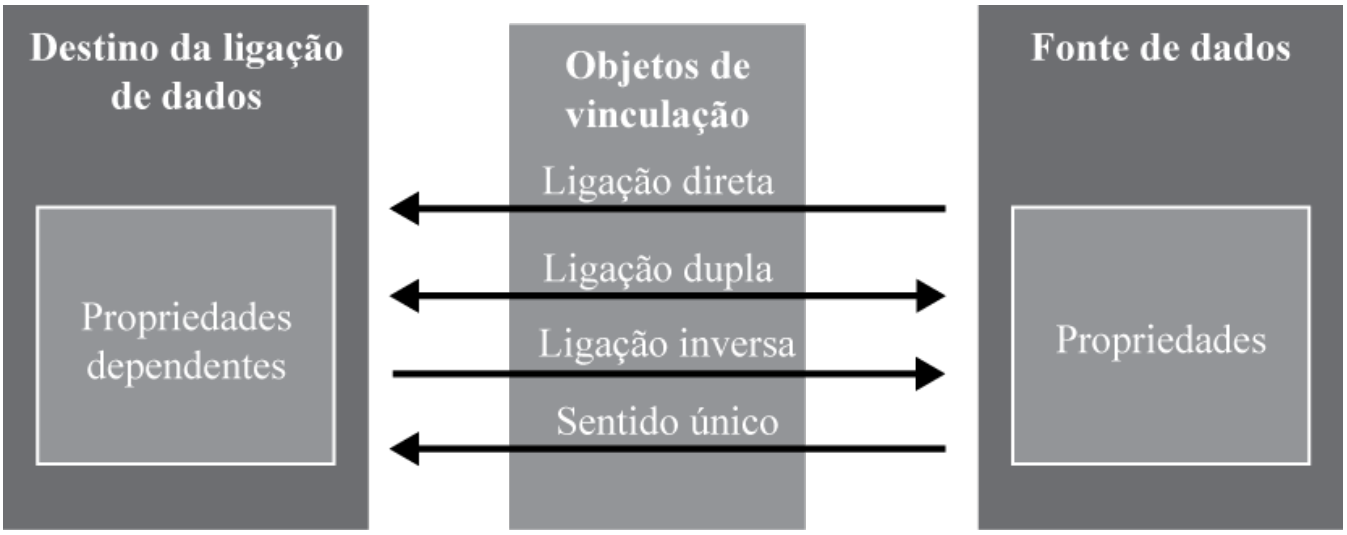


Figura 1 - Modelo de Databinding do sistema operacional Android, indicando os módulos e suas correlações. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta três quadros, um do lado do outro. O primeiro é: Destino da ligação de dados e dentro dele tem um quadrado cinza claro escrito: Propriedades dependentes. O terceiro é: Fonte de dados e dentro dele tem um quadrado cinza claro escrito: Propriedades. O segundo é: Objetos de vinculação e abaixo tem ligação direta com a seta apontando do terceiro quadro para o primeiro; embaixo tem Ligação dupla com seta de via dupla entre o primeiro e o terceiro quadro; embaixo tem ligação inversa com a seta apontando do primeiro para o terceiro quadro; embaixo tem Sentido único com a seta apontando do terceiro para o primeiro quadro.

A Databinding do Android gera uma classe “Binding”, com base nesse *layout*. Esta classe contém todas as ligações das propriedades de *layout*, isto é, a variável definida para as visualizações correspondentes. Ele também fornece as classes “setters” geradas para seus elementos de dados, a partir do *layout*. O nome da classe gerada é baseado no nome do arquivo de *layout*. Este nome é convertido e é

adicionado a ele. Por exemplo, se o arquivo de *layout* for chamado activity_main.xml(), o método generate() será chamada ActivityMainBinding, sendo possível incrementar o *layout* e conectar o modelo por meio dessa classe.

Portanto, a vinculação de dados é construída com a ideia de usar dados de um objeto Java para definir atributos em seus *layouts*, e essa é a medida em que é muito utilizado. Isto permite que se defina a lógica de visualização independentemente do Android *Framework*.

3.1.2 Aplicabilidade da arquitetura MVP ou MVVM

O padrão MVP (*Model-View-Presenter*) é um derivativo do conhecido MVC (*Model-View-Controller*), que tem sido importante para o desenvolvimento de aplicativos Android. O padrão MVP permite separar a camada de apresentação da camada lógica, de modo que as informações sobre como a interface funciona são separadas de como será representada a tela. Inicialmente, o padrão MVP teria essa mesma lógica, podendo ter visões completamente diferentes e intercambiáveis.

VOCÊ SABIA?

Os aplicativos móveis, que incluem calendários, navegadores e mapas interativos, fazem parte da vida da maioria das pessoas hoje em dia. A maioria dos aplicativos móveis é de usuário único; e eles não permitem a colaboração síncrona entre os usuários. A colaboração móvel permite que vários usuários em diversos locais combinem sinergicamente suas contribuições de maneira conveniente.

Os pesquisadores Pichiliani e Hirata (2014) investigaram o potencial de criar aplicativos colaborativos com base em aplicativos móveis de usuário único. Neste trabalho foi proposta uma técnica de adaptação, baseada na reutilização de SDKs (*Software Development Kits*) de fabricantes para criar aplicativos protótipos multiusuários. Leia mais em <https:// (https://link.springer.com/article/10.1007/s11036-014-0512-0)*link (https://link.springer.com/article/10.1007/s11036-014-0512-0)*.springer.com/article/10.1007/s11036-014-0512-0 (https://link.springer.com/article/10.1007/s11036-014-0512-0)>.

É fundamental esclarecer que o MVP não é um padrão de arquitetura, ele é responsável apenas pela camada de apresentação. Mas, como recomendam Deitel, Deitel e Wald (2016), é sempre melhor usá-lo para sua arquitetura, que não

o usar de forma alguma. No Android, temos um problema decorrente do fato de as atividades do Android serem intimamente associadas aos mecanismos de acesso à interface e aos dados. Podemos encontrar exemplos extremos como o componente CursorAdapter, que combina adaptadores, que são parte da visão, com cursores, algo que deve ser relegado às profundezas da camada de acesso a dados. A seguir é demonstrado o modelo MVP:

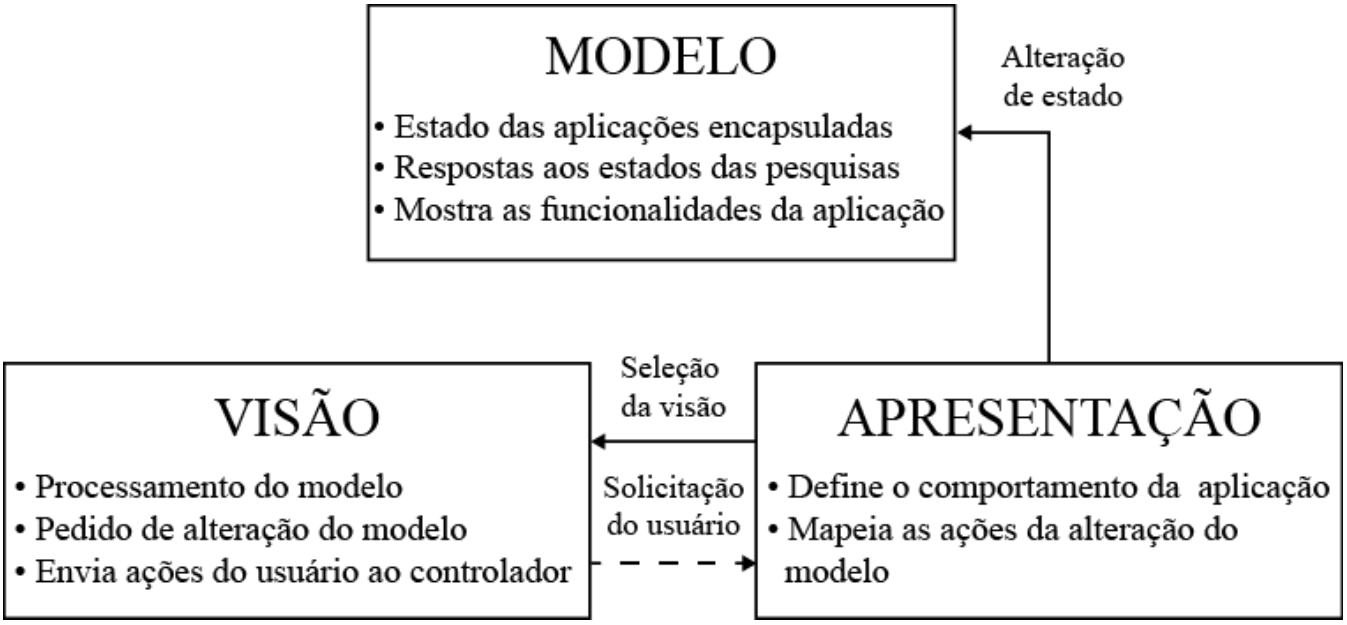


Figura 2 - Modelo MVP do sistema operacional Android, indicando os módulos e suas correlações. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta três quadros, um no topo e dois embaixo um do lado do outro. O primeiro no topo é o quadro Modelo: - Estado das aplicações encapsuladas; - Resposta aos estados das pesquisas; - Mostra as funcionalidades da aplicação. O quadro abaixo e a esquerda é o quadro Visão: - Processamento do modelo; - Pedido de alteração do modelo; -Envia ações do usuário ao controlador. Do lado deste é o quadro Apresentação: - Define o comportamento da aplicação; -Mapeia as ações da alteração do modelo. O quadro Apresentação se conecta ao quadro Modelo com uma seta e escrita: Alteração de estado. O quadro apresentação se conecta ao quadro Visão com uma seta escrita: Seleção da visão. O quadro visão se conecta ao quadro Apresentação com uma seta pontilhada e escrita: Solicitação do usuário.

Para que uma aplicação seja facilmente extensível e sustentável, é necessário definir camadas separadas. O que faremos se, em vez de recuperar os mesmos

dados de um banco de dados, precisarmos fazer isso de um serviço da *web*?

Seria preciso refazer toda a parte de apresentação do aplicativo, denominada visão pelo MVP. O MVP torna as visualizações independentes da fonte de dados. O aplicativo é dividido em, pelo menos, três camadas diferentes, o que permite testá-las independentemente. Com o MVP, pode-se tirar a maior parte da lógica das atividades para que possa testá-la sem usar testes de instrumentação.

Existem muitas variações do MVP e todas podem ajustar a ideia do padrão às necessidades de desenvolvimento de cada aplicativo. O modelo varia dependendo basicamente da quantidade de funcionalidades do aplicativo. A visão é responsável por habilitar ou desabilitar uma barra de progresso, ou deveria ser feita pelo módulo apresentador? E qual módulo decide quais ações devem ser mostradas na barra de ação?

Por outro lado, temos o MVVM (*Model-View-ViewModel*). À primeira vista, o MVVM parece semelhante ao padrão *Model-View-Presenter*, porque ambos fazem o trabalho de abstrair o estado e o comportamento da visão.

O Modelo de Apresentação abstrai uma visualização independente de uma plataforma específica de interface com o usuário, enquanto o padrão MVVM foi criado para simplificar a programação orientada a eventos das interfaces com o usuário. Na figura a seguir, é demonstrado o modelo MVVM.

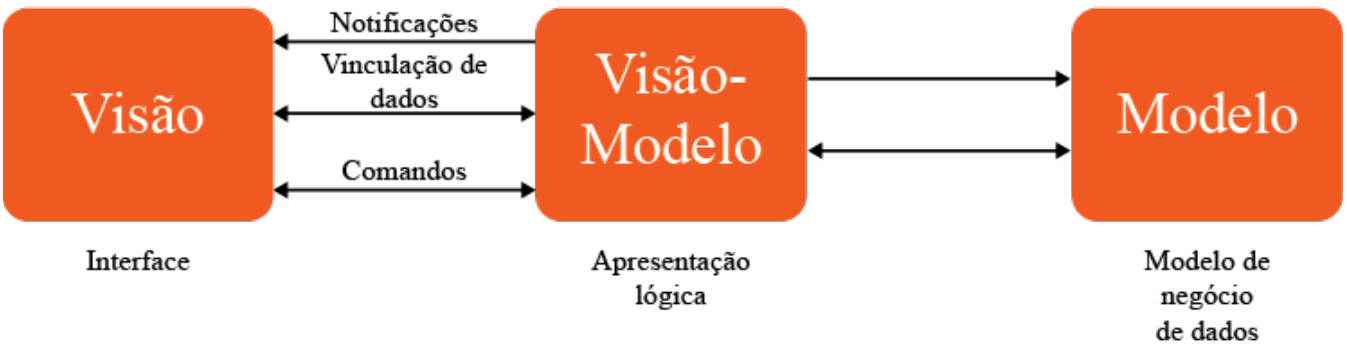


Figura 3 - Representação do modelo MVVM (Model-View-ViewModel), mostrando a relação entre os módulos e suas vinculações. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta três quadros laranjas, um do lado do outro. O primeiro é o quadro Visão, embaixo dele está escrito Interface; o Segundo é o quadro Visão-Modelo, embaixo dele está escrito Apresentação lógica; o Terceiro é o quadro Modelo, embaixo dele está escrito Modelo de negócio de dados. O primeiro e o

segundo quadro estão ligados através de uma seta dupla escrito: Vinculação de dados e outra seta dupla escrito: Comandos. O segundo quadro está ligado ao primeiro por uma seta escrito: Notificações. O segundo quadro está ligado ao terceiro por uma seta e o terceiro quadro está ligado ao segundo por uma uma seta.

Se o padrão do MVP significava que o módulo de apresentação estava ligado diretamente à visualização, demonstrando o que exibir, no MVVM, o ViewModel expõe fluxos de eventos, aos quais, o módulo visão pode se associar. Assim, o ViewModel não precisa mais manter uma referência ao módulo visão, como o módulo de apresentação. Isso também significa que todas as interfaces que o padrão MVP requer agora, são descartadas. As visões também notificam ao ViewModel sobre diferentes ações. Portanto, o padrão MVVM suporta interação de dados bidirecional entre View e ViewModel e há um relacionamento de “muitos para um” entre View e ViewModel. View tem uma referência ao ViewModel, mas o ViewModel não possui informações sobre o View.

3.1.3 Injeção de dependência pela ferramenta Dagger 2

Muitos aplicativos Android dependem da instanciação de objetos que, geralmente, exigem outras dependências. Por exemplo, um cliente do aplicativo Twitter pode ser criado usando uma biblioteca de rede, como o Retrofit. Para usar essa biblioteca, talvez seja necessário adicionar bibliotecas de serialização, como o Gson. Além disso, as classes que implantam a autenticação ou o armazenamento em cache podem exigir o acesso a preferências compartilhadas ou outro armazenamento comum, exigindo sua instanciação em primeiro e a criação de uma cadeia de dependência.

Para esta situação, o Dagger 2 analisa as dependências e gera o código para ajudar a conectá-las. Embora existam outras estruturas de injeção de dependência em Java, muitas delas sofreram limitações ao depender de XML, ou exigiram a validação de problemas de dependência em tempo de execução, ou ainda, incorreram em penalidades de desempenho durante a inicialização. O Dagger 2 baseia-se puramente na utilização de processadores de anotação Java e verificações em tempo de compilação para analisar e verificar dependências. É considerado um dos *frameworks* de injeção de dependência mais eficientes construídos (LEE *et al.*, 2005).

Por fim, o Dagger 2 é um dos *frameworks* de código aberto que gera ainda muitas dúvidas. Mas por que é melhor que os outros?

Atualmente, é a única estrutura que gera código-fonte totalmente rastreável em Java, que imita o código que o usuário pode escrever à mão. Isso significa que não há componente oculto na construção do gráfico de dependências. O Dagger 2 é menos dinâmico do que os outros, mas a simplicidade e o desempenho do código gerado estão no mesmo nível do código escrito à mão, tornando-o, portanto, melhor que os outros.

O Dagger 2 analisa as dependências e gera o código para conectá-las. Embora existam outras estruturas de injeção de dependência de Java, muitas delas sofreram limitações ao depender de XML, exigiram a validação de problemas de dependência em tempo de execução ou incorreram em penalidades de desempenho durante a inicialização. O Dagger 2 baseia-se puramente na utilização de processadores de anotação Java e verificações em tempo de compilação para analisar e verificar dependências. É considerado um dos *frameworks* de injeção de dependência mais eficientes construídos. O Dagger 2 simplifica o acesso a instâncias compartilhadas, fornecendo uma maneira simples de obter referências a estas instâncias. O Dagger 2 também oferece uma configuração fácil, como também é mais fácil executar os testes de unidade e integração.

3.2 Testes e depuração

Para o processo de desenvolvimento de aplicativos para dispositivos móveis, uma das fases mais importantes é a fase de teste de depuração. Isto garante a entrega de aplicativos com o mínimo de erros e com boa qualidade. Como é possível realizar testes e depuração de forma automatizada nos sistemas operacionais disponíveis nos aplicativos móveis? Quais são as ferramentas disponíveis? Quais são os procedimentos necessários?

Vamos ver a seguir, como é possível realizar testes e depuração em aplicativos móveis.

3.2.1 Testes unitários pela ferramenta JUnit

JUnit é uma biblioteca de código aberto com o propósito de escrever e executar

casos de teste para programas Java e, também, para aplicativos que executam em dispositivos móveis. No caso de programas Java, quando o foco são aplicações *web*, o JUnit é usado para testar o aplicativo sem o servidor. Este *framework* constrói uma relação entre desenvolvimento e processo de teste.

O teste unitário é um método de teste de *software* usado para testar unidades individuais do código-fonte, para determinar se elas estão adequadas para uso. O objetivo é escrever código para cada função ou método não trivial. Ele permite que seja possível verificar, com relativa rapidez, se a última alteração no código causa regressão, isto é, se novos erros aparecem na parte do programa que já foi testada e facilita a identificação e a eliminação desses erros. Ou seja, os testes são métodos que verificam a eficiência do seu código. Mas como exatamente isso tudo funciona? O teste é considerado concluído se nenhum erro for encontrado. E para vários tipos de verificações, são usados métodos suplementares.

Existem dois tipos de testes unitários no sistema operacional Android:

- **testes de unidade local**: são executados usando apenas a JVM. Eles servem para experimentar a lógica de negócios que não está interagindo com o sistema operacional;
- **testes de unidade instrumentados**: são usados para provar a lógica interconectada com a API do Android. Eles são executados no dispositivo físico/emulador e, portanto, levam mais tempo do que os testes locais.

A seguir, vamos listar as vantagens de se utilizar o JUnit. A primeira é o teste automatizado:

- a automação rápida executa casos de teste de forma significativamente mais rápida do que recursos humanos;
- menos investimento em recursos humanos: os casos de teste são executados usando a ferramenta de automação, o que reduz a quantidade de testes;
- mais confiável: os testes de automação executam precisamente a mesma operação, sempre que são executados;
- programável: os testadores podem programar testes sofisticados para trazer informações relevantes e diferentes.

O *framework* JUnit possui um conjunto de métodos básicos:

- void assertEquals (esperado, atual) - verifica se os dois primitivos/ objetos, são iguais;
- void assertTrue (*condition*) - verifica se uma condição é verdadeira;
- void assertFalse (*condition*) - verifica se uma condição é falsa;
- void assertNotNull (*object*) - verifica se um objeto não é nulo;
- void assertNull (*object*) - verifica se um objeto é nulo;
- void assertSame (esperado, atual) - testa se duas referências de objeto apontam para o mesmo objeto;
- void assertNotSame (inesperado, real) - testa se duas referências de objetos não apontam para o mesmo objeto;
- void assertArrayEquals (expectedArray, actualArray) - testa se dois vetores são iguais entre si.

Assim, percebemos que os testes realizados pela ferramenta JUnit permitem detectar uma série de possíveis erros de código do aplicativo, de forma automatizada e confiável.

3.2.2 Analisar testes de interface do usuário com as ferramentas Espresso e Robotium

O Robotium é uma extensão da estrutura de teste do Android e foi criado para facilitar a criação de testes de interface do usuário para aplicativos Android. Os testes permitem que se definam casos de teste nas atividades do Android. Na figura a seguir, podemos ver a hierarquia de funcionalidades do Robotium.

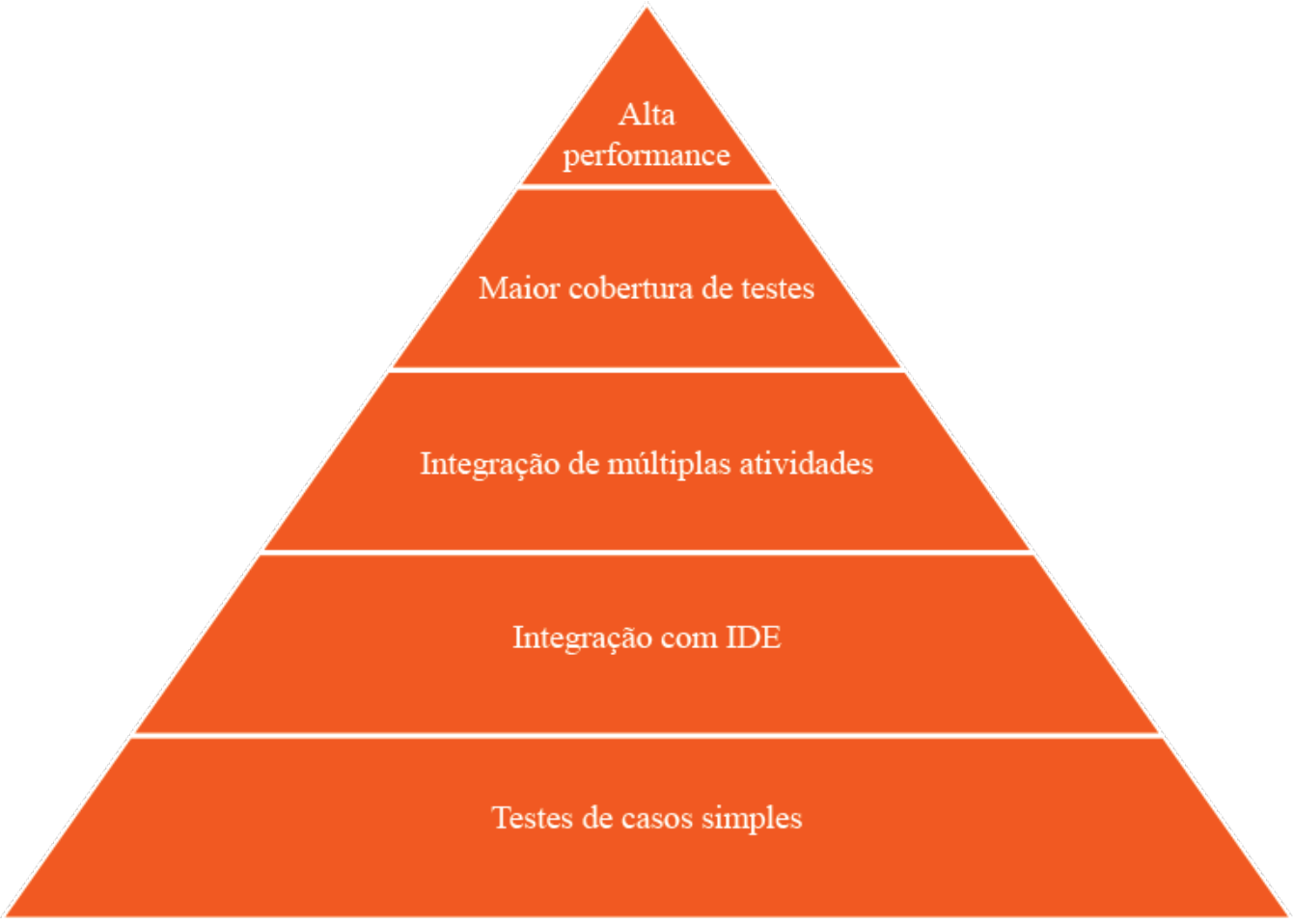


Figura 4 - Vantagens hierarquizadas do Robotium no sistema operacional Android. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta uma pirâmide hierárquica de cor laranja,composta por 5 níveis. No topo está escrito: Alta performace; no nível abaixo está escrito: Maior cobertura de testes; no nível abaixo está escrito: Integração de multiplas atividades; no nível abaixo está escrito: Integração com IDE; no nível abaixo (último nível - base da pirâmide) está escrito: Testes de casos simples.

O Robotium se baseia nas principais bibliotecas de testes de integração do Android, mas fornece uma camada "extra" no topo para facilitar o teste com o Solodriver. O Solodrive permite definir valores em campos de entrada, clicar nos botões e obter resultados de outros componentes da interface do usuário. Os métodos da classe JUnits Assert podem ser usados para verificar esses resultados. Portanto, o objeto Solo pode ser usado nos métodos para obter e definir valores dos componentes da interface do usuário.

O Espresso é uma biblioteca de teste de instrumentação, construída com o

objetivo de realizar testes de interface do usuário (parte da Biblioteca de suporte a testes do Android), permitindo criar testes automatizados para seu aplicativo Android. Os testes são executados em dispositivos ou emuladores reais (são testes baseados em instrumentação) e se comportam como se um usuário real estivesse usando o aplicativo (DEITEL; DEITEL; WALD, 2016).

VOCÊ SABIA?

No artigo dos pesquisadores Pinto e Araújo (2016), podemos acompanhar um estudo de caso para testes automatizados com *frameworks* de código aberto em aplicativos móveis em sistema operacional Android, no nível de unidade e interface de sistema. O estudo demonstra uma análise crítica de cada ferramenta. Leia mais em <https://seer.cesjf.br/index.php/cesi/article/view/922 (https://seer.cesjf.br/index.php/cesi/article/view/922)>.

Sendo uma ferramenta simples e extensível, o Espresso permite a sincronização automática de ações de teste com a interface do usuário do aplicativo, fornecendo as informações sobre falhas, tornando-se uma opção para testes de interface do usuário. Para vários pesquisadores, como Deitel, Deitel e Wald (2016), o Espresso é considerado um substituto completo do Robotium.

Por fim, no caso do Espresso, a API principal é reduzida e fácil de aprender, mas permanece aberta para personalização. O Espresso testa as expectativas, interações e afirmações do estado sem a interação de conteúdo, infraestrutura personalizada ou detalhes complicados de implementação. Os testes do Expresso são executados de forma otimizada e rápida. Ele permite que deixe as sincronizações, pausas e pesquisas enquanto manipula e confirma a interface do usuário do aplicativo, quando está parado.

3.2.3 Android Debug Bridge

O Android *Debug Bridge* (ADB) é um programa cliente-servidor usado no desenvolvimento de aplicativos Android. O Android *Debug Bridge* faz parte do Android SDK e é composto de três componentes: um cliente, um daemon e um servidor. Ele é usado para gerenciar uma instância de emulador ou um dispositivo Android real.

Além do Android SDK, do qual a depuração do Android faz parte, os requisitos básicos de uma configuração de desenvolvimento do Android são um computador que cumpre os requisitos mínimos do sistema para executar o Android SDK e, na maioria dos casos, um dispositivo Android. Na linguagem de desenvolvimento de *software*, o computador mencionado é conhecido como a máquina de desenvolvimento. O componente cliente do Android *Debug Bridge* é executado na máquina de desenvolvimento. Ele pode ser chamado a partir do *prompt* de comando, usando o comando adb. Há também outras ferramentas que podem criar clientes adb, como o *plug-in* ADT (Android *Development Tools*) e o DDMS (Dalvik *Debug Monitor Service*). O daemon ADB, por outro lado, é executado como um processo de segundo plano em uma instância de emulador ou no próprio dispositivo. Na figura a seguir, podemos visualizar a estrutura do Android *Debug Bridge*.

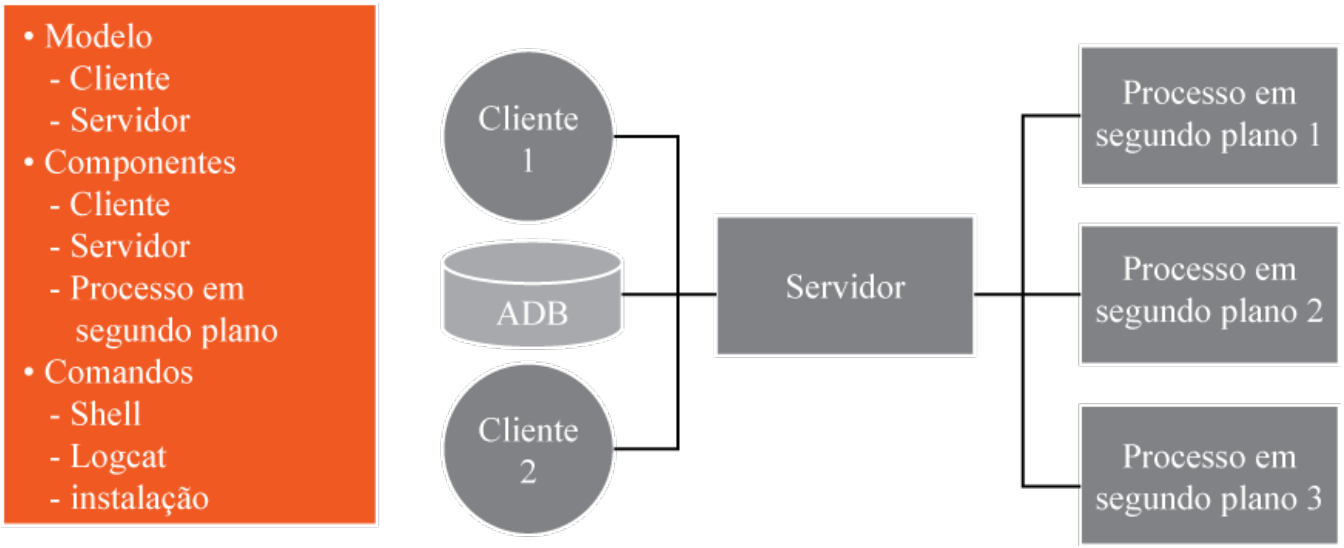


Figura 5 - Estrutura e componentes do Android Debug Bridge do sistema operacional Android. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta um quadro laranja à esquerda escrito: Modelo: - Cliente, - Servidor; Componentes: - Cliente, - Servidor, - Processo em segundo plano; Comandos: - Shell, - Logeat, - Instalação. A direira em cinza tem um quadro central escrito Servidor e conectado à ele sai uma linha horizontal à esquerda que que liga a uma linha vertical ligando-o a no topo um cículo cinza escrito Cliente 1, ligando no meio a um cilindro cinza claro escrito ADB, ligando abaixo a um círculo cinza escrito Cliente 2. O quadro central Servidor também é conectado por uma linha horizontal à direita que liga a uma linha vertical ligando-o a no topo um quadro

cinza escrito Processo em segundo plano 1, ligando no meio a um quadro cinza escrito Processo em segundo plano 2, ligando abaixo a um quadro cinza escrito Processo em segundo plano 3.

Finalmente, o componente servidor do ADB, que também é executado na máquina de desenvolvimento, mas apenas em segundo plano, se encarrega de gerenciar a comunicação entre o cliente ADB e o daemon ADB. Quando o Android *Debug Bridge* está ativo, o usuário pode emitir comandos adb, para interagir com uma ou mais instâncias do emulador.

O adb também pode executar várias instâncias do cliente adb, que podem ser usadas para controlar todas as instâncias de emuladores existentes. Uma forma de se usar o Android *Debug Bridge* é instalar o *plug-in* do ADT no IDE do Eclipse (*Integrated Development Environment*). Dessa forma, o desenvolvedor não precisaria inserir comandos por meio do *prompt* de comando. O usuário pode emitir comandos adb para interagir com uma ou mais instâncias de emulador.

3.3 Análise de performance

O nível de eficiência de um aplicativo está relacionado ao uso dos recursos de forma eficaz, à sua confiabilidade e ao nível de manutenibilidade do código. Pode-se considerar que estas variáveis estão relacionadas ao requisito de performance de um aplicativo. Isto significa a relação que existe, entre o tempo gasto para processar as funções, e a quantidade de processamentos realizados para executar o *software*.

Essas características não são percebidas pelos usuários de forma direta, mas o resultado final do uso do sistema será percebido como um todo, sendo mais ou menos eficiente. Nos aplicativos de dispositivos móveis, este requisito é ainda mais crítico, devido às limitações de processamento e memória (DEITEL; DEITEL; WALD, 2016).

VOCÊ QUER LER?

No artigo escrito por Zhao (*et al.*, 2016), foi proposta uma nova solução de serviço de *middleware* que supera as desvantagens do uso da abordagem pré-cache, a tecnologia PrecAche do sistema Android. O método proposto usa HTML para projetar a interface do aplicativo e armazenar separadamente o *Page Framework* e o *Page Data*. O artigo mostra como foi criado um novo *middleware* de páginas da *web*, o Version Flags, para indicar se o PF e o PD estão vencidos. Os resultados experimentais representam que a abordagem proposta pode melhorar a eficiência de execução, bem como reduzir os custos de rede, que podem ser amplamente utilizados em sistemas distribuídos, baseados em nuvem. Leia o artigo completo em: < https://www.sciencedirect.com/science/article/pii/S0167739X15001806 (https://www.sciencedirect.com/science/article/pii/S0167739X15001806)>.

Considerando isso tudo, como é possível executar a análise de desempenho de aplicativos de dispositivos móveis? Como são realizadas tais análises? Quais as ferramentas disponíveis?

Nos dispositivos móveis, requisitos como eficiência do código são um ponto fundamental que merece tratamento diferenciado. Isto se deve ao fato de que conexões com componentes externos, como banco de dados e aplicativos externos executados em *background*, mudam de comportamento e eficiência, dependendo do dispositivo móvel no qual o aplicativo está instalado.

3.3.1 Identificação de gargalos de performance por meio de ferramentas de análise

Inicialmente, um dos principais gargalos de um aplicativo é o tempo de inicialização do aplicativo, de acordo com Deitel, Deitel e Wald (2016), sendo uma métrica crítica que todo desenvolvedor deve monitorar com cuidado. Existe uma série de limitações nos dispositivos móveis, como a capacidade de armazenamento ou processamento limitado. Neste sentido, a conectividade entre dispositivos pode ser uma alternativa para estas limitações, podendo serem empregadas tecnologias de comunicação como HTTP, Socket TCP, Socket UDP e WebService, outras exclusivas do mundo móvel, como SMS e MMS.

O que ocorre é que um longo tempo de inicialização criará muito atrito entre o usuário e sua utilização do aplicativo. Isso é particularmente verdadeiro, quando se trata de aplicativos que devem ser usados rapidamente. Por exemplo, um usuário acessaria regularmente um aplicativo de transporte público se precisar de 15 segundos cada vez que quiser usá-lo?

Existem alguns aplicativos que oferecem funcionalidades para este tipo de serviço,

vamos listar a seguir.

- Nimbledroid é um serviço gratuito dedicado para medir a duração de inicialização do aplicativo Android. O principal valor agregado deste serviço é a medida do tempo de inicialização em um dispositivo real e seu monitoramento a médio / longo prazo.
- O AndroidDevMetrics é uma ferramenta incorporável diretamente no seu aplicativo (a versão de depuração). Ele fornecerá várias informações relacionadas ao desempenho, como: tempo de execução dos métodos do ciclo de vida da atividade, duração da instanciação do objeto Dagger 2.

No quadro a seguir, podemos entender as vantagens e desvantagens do AndroidDevMetrics.

AndroidDevMetrics	
Vantagens	Desvantagens
• Fácil integração • Existência de métricas para várias atividades	• Métricas somente para poucos métodos • Não há mecanismo de armazenamento das métricas

Figura 6 - Vantagens e desvantagens da ferramenta de análise AndroidDevMetrics do sistema operacional Android. Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta uma tabela onde a primeira linha é laranja e está escrito no meio da linha: AndroidDevMetries, embaixo dessa linha tem-se duas colunas uma encostada na outra. Na primeira coluna está escrito Vantagens: -Fácil integração, - Existência de métricas para várias atividades. Na segunda coluna está escrito desvantagens: - Métricas somente para poucos métodos, - Não há mecanismo de armazenamento das métricas.

Agora, vamos ver quais as ferramentas que tratam da otimização do desempenho da interface do usuário de um aplicativo, e como são utilizadas.

Existem muitas ferramentas que podem ajudar na identificação de problemas de interface do usuário e fornecer informações sobre os motivos do problema de desempenho.

- Visualizador *overdraw*: Esta ferramenta está disponível diretamente nas "opções do desenvolvedor" do dispositivo Android. É necessário apenas habilitar o campo "Debug GPU *overdraw*". Serão exibidas muitas cores sobrepostas na tela. Cada uma dessas cores informa quantas vezes a área correspondente foi retirada pela GPU. Uma cor transparente/ azul/ verde, corresponde a um baixo número de *overdraw* (respectivamente 0, 1 e 2), uma cor vermelha corresponde a um número de *overdraw* “alto” (3 ou 4).
- *Traceview*: quando é necessário fazer uma análise mais avançada, como por exemplo, fornecendo representações gráficas de logs de rastreio. É possível gerar os registros, instrumentando o código com a Debugclasse. Esse método de rastreamento é muito preciso, porque é possível especificar exatamente onde, no código, se deseja iniciar e parar o registro de dados de rastreamento. O Traceview faz parte do conjunto de ferramentas do Monitor de dispositivos Android fornecido pelo Android Studio.
- Logcat: pode parecer ser a ferramenta mais rudimentar, mas os resultados de logcats às vezes podem ser úteis para ter uma breve ideia de eventuais problemas de desempenho no aplicativo. O monitor do logcat tem como objetivo exibir as mensagens de alerta em tempo real, como, por exemplo, limpeza de memória. É possível criar filtros determinando quais informações devem ser exibidas, os níveis de prioridade e a busca pelo registro. O monitor exibe somente a saída de registro referente à execução mais recente do aplicativo. Quando um aplicativo aciona uma exceção, o monitor do logcat exibe uma mensagem, após o rastreamento de pilha, de que contém *links* para o código. Esse recurso auxilia na correção de erros e melhora o funcionamento do aplicativo.

Portanto, é fundamental identificar os gargalos e a análise de desempenho dos aplicativos, pois um alto tempo de resposta, pode prejudicar o desempenho do

aplicativo. Como, por exemplo, tem-se o Traceview como um visualizador gráfico para logs de execução que se cria usando a Debugclasse para registrar informações de rastreamento do código. O Traceview pode ajudar a depurar seu aplicativo e a analisar seu desempenho. Quando tiver um arquivo de log de rastreio (gerado ao incluir código de rastreio no aplicativo), poderá fazer com que Traceview carregue os arquivos de log e exiba os dados em uma janela para visualizar seu aplicativo em dois painéis: um painel de cronograma – que descreve quando cada *thread* e método foi iniciado e interrompido –, e um painel de perfil – que fornece um resumo do que aconteceu dentro de um método.

3.3.2 Memória e otimização da performance

A memória de acesso aleatório (RAM) é um recurso importante em qualquer ambiente de desenvolvimento de *software*, mas é ainda mais valiosa em um sistema operacional móvel, no qual a memória física é frequentemente restrita. Embora o Android Runtime (ART) e a máquina virtual Dalvik executem a coleta de lixo de rotina, isso não significa que se possa ignorar quando e onde seu aplicativo aloca e libera memória. É necessário, ainda, evitar a introdução de vazamentos de memória, geralmente causados por manter referências de objetos em variáveis de membros estáticos e liberar quaisquer referências no momento apropriado, conforme definido por retornos de chamada de ciclo de vida.

Os dispositivos Android têm menos energia que os *desktop*s ou notebooks padrão. Especialmente em dispositivos anteriores ao Android 5.0, é necessário evitar o acionamento do coletor de lixo da Java Virtual Machine. Isso resulta em um congelamento do tempo de execução do Android por cerca de 200 ms. Este pode ser um atraso notável, se o usuário estiver, por exemplo, rolando uma lista.

Deve-se evitar a criação de objetos desnecessários, sendo aconselhável reutilizar os objetos, por exemplo, evitando criar objetos em *loop*s ou no onDraw(), que é o método de exibição personalizada.

Por outro lado, o Android fornece estruturas de dados que são mais eficientes para mapear valores para outros objetos. Outro fator importante de otimização é quando não é necessário acessar os campos de um objeto, o melhor é torná-lo um método estático. As invocações serão cerca de 15% a 20% mais rápidas. Uma observação importante sobre a otimização é usar a sintaxe aprimorada do *loop* "*for*". O "*for*" *loop* aprimorado (também conhecido como *loop* "*for-each*") pode ser usado para coleções que implementam a "*Iterable*" interface e para matrizes. Um

iterador é alocado para fazer chamadas de interface para hasNext() e next(). Com um ArrayList, um *loop* contado escrito à mão é cerca de três vezes mais rápido, mas para outras coleções a sintaxe aprimorada do *loop for* será exatamente equivalente ao uso explícito do iterador.

3.4 Ferramentas de automatização de *builds*

O conceito fundamental de um *build* significa que as tarefas de compilação, empacotamento e testes (sejam unitários ou de interface), as coletas de métricas, validações de código e a verificação de dependências, como também a geração de relatórios e documentação são realizadas como um processo, que poderia ser melhorado se fosse realizado de forma automática. Isto torna a tarefa de desenvolvimento mais ágil e com menor índice de erro. Como é possível fazer a automatização de *build*s em aplicativos móveis? Quais são as ferramentas disponíveis? Como executar o *build* no sistema operacional Android? Estas questões serão discutidas a seguir.

3.4.1 Processo de building *do Android com o Gradle*

O sistema de criação do Android Studio é baseado no Gradle, e o *plug-in* do Android para o Gradle adiciona vários recursos específicos para a criação de aplicativos para Android. Embora o *plug-in* do Android seja normalmente atualizado na etapa de bloqueio com o Android Studio, o *plug-in* (e o restante do sistema Gradle) pode ser executado independentemente do Android Studio e ser atualizado separadamente.

VOCÊ QUER VER?

Na palestra de Felipe Lima (*Software Engineer* da Airbnb), você pode conhecer a evolução do desenvolvimento Android no serviço Airbnb. A palestra foi proferida na Conferência Android Dev de 2017. Para assistir acesse <https://www.youtube.com/watch?v=StyOQUWPClE (https://www.youtube.com/watch?v=StyOQUWPClE)>.

O Gradle é disponibilizado pelo sistema operacional Android. Ele tem a função de automatizar o processamento de *build* de forma avançada. Ele une a flexibilidade do Ant e executa o processamento de dependências do Maven. O Android ANT SDK permite que os desenvolvedores conectem aplicativos Android a dispositivos ANT. Ele é fornecido para permitir que você se conecte a dispositivos proprietários e construa topologias complexas utilizando recursos ANT avançados, como a varredura em segundo plano e aplicativos de telefone celular. Portanto, permite que os aplicativos se estendam a todos os fornecedores de *smartphones* sem fragmentação nas versões do Android.

Já O *Plug-in* do Android Maven é usado para criar aplicativos para o sistema operacional Android, podendo criar bibliotecas para serem usadas no formato legado APKLIB usando o Apache Maven. Enquanto os formatos de construção do Maven utilizam arquivos XML para a configuração do *build*, os arquivos de *build* do Gradle são escritos na linguagem Groovy. Neste sentido, pelo fato de serem baseados em *script*, os arquivos do Gradle permitem que nos arquivos de configuração sejam desenvolvidas tarefas de programação. Além disto, o Gradle tem um sistema de *plug-ins* que fazem a adição de novas funcionalidades.

3.4.2 Processo de building *automatizado com o Jenkins*

Jenkins é um servidor ou ferramenta de Integração Contínua (CI) que é escrito em Java. Ele fornece serviços de Integração Contínua para desenvolvimento de *software*, que podem ser iniciados via linha de comando ou servidor de aplicativos da *web*.

O Jenkins suporta a criação e teste automáticos de aplicativos Android. Para criar um trabalho de construção no Jenkins, é necessário ter uma configuração de construção funcional. O conjunto de ferramentas do Android gera automaticamente um arquivo de criação Gradle válido, que pode ser usado no Jenkins.

VOCÊ O CONHECE?

O criador do Java, James Gosling, quando adolescente, teve a ideia de criar um pequeno intérprete para resolver um problema em um projeto de análise de dados em que estava trabalhando na época. Ao longo dos anos, como aluno de pós-graduação e na Sun, como criador de Java e da Java Virtual

Machine, ele usou variações dessa solução. Leia mais sobre o criador do Java no artigo escrito por Allman (2004), disponível em <https://queue.acm.org/detail.cfm?id=1017013 (https://queue.acm.org/detail.cfm?id=1017013)>.

Na prática de Integração Contínua (CI), os desenvolvedores devem integrar o código em repositório, para compartilhamento constante.

É o processo de executar os testes em uma máquina que não seja a do desenvolvedor, automaticamente, quando é enviado um novo código para o repositório de origem. Na figura a seguir é mostrada a Integração Contínua no sistema operacional Android.

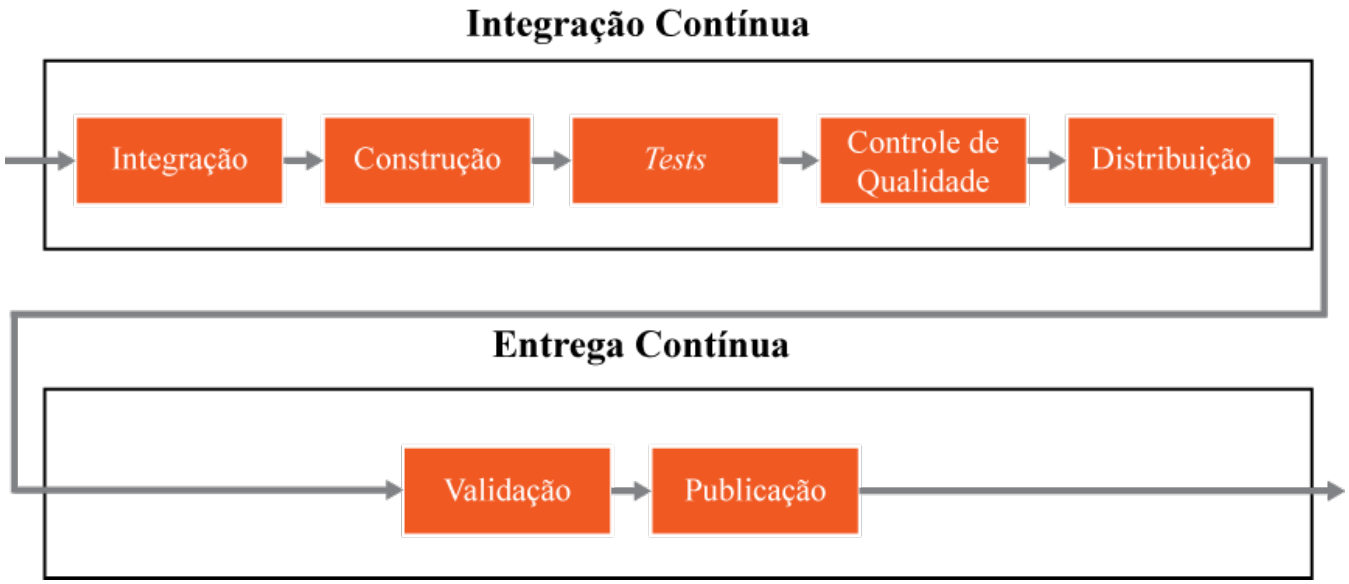


Figura 7 - Estrutura da Integração Contínua do processo automatizado utilizando a ferramenta Jenkins.

Fonte: Elaborado pelo autor, 2018.

#PraCegoVer: Apresenta uma estrutura dividida em duas partes em formato de retangulo na horizontal um em cima e um embaixo. O primeiro retângulo (em cima) é a Integração Contínua, dentro dele começa com uma seta apontada para frente para um quadro laranja escrito Integração, em seguida uma seta apontada para frente para um quadro laranja escrito Construção, em seguida uma seta apontada para frente para um quadro laranja escrito Tests, em seguida uma seta apontada para frente para um quadro laranja escrito Controle de Qualidade, em seguida uma seta apontada para frente para um quadro laranja escrito Distribuição. Do quadro distribuição sai uma seta que aponta para o início do

retângulo embaixo, esse retângulo é chamado de Entrega contínua e dentro dele tem-se o quadro laranja escrito Validação e em seguida uma seta apontada para frente para um quadro laranja escrito Publicação e em seguida uma seta cinza aponta para frente até o final deste retêngulo.

Nesse tipo de procedimento, há uma enorme vantagem de saber se todos os trabalhos (projeto configurado no Jenkins) funcionam corretamente ou não. E também é possível obter *feedback* rápido. O *feedback* rápido é muito importante, assim sempre saberá, logo depois de ter sido interrompida ou finalizada a compilação. No *console* serão recebidas as mensagens de log detalhadas. A partir disso, se saberá qual foi a razão da falha no trabalho e também poderá saber como pode reverter isso.

CASO

O mercado consumidor está cada vez mais exigente com relação aos aplicativos para dispositivos móveis e suas funcionalidades e, para não perder oportunidades, as empresas devem estar atentas para o que o público espera, a fim de atender a essas demandas. Um grande escritório de advocacia percebeu um nicho de utilização de aplicativos: a capacidade de realizar as sessões de conciliação de processos virtualmente. Para isto, a empresa solicitou o desenvolvimento de um aplicativo que permite que as partes envolvidas no processo, possam fazer uma videoconferência, utilizando o aplicativo no dispositivo móvel. Isto agiliza as decisões, as torna mais eficientes e reduz custos de deslocamento para as partes envolvidas e os advogados.

Quando se faz as alterações manualmente, é mais complexo analisar o impacto das alterações no código. Por isso, será difícil descobrir quais alterações introduziram o problema. Mas quando o Jenkins é configurado para rodar automaticamente em cada envio de código para o repositório, é sempre fácil saber o que e, quem, introduziu o problema (DEITEL, DEITEL; WALD, 2016). Na lista a seguir, podemos ver algumas das razões pelas quais é preciso automatizar o teste e as integrações de *build.*

- **O tempo do desenvolvedor é concentrado no trabalho que importa**:

a maior parte do trabalho, como integração e teste, é gerenciada por sistemas automatizados de criação e teste.
- **A qualidade do *software* é aprimorada**: os problemas são detectados e resolvidos quase imediatamente, o que mantém o *software* em um estado em que ele pode ser liberado a qualquer momento com segurança.
- **Torna o desenvolvimento mais rápido**: a maior parte do trabalho de integração é automatizada. Por isso, os problemas de integração são menores. Isso economiza tempo e dinheiro ao longo da vida útil de um projeto.

O Sistema de Construção Contínua pode incluir ferramentas como Jenkins, Bamboo ou Cruise Control. O Bamboo tem melhor suporte a interface do usuário, mas não é uma ferramenta gratuita. O Jenkins é uma ferramenta de código aberto, mais fácil de configurar e também possui uma comunidade de desenvolvimento de *plug-ins* muito ativa, que o torna favorecido. Vantagens do Jenkins:

- Jenkins é uma ferramenta de código aberto com muito apoio de sua comunidade;
- a instalação é mais fácil;
- tem mais de 1000 *plug-ins* para facilitar o trabalho;
- é fácil criar um novo *plug-in* Jenkins, se não houver um disponível;
- é uma ferramenta que está escrita em Java. Por isso, pode ser portável para quase todas as principais plataformas.

Portanto, existem ferramentas muito interessantes para tornar o processo de desenvolvimento e implantação de aplicativos de forma automatizada, tornando o trabalho mais profissional, com melhores recursos e bom nível de produto final.

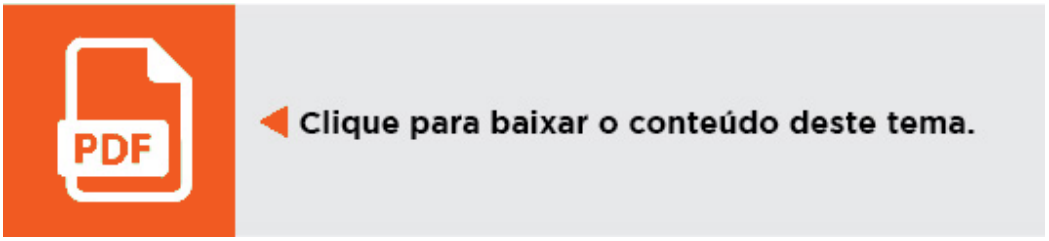
Síntese

Aprendemos aqui, como desenvolver aplicativos com qualidade para dispositivos

móveis no sistema operacional Android. É muito importante para os desenvolvedores utilizarem padrões de projeto específicos para aplicativos em dispositivos móveis, como o MVC e MVVM, pois estes padrões garantem a eficiência da arquitetura do *software*. É também muito relevante saber utilizar os testes, depuração e a análise de desempenho, juntamente com a criação automática de códigos, pois estas atividades mantêm o nível de qualidade dos aplicativos.

Neste capítulo, você teve a oportunidade de:

- utilizar os padrões de projeto por meio do Databinding, arquitetura MVP e MVVM e Dagger 2.
- aplicar os testes e depuração por meio de testes unitários utilizando ferramentas JUnit, Robotium e Espresso.
- identificar os gargalos de desempenho utilizando ferramentas de análise e otimização de desempenho.
- utilizar ferramentas de automatização para desenvolver aplicativos mais eficientes por meio das ferramentas Gradle e Jenkins.



Bibliografia

ALLMAN, E. A conversation with James Gosling. **acmQueue**, Vol.2(5), pp.24-33, July/August, 2004. Disponível em: <https://queue.acm.org/detail.cfm?id=1017013 (https://queue.acm.org/detail.cfm?id=1017013)>. Acesso em: 04/07/2018.

BARBOSA, N. A. **Padrões de projeto de interface para aplicativos Android para o Google TV**. Universidade Estadual de Maringá. Desenvolvimento de Sistemas para *Web*. 2013. Disponível em: < https://silo.tips/download/universidade-estadual-de-maringa-centro-de-tecnologia-departamento-de-informatic-51 (https://silo.tips/download/universidade-estadual-de-maringa-centro-de-tecnologia-departamento-de-informatic-51)>. Acesso em: 04/07/2018.

DEITEL, P. J.; DEITEL, H.; WALD, A. **Android 6 para Programadores**: uma

abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. Disponível na Biblioteca Virtual Ânima: <https://Ânimabrasil.blackboard.com/ (https://Ânimabrasil.blackboard.com/webapps/blackboard/content /listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)*web (https://Ânimabrasil.blackboard.com/webapps/blackboard/content /listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)*apps /blackboard/content/listContent.jsp?course_id=_198689_1& content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com /webapps/blackboard/content/listContent.jsp?course_id=_198689_1& content_id=_4122211_1&mode=reset)>. Acesso em: 04/07/2018.

DEITEL, P. J.; DEITEL, H.; WALD, A. **Android 6 para Programadores**: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. Disponível na Biblioteca Virtual Ânima: <https://Ânimabrasil.blackboard.com/ (https://Ânimabrasil.blackboard.com/webapps/blackboard/content /listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)*web (https://Ânimabrasil.blackboard.com/webapps/blackboard/content /listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)*apps /blackboard/content/listContent.jsp?course_id=_198689_1& content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com /webapps/blackboard/content/listContent.jsp?course_id=_198689_1& content_id=_4122211_1&mode=reset)>. Acesso em: 04/07/2018.

PICHILIANI, M. C.; HIRATA, C. M. Adaptation of Single-user Multi-touch Components to Support Synchronous Mobile. **Collaboration Mobile Networks and Applications**, October 2014, Volume 19, Issue 5, pp 660–679. Disponível em: <https:// (https://link.springer.com/article/10.1007/s11036-014-0512-0)*link (https://link.springer.com/article/10.1007/s11036-014-0512-0)*.springer.com/article /10.1007/s11036-014-0512-0 (https://link.springer.com/article/10.1007 /s11036-014-0512-0)>. Acesso em: 04/07/2018.

PINTO, S. V; ARAÚJO, M. A. P. Análise comparativa de ferramentas para testes em aplicativos Android. **Caderno de Estudos em Sistemas de Informação**, v. 3, n. 2, 2016. Disponível em: <https://seer.cesjf.br/index.php/cesi/article/view/922 (https://seer.cesjf.br/index.php/cesi/article/view/922)>. Acesso em: 04/07/2018.

SAWADA, Y.; *et al.* Performance of Android Cluster System Allowing Dynamic Node Reconfiguration. **Wireless Personal Communications**. April 2017, Volume 93, Issue 4, pp 1067–1087. Disponível em: <https:// (https://link.springer.com/article

/10.1007/s11277-017-3978-9)*link (https://link.springer.com/article/10.1007/s11277-017-3978-9)*.springer.com/article/10.1007/s11277-017-3978-9 (https://link.springer.com/article/10.1007/s11277-017-3978-9)>. Acesso em: 04/07/2018.

TANENBAUM, A. S.; BOS, H. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Person Education do Brasil, 2016.

ZHAO, H.; *et al.* A novel pre-cache schema for high performance Android system. **Future Generation Computer Systems**, March 2016, Vol. 56, pp. 766-772. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X15001806 (https://www.sciencedirect.com/science/article/pii/S0167739X15001806)>. Acesso em: 04/07/2018.