Desenvolvido por [Paulo Salvatore]

Movile Next Program
Formação em Desenvolvimento *Android*
Prof. Paulo Salvatore
Tópicos abordados:

- ***Architecture Principles***
  - ***SOLID***
  - ***Clean Architecture***
- ***Testing (JUnit 4, Espresso, UI Automator, Mockito* e *Robot Pattern)***
- ***ProGuard***
- ***Git Flow***

1. *Movile Next Program*

O *Movile Next* é uma iniciativa criada pelo Grupo *Movile*, em parceria com a *GlobalCode*, para capacitar desenvolvedores *Android*, *iOS* e *Backend* de nível pleno e sênior e possibilitar um avanço em carreira. Durante 4 sábados acontecerão, simultaneamente, as formações de cada especialidade com aulas presenciais ministradas por especialistas do mercado, além de atividades e desafios complementares online durante a semana.

2. Introdução

Nesse material iremos concluir alguns assuntos importantes no desenvolvimento de aplicações de alto nível, finalizando o capítulo de princípios de arquitetura que foram passados aos poucos em cada aula. Falaremos sobre os princípios *SOLID* e também sobre a *Clean Architecture*, ambos sintetizados por *Robert C. Martin*, grande colaborador para o desenvolvimento de grandes aplicações.

Iniciaremos a abordagem de *testing*, apresentando as principais maneiras para a realização dos testes, os propósitos e as possibilidades dentro do universo *Android* para construção de um bom sistema de testes. Resumidamente toda aplicação de qualidade geralmente é refletida por uma grande quantidade de testes. Muitas vezes subestimamos o real poder da utilização dos testes automatizados mas com o tempo entendemos o real ganho em produtividade e tempo, além da qualidade da aplicação desenvolvida aumentada drasticamente pois as falhas são detectadas logo no início.

Um outro assunto importante que será abordado é o *ProGuard*, uma ferramenta que otimiza o arquivo compilado das nossas aplicações e os protege da engenharia reversa, uma prática maliciosa que desenvolvedores usam para acessar o código-fonte de uma aplicação e acessar as regras de negócio. Entenderemos as

principais ferramentas utilizadas para a engenharia reversa e a importância de manter o *ProGuard* ativado no projeto. Também abordaremos os principais problemas encontrados na utilização dessa ferramenta e como contornar alguns deles, entendendo melhor seu funcionamento para conseguir utilizá-lo com mais eficiência.

Por fim, abordaremos a utilização do *git flow*, uma ferramenta que se faz extremamente necessária para controle da aplicação no sistema de versionamento *git*, principalmente em projetos com várias pessoas envolvidas ou projetos *open source* onde pessoas desconhecidas pretendem entender o seu código para melhorá-lo ou otimizá-lo. Saber utilizar ferramentas que aumentam a produtividade e a organização são habilidades essenciais para qualquer desenvolvedor.

3. Sumário

4. Architecture Patterns

Finalizando nossa série de padrões de arquitetura, falaremos sobre alguns princípios fundamentais em qualquer desenvolvimento, que tem como objetivo melhorar o resultado de nossos códigos e *softwares* e principalmente melhorar o fluxo de desenvolvimento de uma aplicação, reduzindo a possibilidade de problemas futuros e melhorando nosso desempenho individual e coletivo.

4.1. SOLID

O termo *SOLID* representa cinco princípios com o objetivo a fazer o desenvolvimento de *softwares* algo mais fácil de entender, flexível e fácil de manter. Os princípios são um subconjunto de muitos princípios promovidos por *Robert C. Martin*, conhecido como *Uncle Bob* e citado diversas vezes na produção dessa série de apostilas. O termo é comumente apresentado sob o tópico de desenvolvimento ágil e deve receber a devida atenção pois dita uma série de regrinhas básicas para o bom funcionamento do desenvolvimento de qualquer aplicação [ref 1].

Para mais informações sobre os princípios e ver mais exemplos, incluindo alguns direcionados ao universo *Android*, sugiro dar uma olhada na leitura adicional sobre o assunto.

4.1.1. Single responsibility principle

O princípio da responsabilidade única, como o próprio nome diz, afirma que uma classe deve possuir apenas um único motivo para ser alterada, ou seja, possuir apenas uma responsabilidade.

Quando analisamos uma classe e nos perguntamos o que ela realmente faz, se a resposta for: "faz isso **e** isso **e** isso **e**..." temos uma infração desse princípio. Nesse caso, é recomendado que cada uma dessas funcionalidades sejam separadas em outras classes. Isso convém na hora de fazer uma alteração, onde iremos procurar exatamente a classe que realiza aquela função para que ela seja alterada, agilizando esse processo.

4.1.2. Open/closed principle

O princípio do aberto/fechado diz que uma classe deve estar aberta para extensão e fechada para modificação. A melhor forma de entender esse princípio é visualizando um exemplo prático dele.

Supondo que temos uma classe chamada 'Forma' que será estendida por outras classes que representam formas geométricas, como um quadrado ou um círculo.

```
open class Forma {
  var tipo: Int = 0
}

class Quadrado : Forma()
class Circulo : Forma()
```

Para exibir essas formas, precisamos de uma classe chamada 'EditorGrafico', que saberá como criar cada forma recebida.

```
class EditorGrafico {
  fun desenharForma(forma: Forma) {
     if (forma is Quadrado) {
        desenharQuadrado()
     } else if (forma is Circulo) {
        desenharCirculo()
     }
  }

  private fun desenharQuadrado() {
     println("Quadrado")
  }

  private fun desenharCirculo() {
     println("Circulo")
  }
}
```

Agora surge a primeira necessidade do sistema, incluir uma nova forma. No exemplo de código acima, é impossível adicionar uma nova forma sem modificar a classe 'EditorGrafico', quebrando o princípio do aberto/fechado. Portanto, nossa classe 'Forma' deveria possuir um método chamado 'desenhar' que permitisse que sua implementação fosse diferente para cada forma.

```
abstract class Forma {
  var tipo: Int = 0

  abstract fun desenhar()
}

class Quadrado : Forma() {
  override fun desenhar() {
     println("Quadrado")
  }
}

class Circulo : Forma() {
  override fun desenhar() {
     println("Circulo")
  }
}
```

Com isso, podemos facilmente desenhar qualquer forma, sem que a classe 'EditorGrafico' precise saber qual forma está sendo passada.

```
class EditorGrafico {
  fun desenharForma(forma: Forma) {
     forma.desenhar()
  }
}
```

Por exemplo, se quisermos desenhar um retângulo, podemos simplesmente criar a classe retângulo e informar como ele deve ser desenhado.

```
class Retangulo : Forma() {
  override fun desenhar() {
     println("Retângulo")
  }
}
```

4.1.3. Liskov substitution principle

O princípio de substituição de *Liskov* é um pouco complicado na leitura mas é mais fácil de entender na exemplificação. Resumindo esse princípio ele diz que qualquer classe filha deve permitir que todos os métodos da sua classe pai possam ser chamados.

```
open class Passaro {
  open fun comer() { }

  open fun voar() { }
}

class Corvo : Passaro()

class Avestruz : Passaro() {
  override fun voar() {
     throw Exception("Avestruz não pode voar.")
  }
}
```

Esse exemplo infringe claramente o princípio citado pois ao chamar o método 'voar()' para um avestruz, uma exceção é causada, mesmo que a princípio quando estamos chamando não esperamos esse resultado. O certo nesse caso seria separar a classe 'Passaro' em duas subclasses, pássaros que voam e que não voam, pois ambos comem mas nem todos voam.

```
open class Passaro {
  open fun comer() { }
}

open class PassaroVoador : Passaro() {
  open fun voar() { }
}

open class PassaroNaoVoador : Passaro()
```

```
class Corvo : PassaroVoador()

class Avestruz : PassaroNaoVoador()
```

4.1.4. Interface segregation principle

O princípio de segregação de interface diz que é melhor ter muitas interface específicas do que uma genérica. Isso evita com que tenhamos diversos métodos que precisam ser implementados mas que não são necessários.

```
button.setOnClickListener(new View.OnClickListener {
  public void onClick(View v) {
  }

  public void onLongClick(View v) {
  }

  public void onTouch(View v, MotionEvent event) {
  }
});
```

Uma melhor aplicação seriam interfaces separadas para cada ação, onde chamaremos apenas as que precisarmos.

```
public interface OnClickListener {
  void onClick(View v);
}
public interface OnLongClickListener {
  void onLongClick(View v);
}
public interface OnTouchListener {
  void onTouch(View v, MotionEvent event);
}
```

4.1.5. Dependency inversion principle

O princípio de inversão de dependência diz que uma determinada parte do código deve depender de abstrações e não de algo concreto, por exemplo, para executar um determinado método, não podemos depender de outras classes para casos opcionais, pois isso fará com que tornamos dependentes dessa classe. A dependência deve ocorrer apenas em casos em que é absolutamente obrigatória para todo o contexto da classe.

Supomos que temos uma classe 'Copiadora' que possui um método 'copiar()' que recebe um texto digitado pelo usuário e o exibe na tela.

```
class Copiadora {
  fun copiar() {
     var mensagem: String? = null

     while (mensagem != "q") {
        mensagem = readLine()
        println(mensagem)
     }
  }
}
```

Agora vamos supor que desejamos que o usuário decida se a mensagem digitada será enviada para seu *clipboard (Ctrl + C)* ou se irá ser impressa. Fazemos isso preparando o método 'copiar()' para receber uma variável booleana que indicará o que o método deve fazer.

```
class Copiadora {
  fun copiar(enviarParaClipboard: Boolean) {
     var mensagem: String? = null

     while (mensagem != "q") {
        mensagem = readLine()

        if (enviarParaClipboard) {
           // Enviar mensagem para clipboard
        } else {
```

```
            println(mensagem)
        }
    }
  }
}
```

Com isso estamos violando o princípio, pois agora essa classe é obrigada a saber como realizar a tarefa de copiar para o *clipboard* mesmo que isso não vá ser executado em todas as chamadas. Para contornar esse problema, podemos criar uma interface 'Escritora' com duas implementações diferentes, que recebe a mensagem em questão e realiza a escrita conforme foi programada para fazer.

```
interface Escritora {
  fun escrever(mensagem: String)
}

class EscritoraConsole : Escritora {
  override fun escrever(mensagem: String) {
     println(mensagem)
  }
}

class EscritoraClipboard : Escritora {
  override fun escrever(mensagem: String) {
     // Copiar mensagem para clipboard
  }
}
```

4.2. Clean Architecture

Finalizando nosso assunto sobre arquiteturas não poderia ficar sem falar da *Clean Architecture* [ref 2], também proposta pelo *Robert C. Martin* e um dos conceitos mais atuais de arquitetura, também muito utilizado no universo *Android* e com vários artigos sobre. A representação clássica da arquitetura pode ser visualizada na figura 1.

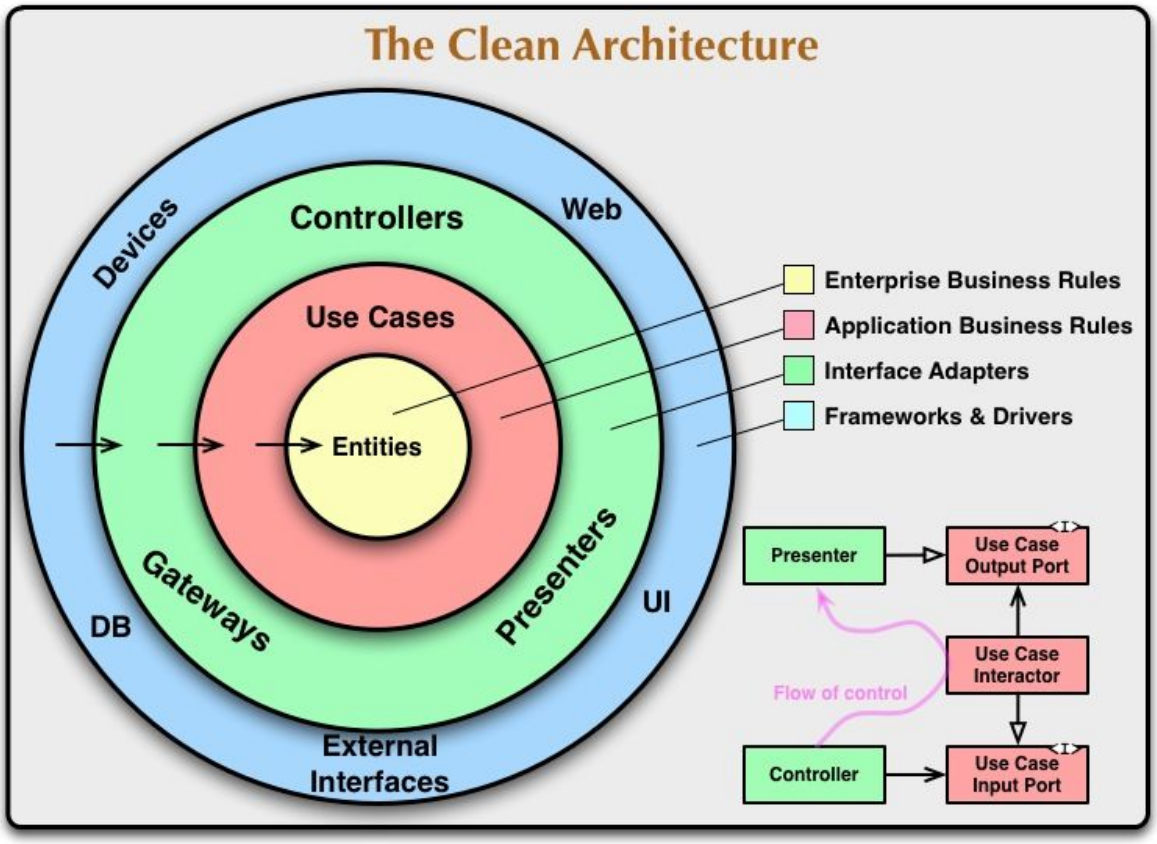


Figura 1: Representação clássica de *Clean Architecture*
Fonte: Uncle Bob [ref 2]

Apesar de início parece bastante coisa e um tanto complicada, a ideia por trás é bem simples e apresenta um conjunto de práticas cuja proposta é desenvolver um sistema que seja:

- **Independente de Frameworks:** A arquitetura não deve depender de alguma biblioteca, sendo que elas devem ser apenas usadas como ferramentas, sem que o sistema dependa delas para existir.
- **Testável:** Qualquer regra de negócio deve poder ser testada independente de qualquer agente externo como interface de usuário, banco de dados, *webservice*, entre outros.
- **Independente da UI:** Deve ser possível substituir facilmente a interface de usuário atual sem perder as regras de negócios. Por exemplo, se queremos remover o *Android* e passar a utilizar uma *console application* para utilização da ferramenta isso deve ser possível ser muito esforço.

- **Independente do banco de dados:** A aplicação não deve depender de um banco de dados específico. Podemos remover nosso *SQLite* por um *MongoDB* sem que as regras de negócio sejam alteradas durante esse processo.
- **Independente de qualquer agente externo:** As regras de negócio não podem saber nada no mundo exterior, não interessa se a informação que chega até elas veio de um banco de dados, de um *webservice* ou de um arquivo *txt*, elas saberão como a informação deve estar estruturada e saberá trabalhar com isso.

4.2.1. Dependency Rule

A regra suprema que faz essa arquitetura funcionar é a regra de dependência, que dita que as dependências do sistema só devem ser direcionadas para dentro do círculo, afirmando que nada de dentro do círculo deve saber ou conhecer as informações das partes exteriores dele. Em outras palavras, uma classe responsável por realizar a comunicação com o banco de dados não deve conter nenhuma referência sobre o *controller* que está esperando a informação ser processada.

Do mesmo jeito, estruturas de dados usadas em componentes nos círculos externos não devem ser as mesmas utilizadas pelo círculos internos, especialmente se essas estruturas foram geradas por *frameworks* específicos dos círculos externos. Isso fará com que o círculo exterior impacte diretamente a maneira que um círculo interior está construído, quebrando o princípio.

4.2.2. Entidades

As entidades são responsáveis pelas regras de negócio da informação, podendo ser um objeto com métodos ou apenas uma estrutura de dados e funções, o importante é que ela possa ser usada por diferentes aplicações no contexto geral.

Elas englobam as regras gerais e de alto nível e são as menos propensas a mudar quando algo externo muda, por exemplo, não deve ser necessário alterar algum objeto da entidade quando o sistema de navegação entre as páginas muda ou quando alguma forma com o sistema opera foi alterada.

4.2.3. Casos de uso

A aplicação nesta camada consiste em diversos adaptadores que trabalharão com os dados para deixá-los no formato mais conveniente para os casos de uso e

as entidades, por exemplo, para utilização de algum agente externo como banco de dados ou algum *webservice*.

É nessa camada que está localizada toda a arquitetura *MVC*, ou no caso do *MVP*, os *presenters*, *views* e *controllers*. Os *models* provavelmente são apenas estruturas de dados que são enviadas dos *controllers* aos casos de uso, que em seguida retornam desses para que sejam utilizados pelos *presenters* e *views*.

4.2.4. Frameworks e Drivers

A camada mais externa da aplicação são os *frameworks* e os *drivers*, que são ferramentas como *databases*, *web services*, onde geralmente não escrevemos muito código nesta camada, apenas o necessário para conectá-la com as camadas mais internas. Todas as informações dessas camadas são mantidas na parte externa para que façam o menor mal possível para a aplicação e para que sejam facilmente substituídas.

5. Testing

Na medida que nossa aplicação ganha tamanho, é necessário implementar cada vez mais funcionalidades específicas e fazer tudo isso interagir da melhor maneira possível, evitando vários erros no meio do caminho. A versatilidade da aplicação demanda uma boa estratégia de testes, apontando possíveis problemas antes mesmo de um usuário precisar testar todas as funcionalidades.

Neste capítulo, entenderemos a importância da realização de testes, modelos de desenvolvimento orientados à construção de testes e veremos as principais bibliotecas disponíveis que irão nos auxiliar durante esse processo.

Antes de iniciar a leitura do conteúdo deste capítulo, recomendo assistir o vídeo 1 (ir para vídeo), que contém muita informação interessante sobre *testing* de uma perspectiva dos desenvolvedores do *Google* que desenvolvem diversas aplicações diariamente e mantém uma série de bibliotecas atualizadas, muitas das quais são amplamente utilizadas por nós, desenvolvedores.

5.1. Fundamentos dos Testes

Qualquer aplicativo *mobile* possui diversos pontos de interação com o usuário, desde apertar um botão até baixar um documento da internet. Independente da interação, devemos testar uma série de casos de uso e interações possíveis conforme desenvolvemos nossa aplicação. [ref 3]

Uma boa estratégia de testes está atrelada a um desenvolvimento *clean* da aplicação, utilizando os conceitos mais atuais no desenvolvimento, se preocupando

com que cada parte da arquitetura esteja devidamente desacoplada do resto do sistema visando um melhor fluxo dos testes e do desenvolvimento como um todo. Bons exemplos de padrões em testes podem ser visualizados no vídeo 2, que é uma *playlist* pequena e rápida produzida pelo *google* mas extremamente útil.

5.2. Categorias de Testes

5.2.1. Unit Tests

Quando vamos desenvolver uma funcionalidade, começamos escrevendo um novo teste de unidade ou adicionando casos e checagens em um teste existente. O teste irá falhar no início pois a funcionalidade ainda não está implementada.

Um teste de unidade (*unit tests*) deve consistir a maior parte dos testes da aplicação e visa o teste das funcionalidades isoladas, utilizando o conceito de *mock* para qualquer outro componente relacionado e deve ser executado rapidamente na máquina de desenvolvimento.

Esses testes são considerados **@SmallTests**, devido a sua rapidez de execução e simplicidade na implementação. Uma boa proporção para o número de testes de unidade em relação ao restante da aplicação é de **70%**, conforme pode ser visto na figura 2.

No *Android*, eles devem estar localizados no pacote da aplicação marcado com ***(test)***.

5.2.2. Integration Tests

Os testes de integração (*integration tests*) são o próximo passo em relação aos testes de funcionalidades de uma aplicação. O início do desenvolvimento desses testes geralmente ocorre após a implementação e execução dos testes de unidade e geralmente testam a integração de partes diferentes do sistema, utilizando um dispositivo real ou emulado. São parte dos *instrumentation tests*.

Esses testes são considerados **@MediumTests**, devido a necessidade de execução no próprio ambiente do dispositivo e aguardando os tempos de resposta necessários. Uma boa proporção para o número de testes de integração em relação ao restante da aplicação é de **20%**, conforme pode ser visto na figura 2.

No *Android*, eles devem estar localizados no pacote da aplicação marcado com ***(androidTest)***.

5.2.3. UI Tests

Por último, os testes de interface de usuário (*UI Tests*) consistem nos testes de checagens mais delicadas do sistema, executando todo o *workflow* da *UI*, garantindo que todas as principais tarefas que o usuário final executará estão funcionando como esperado em um dispositivo real ou emulado. Também são parte dos *instrumentation tests*.

Esses testes são considerados **@LargeTests**, devido além da necessidade de execução no próprio ambiente do dispositivo, o tempo de resposta demorado de possíveis requisições ou da própria execução de um fluxo maior de comandos. Uma boa proporção para o número de testes de *UI* em relação ao restante da aplicação é de **10%**, conforme pode ser visto na figura 2.

No *Android*, eles devem estar localizados no mesmo pacote dos testes de integração.

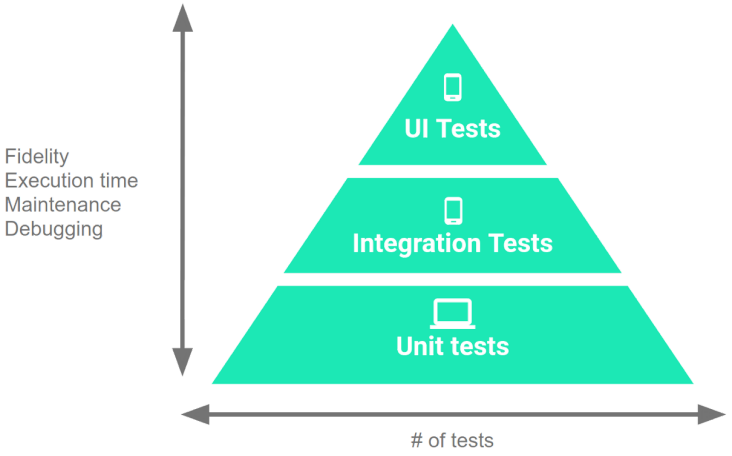


Figura 2: Pirâmide de Testes
Fonte: Android Doc [ref 3]

5.3. Test-Driven-Development (TDD) + ATSL

O *Test-Driven-Development* (*TDD*) [ref 4] é uma técnica utilizada durante o desenvolvimento de aplicações que está diretamente relacionada ao conceito de verificação e validação e se baseia em um ciclo de desenvolvimento, como pode ser observado na figura 3.

Primeiro o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade. Depois, é produzido o código

que será necessário para que o teste possa validar para posteriormente ser refatorado para algo nos padrões aceitáveis.

Esse conceito é bastante utilizado no universo *Android* e possui diversas abordagens possíveis, com vários padrões que visam facilitar a escrita dos testes visando encorajar os desenvolvedores a utilizarem os testes automatizados para testarem as funcionalidades de suas aplicações, diminuindo o número de erros e melhorando a experiência do usuário.

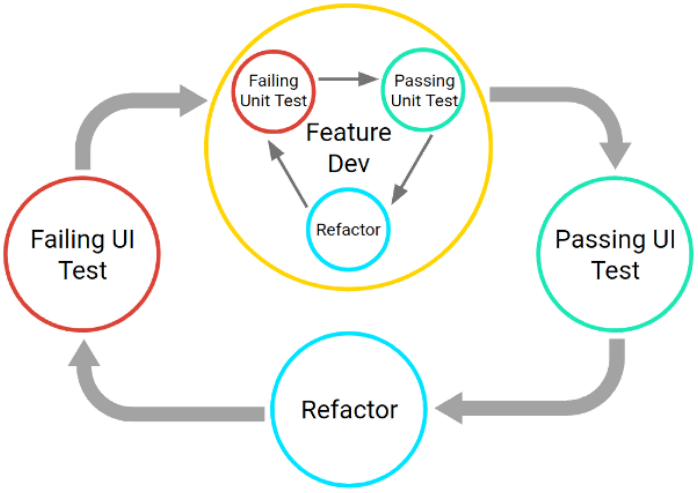


Figura 3: Ciclo de desenvolvimento no *Test-Driven-Development*
Fonte: Android Doc [ref 3]

5.4. Bibliotecas disponíveis

Como dito anteriormente, visando a utilização do *TDD* na aplicação e buscando sempre melhorar a maneira com que os testes são construídos e onde eles são executados, temos diversas bibliotecas poderosas que são capazes de nos auxiliar em diversos aspectos do desenvolvimento direcionado a testes.

5.4.1. JUnit4

O *JUnit4* [ref 5] é um *framework* de testes muito conhecido que já vem importado como *default* em qualquer aplicação *Android*. Ele provê uma série de recursos básicos para a realização de testes. As *annotations* presentes no *JUnit 4* são utilizadas em conjunto com qualquer outra biblioteca de testes dentro do *Android*.

O *JUnit* geralmente está presente em todos os tipos de testes, e pode ser usado quando queremos verificar se a execução de algum método está conforme o

esperado ou desejamos checar as variações possíveis na execução de determinada função no código.

Para importar as dependências do *JUnit 4*, a seguinte linha deve estar presente no arquivo *build.gradle* do *app*.

```
testImplementation "junit:junit:4.12"
```

5.4.1.1. @Test

Quando criamos um projeto no *Android*, automaticamente dentro do pacote de testes de unidade temos um teste exemplo utilizando o *JUnit 4*.

```
import org.junit.Test
import org.junit.Assert.*

class ExampleUnitTest {
  @Test
  fun addition_isCorrect() {
     assertEquals(4, 2 + 2)
  }
}
```

Note que todo método de um teste está anotado com a **@Test** e que os métodos de *assert()* estão todos importados de maneira estática.

Além disso, duas outras implementações são interessantes pois trazem uma série de declarações que podem ser utilizadas em conjunto com o *JUnit 4* e outras bibliotecas, quando estamos realizando testes que necessitam da simulação em um dispositivo real ou emulado (*instrumentation tests*).

```
androidTestImplementation
"com.android.support.test:runner:1.0.2"
androidTestImplementation
"com.android.support.test:rules:1.0.2"
```

Com isso, para criar um exemplo de teste instrumentado, devemos utilizar a anotação **@RunWith()**, informando qual classe desejamos que o teste execute

junto, podendo referenciar tanto a *JUnit 4* quanto outros bibliotecas, dependendo das funcionalidades necessárias na execução dos testes em questão

```
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.Assert.*

@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
  @Test
  fun useAppContext() {
     // Context of the app under test.
     val appContext =
InstrumentationRegistry.getTargetContext()
     assertEquals("expectedPackage", appContext.packageName)
  }
}
```

Note que a execução desse teste irá precisar de um dispositivo atrelado.

5.4.1.2. Filtrando os testes

Além disso, também podemos anotar os métodos de testes para definí-los entre *small, medium* ou *large*, podendo executar todos os testes categorizados quando desejarmos. Para isso, basta anotar com **@SmallTests**, **@MediumTests** ou **@LargeTests**, annotations presentes no pacote *'android.support.test.filters'*.

Esse filtro é possível realizar tanto na classe inteira do teste quanto individualmente em cada método.

```
android {
  // ...
  defaultConfig {
     // ...
     testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
     testInstrumentationRunnerArgument "size", "small"
  }
```

Para que o filtro funcione corretamente, tanto os métodos quanto a classe devem estar anotados corretamente. Para executar todos os testes com o argumento desejado, basta executar a classe de testes em questão. Esse argumento fará com que o comando passado pelo *adb* receba um argumento *size* no valor informado. Mais informações sobre linha de comando para realização de testes está disponível na referência 9.

5.4.1.3. @Rule

Um outro atributo interessante do *JUnit 4* para execução de testes instrumentais é o **@Rule**, que nos permite acessar uma *Activity* e realizar algumas checagens e validações. Note que no *Kotlin* devemos alterar a anotação para **@get:Rule**, que a tornará pública (requisito para o teste funcionar).

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
  @get:Rule
  var rule = ActivityTestRule(MainActivity::class.java)

  @Test
  fun testWithRuleActivity() {
     val activity = rule.activity
     val viewById = activity.findViewById(R.id.listview)
     // ...
  }
}
```

Com isso, podemos realizar testes para checar de uma *listView* está devidamente definida e com itens adicionados em seu *adapter*, por exemplo [ref 8].

```
import android.support.test.filters.MediumTest
import android.support.test.rule.ActivityTestRule
import android.support.test.runner.AndroidJUnit4
import android.view.View
import android.widget.ArrayAdapter
import android.widget.ListView
import org.hamcrest.Matchers.*
```

```
import org.junit.Assert.assertThat
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@MediumTest
@RunWith(AndroidJUnit4::class)
class MainActivityTest {

  @get:Rule
  var rule = ActivityTestRule(MainActivity::class.java)

  @Test
  fun ensureListViewIsPresent() {
     val activity = rule.activity
     val viewById = activity.findViewById(R.id.listview)
     assertThat<View>(viewById, notNullValue())
     assertThat<View>(viewById,
instanceOf(ListView::class.java))
     val listView = viewById as ListView
     val adapter = listView.adapter
     assertThat(adapter,
instanceOf(ArrayAdapter::class.java))
     assertThat(adapter.count, greaterThan(5))
  }
}
```

Além da atribuição do **@Rule** para uma *Activity*, também é possível utilizar para testar um *Service*, utilizando a classe *ServiceTestRule*.

5.4.1.4. @Before e @After

As anotações **@Before** e **@After** servem para que os métodos em questão sejam executados antes do início dos testes ou após o término de todos eles, dependendo da anotação, claro.

Geralmente o método mais utilizado é o **@Before** pois é onde as configurações iniciais são feitas. Por exemplo, no caso de diversos testes precisarem da mesma *Activity*, criar a instância dela dentro do método anotado com **@Before** é mais interessante do que declarar em todos os testes em questão.

No *Android Studio*, podemos utilizar o atalho *'Alt + Insert'* para iniciar a declaração dos métodos com esses comportamentos. O nome dos métodos não importa, no entanto, a convenção recomenda que o método com **@Before** chame *setUp()* e o método com **@After** chame *tearDown()*.

```
lateinit var activity: MainActivity

@Before
fun setUp() {
  activity = /* getActivity */
}
```

5.4.2. Roboletric

O *Roboletric* [ref 6] é um *framework* desenvolvido para a realização de testes de unidade que interajam com o *Android* sem a necessidade de um dispositivo real ou emulado, realizando todo o processo de *testing* no ambiente de desenvolvimento.

Para implementarmos a biblioteca devemos adicionar a seguinte linha ao arquivo *build.gradle* do *app*.

```
testImplementation "org.robolectric:robolectric:3.8"
```

Com isso, podemos solicitar a criação de uma *Activity*, realizar a busca em diversos elementos presentes e verificar se o conteúdo desses elementos está como o esperado.

```
@RunWith(RobolectricTestRunner::class)
class MainActivityTest {
  fun loadActivity_shouldShowHelloWorld() {
     val activity =
Robolectric.setupActivity(MainActivity::class.java)

     val results = activity.findViewById(R.id.textView) as
TextView
```

```
    assertThat(results.text.toString(), equalTo("Hello
World!"))
  }
}
```

Analisando o método *setupActivity()* do *roboletric* é possível analisar como ele constrói uma *Activity* para que nós possamos usá-la. Primeiro ele constrói uma instância da classe utilizando alguns métodos de *Reflection*.

```
public static <T extends Activity> T setupActivity(Class<T>
activityClass) {
 return buildActivity(activityClass).setup().get();
}
```

É possível visualizar que o método *setup()* faz a chamada de todo o ciclo de vida da *Activity* inicial que conhecemos.

```
public ActivityController<T> setup() {
 return create().start().postCreate(null).resume().visible();
}
```

A grande vantagem do *roboletric* em relação aos testes instrumentados é que ele não precisa de um dispositivo real ou emulador para executar os testes. O primeiro teste da lista pode demorar um pouco mais para ser realizado pois o *device* será preparado na *JVM*, porém, os testes subsequentes são executados em uma velocidade muito mais rápida que o de costume em comparação ao emulador.

5.4.2.1. Problemas encontrados

Apesar de muito interessante, o *Roboletric* possui alguns problemas em sua configuração e na utilização em alguns aspectos, estando presente há um bom tempo no universo *Android*, porém, só começou a receber um suporte mais efetivo no *Android Studio* em suas versões posteriores.

Um problema muito comum é a referenciação de recursos, onde ele não consegue acessar o *AndroidManifest.xml* nem os recursos básicos da aplicação presentes no diretório *resources*. Para que ele funcione como esperado, é interessante adicionar as seguintes linhas no *build.gradle* do *app*:

```
android {
  // ...
  testOptions {
     unitTests {
        returnDefaultValues = true
        includeAndroidResources = true
     }
  }
}
```

Além disso, quando executamos pelas primeiras vezes, é comum o *roboletric* fazer *download* de algumas dependências externas que não são importadas durante o *sync* do *gradle*, geralmente acessadas via acesso direto ao *link* do *sonatype*. Existem diversas reclamações na internet sobre o tema [ref 7] e algumas soluções para agilizar o processo, visto que a *url* onde os arquivos necessários ficam disponíveis não possui um *link* muito rápido.

No meu caso, apenas deixei um dos testes executando e fazendo o *download* dos arquivos necessários e depois não precisei mais me preocupar com isso. Até onde percebi, o *Android Studio* vai armazenando parte do *download* em um *cache*, então encerrar e executar os testes não pareceu atrapalhar no progresso de obtenção do arquivo, que a princípio não possui nenhum indicador de progresso.

5.4.3. Espresso

O *Espresso* é a biblioteca para testes instrumentados muito utilizada e com várias funcionalidades que facilitam e melhoram a escrita de testes, além de permitir a visualização do que está sendo executado, permitindo ao desenvolvedor acompanhar a execução dos testes na interface da aplicação.

Para adicionar o *Espresso* ao projeto, basta incluir a sua implementação no arquivo *build.gradle* do *app*.

```
androidTestImplementation
"com.android.support.test.espresso:espresso-core:3.0.2"
```

Com isso é possível utilizar a declaração simples do *Espresso* para criar testes mais interessantes e mais legíveis.

```
@Test
fun sendButton_shouldWriteHelloName() {
  onView(withHint("Type your name")).perform(typeText("Paulo
Salvatore"))
  onView(withText("Send")).perform(click())
  onView(withText("Hello, Paulo
Salvatore!")).check(matches(isDisplayed()))
}
```

Uma sessão de *training* completa para o *Espresso* está disponível na própria documentação do *Android*, e pode ser acessada através da referência 10.

Um detalhe bem interessante do *Espresso* é que ele elimina a necessidade de executarmos pausas nos testes aguardando que o comando seja processado, pois ele entende automaticamente que deve esperar a informação sincronizar para prosseguir com a execução, uma funcionalidade que além de eliminar muito *boilerplate* torna os testes ainda mais legíveis.

5.4.3.1. Espresso Recorder

Uma ferramenta presente no *Android Studio* que possui uma funcionalidade bem interessante é o *Espresso Recorder*, que consiste em um gravador de ações na interface. Ao iniciar o *recorder*, a aplicação será executada no dispositivo selecionado. Cada ação realizada na tela será gravada e exibida em uma linha do tempo inteligente. Ao final da gravação, um código contendo exatamente a sequência de ações realizada será criado, nos permitindo executar exatamente o mesmo teste realizado, eliminando a necessidade de escrevermos as instruções manualmente.

Apesar de muito interessante, o código gerado é um tanto quanto complicado de ler e de realizar manutenção, portanto, essa ferramenta não substitui a necessidade de criar nossos próprios testes manualmente.

Para iniciar a gravação de um teste, basta acessar o menu "Run | Record Espresso Test", selecionar um dispositivo, esperar a aplicação rodar e realizar as ações. A qualquer momento é possível adicionar *Assertions* que irão garantir que um elemento possui o valor desejado. O processo de gravação de um teste pode ser observado na figura 4.

Uma funcionalidade interessante na ferramenta é a parte de *Assertions*, que tira uma *screenshot* em tempo real da aplicação, nos permitindo selecionar qualquer elemento da tela de forma inteligente, algo similar ao que veremos com o *UI Automator*.

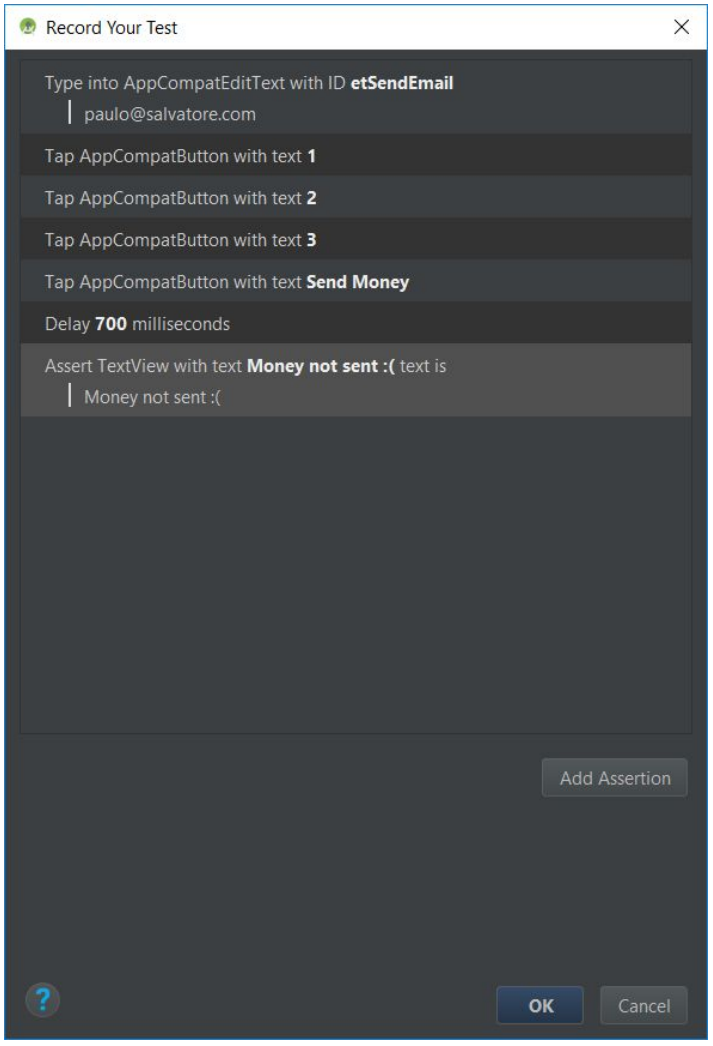


Figura 4: Exemplo de um teste gravado com o *Espresso Recorder*
Fonte: autor - *Android Studio*

Com o teste gravado, é necessário criar uma classe nova para que o código seja gerado. Essa nova classe será criada em *Java*, o que é possível converter para *Kotlin* utilizando o atalho *'Ctrl + Shift + Alt + K'*.

5.4.4. UI Automator Viewer

O *UI Automator Viewer* é uma ferramenta semelhante ao *'Inspecionar Elemento'* presente nos navegadores da *web*, permitindo uma análise da interface de usuário construída na aplicação. Ela consiste em tirar uma *screenshot* da tela do dispositivo, realizar uma análise da interface, e exibir detalhadamente cada *view* construída com suas respectivas informações, permitindo entender como a tela está estruturada e os valores de cada elemento.

O arquivo geralmente está localizado na pasta do *Sdk* do *Android* e permite uma execução da aplicação de forma separada do *Android Studio*. No meu caso (ambiente *windows*) a localização é no *AppDataLocal* (pode ser acessado por *%LocalAppData%*) no caminho *'...\Android\Sdk\tools\bin\uiautomatorviewer.bat'*. Basta executar o arquivo em questão que o programa irá abrir.

Com o programa aberto e um dispositivo atrelado, basta clicar no segundo ícone *'Device Screenshot (uiautomator dump)'* para tirar uma *screenshot* da tela do dispositivo (independente de qual seja) para explorar seus elementos, conforme é possível visualizar na figura 5.

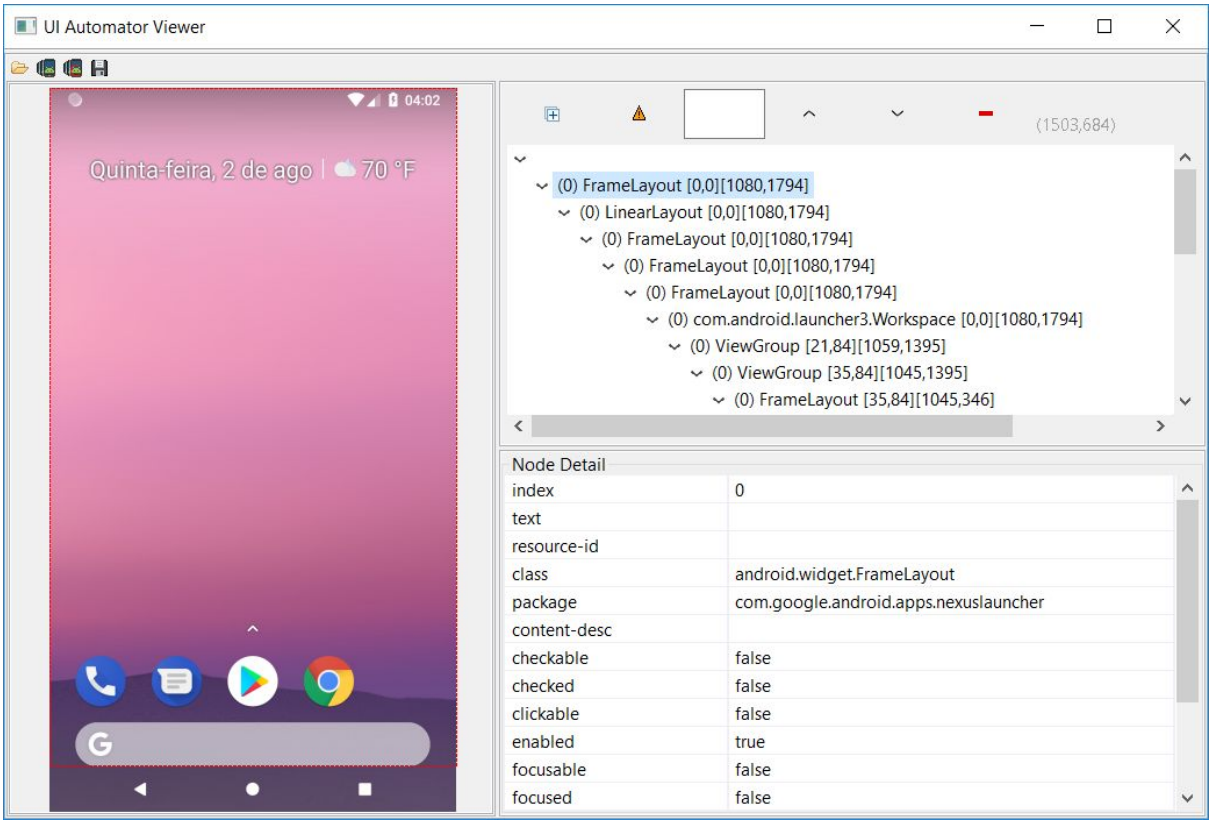


Figura 5: Inspeção da tela inicial de um dispositivo *Android*.
Fonte: autor - *UI Automator Viewer*

Uma ferramenta semelhante é possível encontrar no *Android Studio*, o *'Layout Inspector'* [ref 11], que possui as mesmas características, porém, necessita do um processo da aplicação atrelado, nem sempre exibindo os aplicativos abertos no dispositivo. Para acessá-lo basta ir no menu "Tools | Layout Inspector", selecionar um processo (como é possível ver na figura 6) e realizar a inspeção do *layout* dentro do próprio *Android Studio* em uma nova aba que irá abrir.

Cada *screenshot* realizada será gravada em uma pasta chamada *captures*, disponível na raiz do projeto.

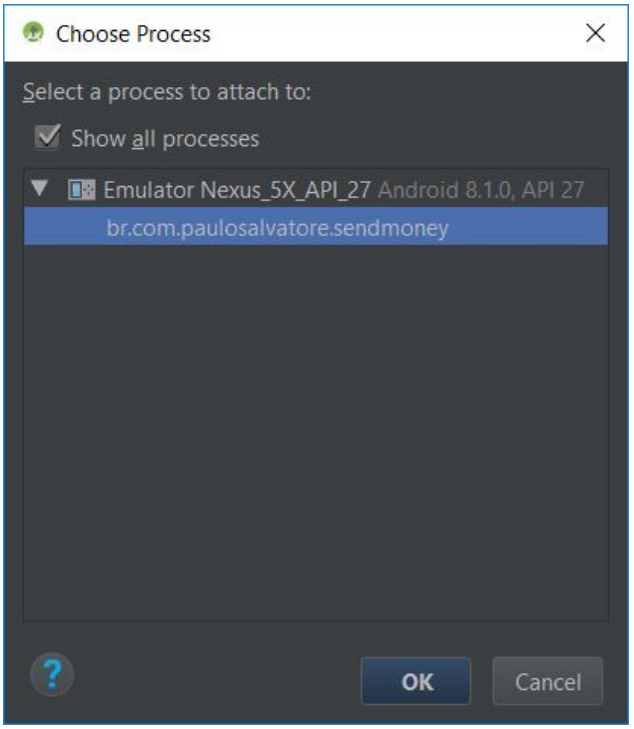


Figura 6: *Layout Inspector* precisa de um processo atrelado
Fonte: autor - *Android Studio*

5.4.5. UI Automator

Apesar do nome parecido, o *UI Automator* é diferente do *UI Automator Viewer*, afinal, ele consiste em uma biblioteca que permite a realização de testes instrumentais em uma aplicação, fazendo solicitações diretas no dispositivo como *'apertar botão de home'* e realizar buscas diretas no aparelho.

A grande vantagem da utilização dessa ferramenta é para a realização dos testes que envolvem funcionalidades conjuntas em diferentes aplicações, onde é necessário realizar testes de comunicação entre uma aplicação e outra, por exemplo [ref 12].

Para saber como navegar entre as *views default* do dispositivo, é interessante que a ferramenta seja utilizada em conjunto com o *UI Automator Viewer*, que permite visualizar os textos de cada elemento, por exemplo, facilitando na hora de buscar os elementos e realizar os cliques.

A implementação da biblioteca pode ser realizada realizando a seguinte declaração no arquivo *build.gradle* do *app*.

```
androidTestImplementation
"com.android.support.test.uiautomator:uiautomator-v18:2.1.3"
```

É recomendado desativar as animações no dispositivo utilizado para os testes. Além disso, a biblioteca só funcionará nas *APIs* do *Android* a partir da versão 18, portanto, não é possível executar em *APIs* anteriores a essa. Caso queira manter a *SDK* mínima do projeto anterior à 18, será necessário forçar utilização da biblioteca, criando um novo arquivo *AndroidManifest.xml* no diretório *androidTest* ou *test*, com o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:tools="http://schemas.android.com/tools"
  package="br.com.paulosalvatore.sendmoney">

  <uses-sdk
tools:overrideLibrary="android.support.test.uiautomator.v18"/
>
</manifest>
```

Com isso, podemos criar nosso primeiro teste que abre o aplicativo a partir da tela de *home* e verifica se tudo ocorreu corretamente [ref 12].

```
import android.content.Intent
import android.support.test.InstrumentationRegistry
import android.support.test.filters.SdkSuppress
import android.support.test.runner.AndroidJUnit4
import android.support.test.uiautomator.By
import android.support.test.uiautomator.UiDevice
import android.support.test.uiautomator.Until
import org.hamcrest.Matchers.notNullValue
import org.junit.Assert.assertThat
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
```

```
@SdkSuppress(minSdkVersion = 18)
class MainActivityTest {
  companion object {
     private const val BASIC_SAMPLE_PACKAGE = "yourPackage"
     private const val LAUNCH_TIMEOUT = 5000
  }

  private lateinit var mDevice: UiDevice

  @Test
  fun startMainActivityFromHomeScreen() {
     // Initialize UiDevice instance
     mDevice =
UiDevice.getInstance(InstrumentationRegistry.getInstrumentati
on())

     // Start from the home screen
     mDevice.pressHome()

     // Wait for launcher
     val launcherPackage = mDevice.launcherPackageName
     assertThat(launcherPackage, notNullValue())

mDevice.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)
),
           LAUNCH_TIMEOUT.toLong())

     // Launch the app
     val context = InstrumentationRegistry.getContext()
     val intent = context.packageManager
           .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE)
     // Clear out any previous instances
     intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK)
     context.startActivity(intent)

     // Wait for the app to appear
```

```
mDevice.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).dep
th(0)),
            LAUNCH_TIMEOUT.toLong())
    }
}
```

Um artigo interessante sobre utilização do *UI Automator*, publicado no *blog Vogella* pode ser encontrado na sessão de leitura adicional.

5.4.6. Mockito

O *Mockito* [ref 13] é uma biblioteca muito utilizada pois facilita o desenvolvimento de testes fornecendo *mocks* para determinados objetos. Um *Mock* consiste em uma implementação forçada de uma interface/classe que funcionará da maneira que precisamos, sem que uma implementação real seja necessária.

O conceito é bem semelhante ao de injeção de dependência, com a diferença que um objeto *mockado* não é um objeto real e não deve ser usado como tal, onde basicamente uma implementação falsa de seus métodos servirá como propósito para testar uma determinada funcionalidade sem precisar validar a funcionalidade de outras dependências envolvidas. Esse conceito não é exclusividade da biblioteca *Mockito*, porém, ela é utilizada na maioria dos projetos *Java* que necessitam dessa funcionalidade.

Para implementar o *Mockito* em seu projeto basta adicionar a seguinte declaração no arquivo *build.gradle* do *app*:

```
testImplementation "org.mockito:mockito-core:2.7.21"
```

Para entender um pouco mais do conceito *mock*, podemos visualizar o seguinte exemplo abaixo.

```
// É possível mockar classes concretas, não
// apenas interfaces
val mockedList = mock(LinkedList::class.java)

// Podemos declarar o que queremos que aconteça
```

```
// quando determinado método é chamado
`when`(mockedList.get(0)).thenReturn("first")

// Irá exibir "first", como foi declarado
// anteriormente
println(mockedList.get(0))

// Irá exibir null, pois não foi declarado
// anteriormente
println(mockedList.get(999))
```

Além do método *mock()*, também podemos utilizar o método *spy()*, que basicamente criará um *mock()* a partir de um objeto real, onde alguns métodos desse objeto ainda podem ser alterados para apresentar funcionalidades desejadas, porém, as partes declaradas do objeto real irão funcionar como tal. É uma espécie de *mix* entre um objeto *mockado* e um objeto real.

Sugiro que leia a leitura adicional do *Mockito*, contendo explicações mais detalhadas tanto para diversos termos do *TDD* quanto para a utilização da biblioteca em si.

Quando for utilizar o *mock*, lembre-se de quatro coisas básicas [ref 14]:

- Não 'mock' tipos que você não possui;
- Não 'mock' objetos com apenas valores;
- Não 'mock' qualquer coisa;
- Tenha amor com seus testes!

5.4.6.1. Mockito Kotlin

Apesar do *Kotlin* ser 100% interoperável com o *Java*, é possível deixar a declaração do *Mockito* um pouco mais próxima do que estamos acostumados a realizar no *Kotlin*. Para isso, podemos utilizar a biblioteca *Mockito Kotlin* [ref 15], que melhora a declaração, como é possível visualizar no exemplo abaixo, extraído da documentação da biblioteca.

Para implementá-la, basta adicionar o seguinte trecho ao arquivo *build.gradle* do *app*.

```
testImplementation
"com.nhaarman.mockitokotlin2:mockito-kotlin:2.0.0-RC1"
```

```
@Test
fun doAction_doesSomething(){
  /* Given */
  val mock = mock<MyClass> {
     on { getText() } doReturn "text"
  }
  val classUnderTest = ClassUnderTest(mock)

  /* When */
  classUnderTest.doAction()

  /* Then */
  verify(mock).doSomething(any())
}
```

5.5. Robot Pattern

O *Robot Pattern* é um padrão de testes instrumentais proposto por *Jake Wharton* e consiste em uma forma de construir os testes visando melhorando a legibilidade e reaproveitamento de declarações, visando a separação de "o que" é feito no teste e de "como" o teste é feito [ref 16].

O padrão propõe a construção de uma classe *Robot* que funciona como uma abstração para realização dos testes, contendo toda a lógica para execução de ações individuais nos testes como cliques e ações e propondo que cada ação específica deve possuir um novo *Robot* que contém declarações específicas para realizar o que se é esperado.

5.5.1. Criando um Robot

Para iniciar nossa simulação do *Robot Pattern*, crie um novo projeto, *API 15*, com uma *EmptyActivity* chamada *LoginActivity*.

O início consiste na criação do *Robot* que servirá como base. Crie um pacote dentro de '*androidTest*' chamado '*robot*' com uma nova classe *BaseTestRobot* dentro.

```
import android.support.test.espresso.Espresso.onData
import android.support.test.espresso.Espresso.onView
```

```
import android.support.test.espresso.ViewInteraction
import android.support.test.espresso.action.ViewActions
import android.support.test.espresso.assertion.ViewAssertions
import android.support.test.espresso.matcher.ViewMatchers
import
android.support.test.espresso.matcher.ViewMatchers.withId
import org.hamcrest.CoreMatchers.allOf
import org.hamcrest.CoreMatchers.anything

open class BaseTestRobot {
  fun fillEditText(resId: Int, text: String): ViewInteraction
=

onView(withId(resId)).perform(ViewActions.replaceText(text),
ViewActions.closeSoftKeyboard())

  fun clickButton(resId: Int): ViewInteraction =
onView((withId(resId))).perform(ViewActions.click())

  fun textView(resId: Int): ViewInteraction =
onView(withId(resId))

  fun matchText(viewInteraction: ViewInteraction, text:
String): ViewInteraction = viewInteraction

.check(ViewAssertions.matches(ViewMatchers.withText(text)))

  fun matchText(resId: Int, text: String): ViewInteraction =
matchText(textView(resId), text)

  // Para usos futuros, caso necessário
  fun clickListItem(listRes: Int, position: Int) {
     onData(anything())
           .inAdapterView(allOf(withId(listRes)))
           .atPosition(position).perform(ViewActions.click())
  }
}
```

Depois, criamos nosso primeiro *Robot* que será aplicado de fato para testar uma tela simples de *login*.

```
import android.content.Context
import android.support.test.espresso.ViewInteraction
import br.com.paulosalvatore.loginapplication.R

class LoginRobot(private val context: Context) :
BaseTestRobot() {

  fun setEmail(email: String) = apply {
fillEditText(R.id.etEmail, email) }

  fun setPassword(pass: String) = apply {
fillEditText(R.id.etPassword, pass) }

  fun clickLogin() = apply {
     clickButton(R.id.btLogin)
  }

  fun matchErrorText(err: Int): ViewInteraction {
     return matchText(textView(android.R.id.message),
context.getString(err))
  }
}
```

Com isso, criamos nossa primeira classe de testes, que utilizará o *robot* de *login* criado.

```
import android.support.test.rule.ActivityTestRule
import android.support.test.runner.AndroidJUnit4
import
br.com.paulosalvatore.loginapplication.robot.LoginRobot
import org.junit.Before
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith
```

```
@RunWith(AndroidJUnit4::class)
class LoginActivityTest {
  @get:Rule
  var rule = ActivityTestRule(LoginActivity::class.java)

  private lateinit var robot: LoginRobot

  @Before
  fun setUp() {
     robot = LoginRobot(rule.activity)
  }

  // First Test
  @Test
  fun loginWrongPassword() {
     robot
           .setEmail("admin@admin.com")
           .setPassword("wrong")
           .clickLogin()
           .matchErrorText(R.string.login_fail)
  }

  // After Texts
  @Test
  fun loginMissingEmailPassword() {
     robot
           .clickLogin()
           .matchErrorText(R.string.missing_fields)
  }

  @Test
  fun loginMissingPassword() {
     robot
           .setEmail("admin@admin.com")
           .clickLogin()
           .matchErrorText(R.string.missing_fields)
  }
```

```
  @Test
  fun loginSuccess() {
     robot
           .setEmail("admin@admin.com")
           .setPassword("admin")
           .clickLogin()
           .matchText(R.id.tvNameSurname,
rule.activity.getString(R.string.name_surname))
  }
}
```

Antes de construir a aplicação, precisamos importar as bibliotecas necessárias para que os testes e nossa *Activity* funcionem. Adicione as seguintes linhas no arquivo *build.gradle* do *app*.

```
dependencies {
   // ...
   // Test Rules
   androidTestImplementation
"com.android.support.test:rules:1.0.2"

   // Anko
   implementation "org.jetbrains.anko:anko-commons:0.10.5"
   implementation "org.jetbrains.anko:anko-design:0.10.5"
}
```

Depois, adicione os seguintes conteúdos no *layout* e na classe da *Activity*.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
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"
  tools:context=".LoginActivity">
```

```
<EditText
    android:id="@+id/etEmail"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:hint="Type your E-mail"
    android:inputType="textEmailAddress"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:ignore="HardcodedText" />

<EditText
    android:id="@+id/etPassword"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:hint="Type your password"
    android:inputType="textPassword"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/etEmail"
    tools:ignore="HardcodedText" />

<Button
    android:id="@+id/btLogin"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
```

```
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="Login"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/etPassword"
    tools:ignore="HardcodedText" />

  <TextView
    android:id="@+id/tvNameSurname"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:gravity="center"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btLogin" />

</android.support.constraint.ConstraintLayout>
```

```
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_login.*
import org.jetbrains.anko.alert
import org.jetbrains.anko.longToast
import org.jetbrains.anko.yesButton

class LoginActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```

```
        setContentView(R.layout.activity_login)

        btLogin.setOnClickListener {
            val login = etEmail.text.toString()
            val password = etPassword.text.toString()

            if (login.isEmpty() ||
                    password.isEmpty()) {
                alert(R.string.missing_fields) {
                    yesButton {  }
                }.show()
            } else if (login.equals("admin@admin.com") &&
                    password.equals("admin")) {
                tvNameSurname.text =
getString(R.string.name_surname)

                longToast("Yay!!")
            } else {
                alert(R.string.login_fail) {
                    yesButton {  }
                }.show()
            }
        }
    }
}
```

Com isso, podemos notar que os testes ficam em um padrão muito mais interessante e muito mais legíveis, mesmo que o *Espresso* (biblioteca utilizada no caso) já possua uma abstração interessante, podemos fazer ainda melhor, promovendo a reutilização de código e deixando extremamente fácil e despreocupante a escrita de novos testes.

Acredito que esse seja um dos melhores padrões de testes propostos atualmente, tornando muito mais prazeroso e natural o desenvolvimento de testes das aplicações, o que nem sempre foi uma maneira fácil de estruturar. A única questão é que por enquanto ele é voltado para o desenvolvimento de testes instrumentais, que como visto anteriormente, não são a maioria presente na aplicação, porém, são tão importantes quanto os testes unitários.

O fluxo presente nesse exemplo do *robot pattern* foi inspirado na referência 17, em um *post* feito no *Medium* e produzido pelo desenvolvedor *Faruk Toptaş*.

6. ProGuard

O *ProGuard* é uma ferramenta desenvolvida pela *GuardSquare* que presente no universo de qualquer desenvolvedor que irá publicar o arquivo compilado de seu *software*. Ele constitui em uma ferramenta de análise e otimização do código, reduzindo o tamanho do arquivo compilado e dificultando o trabalho de engenharia reversa, utilizado para que outros desenvolvedores consigam acesso ao seu código fonte através do arquivo de distribuição. Existem casos em que o *ProGuard* reduz até 90% do tamanho dos arquivos finais de uma aplicação.

6.1. Motivos

Apesar de parecer muito óbvia a utilização do *ProGuard*, alguns aspectos complicados de sua implementação em uma aplicação real fazem com que muitos desenvolvedores não considerem seu uso. Na *Google I/O 18'*, esse assunto recebeu uma atenção especial que pode ser conferida no vídeo 3 (ir para vídeo), onde diversos dados interessantes foram mostrados.

Uma justificativa muito comum que os desenvolvedores alegam em relação ao tamanho da aplicação é que o avanço tecnológico irá suprir a necessidade das aplicações ficarem menores, onde os dispositivos vão possuir cada vez mais recursos fazendo com que a gente não tenha que se preocupar com a otimização da utilização desses recursos. Analisando desde o início do *Android* até meados de 2017, é possível ver que por mais que o avanço tecnológico exista, o tamanho médio das aplicações está crescendo cada vez mais, como é possível visualizar na figura 7.

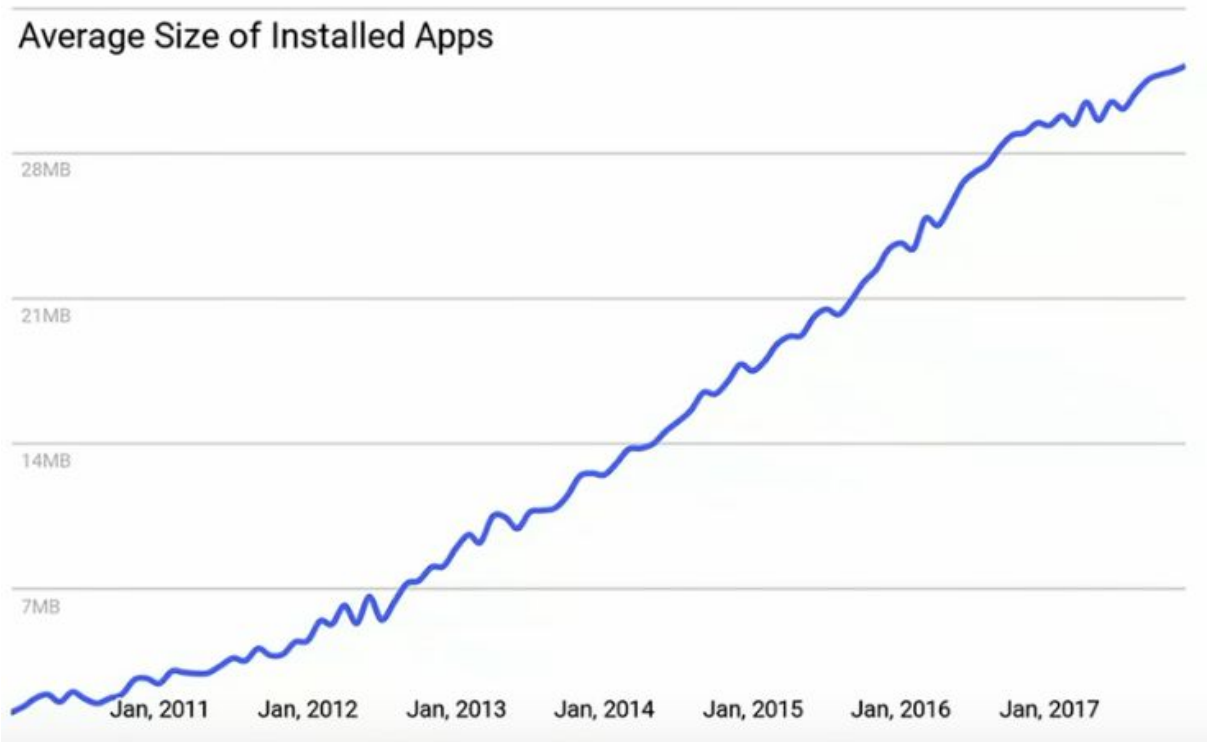


Figura 7: Tamanho médio dos aplicativos instalados
Fonte: *Google Internal Data, 2018*

É justamente nesse ponto que deve entrar nossa preocupação em reduzir o tamanho das aplicações. Isso não impacta apenas no tamanho do arquivo, mas sim na experiência do usuário, afinal, o aplicativo abrirá mais rápido pois menos código para ser lido; ele será baixado mais rápido, pois o tamanho é menor; a resposta das funcionalidades será mais efetiva; e ele funcionará em praticamente qualquer dispositivo, independente da circunstância.

Apesar das vantagens parecerem bem óbvias, na realidade isso acaba não acontecendo. Como é possível visualizar na figura 8, *Jeb Ware* fez um estudo pegando todos os aplicativos instalados no seu celular e checando quais estavam ofuscados ou não. O resultado foi interessante, a grande maioria das aplicações não continham nada ou quase nada de código ofuscado, o que nos dá um indicador que ainda há muito espaço para ser otimizado entre as aplicações a nível mundial.

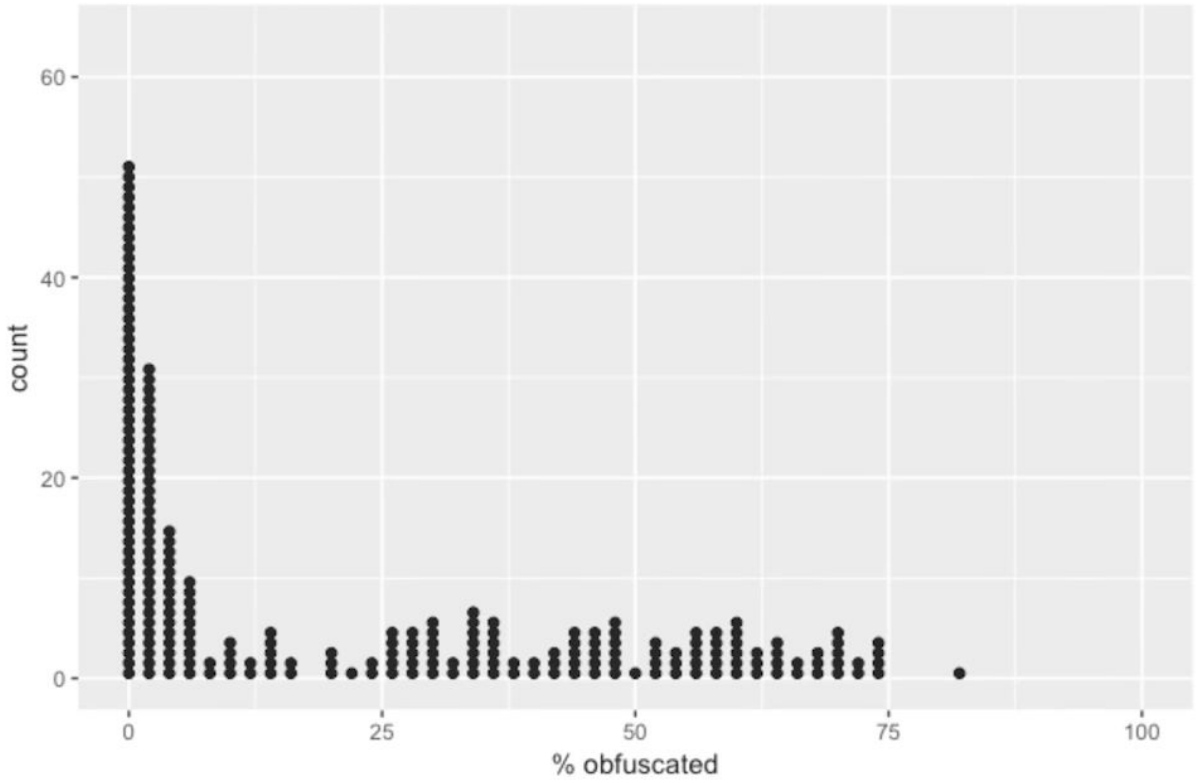


Figura 8: Quantidade de apps ofuscados e porcentagem de ofuscação
Fonte: *Jeb Ware* sobre *How ProGuard Works*, vídeo 4 (ir para vídeo)

6.2. Funcionamento

O funcionamento do *ProGuard* consiste em receber o arquivo compilado; aplicar uma remoção de código morto (*shrink*), ou seja, que não está sendo usado em nenhum lugar; otimizar esse código alterando o tipo de acesso de classes e métodos, removendo parâmetros não utilizados e o conteúdo de alguns métodos pode ser transformado em uma execução *inline*; ofuscar o código otimizado, alterando o nome de todos os membros e classes possíveis, visando dificultar a engenharia reversa e o entendimento do código-fonte, além de otimizá-lo para execução; e por último a pré-verificação, que checa se os arquivos estão estruturados de acordo com as especificações da *J2Me* - Esse passo vem por padrão desativado e não ocorre no *Android*. Os passos descritos aqui podem ser visualizados na figura 9.

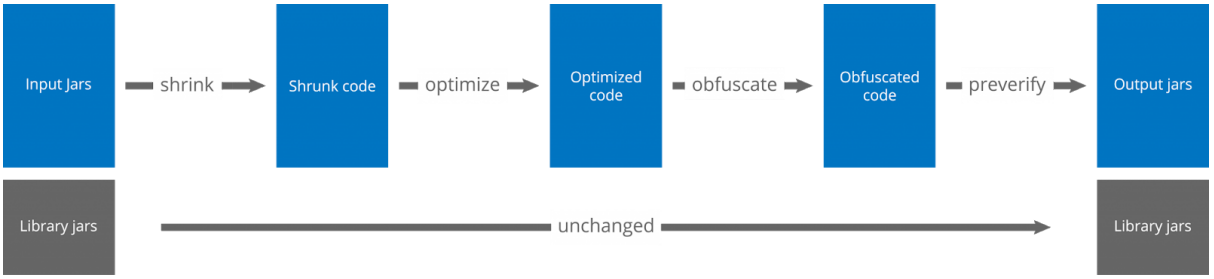


Figura 9: Passos de execução do *Proguard*
Fonte: ProGuard manual [ref 18]

6.3. Configuração do projeto

Para demonstrar a atuação do *ProGuard*, criaremos um novo projeto, *API 15* com uma *EmptyActivity* chamada *MainActivity*. Para realizar nosso primeiro teste com o *ProGuard*, abra o arquivo da *MainActivity* e adicione o seguinte código:

```
class MainActivity : AppCompatActivity() {
  companion object {
     private const val NUM_1 = 1
     private const val NUM_2 = 2
  }

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)

     somar1(1, 2)

     val soma2 = somar2(1, 2)
     println(soma2)

     val soma3 = somar3()
     println(soma3)
  }

  fun somar1(num1: Int, num2: Int) {
     println(num1 + num2)
  }
```

```
  fun somar2(num1: Int, num2: Int): Int = num1 + num2

  fun somar3() = NUM_1 + NUM_2
}
```

O ProGuard vem automaticamente incluído em todo projeto criado pelo *Android Studio*, ele vem por *default* desativado e a sua declaração pode ser encontrada no arquivo *build.gradle* do *app*.

```
buildTypes {
   release {
       minifyEnabled false
       proguardFiles
getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
   }
}
```

Antes de ativar, faremos um *build* do *apk* para ter comparação entre o antes e depois do *ProGuard*. Para isso, vá em "Build | Build APK(s)". Assim que o *build* for finalizado, clique em *'locate'* e abra a pasta onde o *apk* foi salvo. No meu caso, a aplicação gerada ficou com 2.017 KB. Mova o arquivo para uma outra pasta e altere o nome para identificar que essa versão está sem o *ProGuard* ativado.

6.4. ClassyShark

O *ClassyShark* é uma ferramenta que nos permite e visualização de todo o conteúdo presente em um arquivo *APK*. O projeto é mantido pelo *Google* e é disponibilizado de maneira *open source* pelo *GitHub* [ref 21].

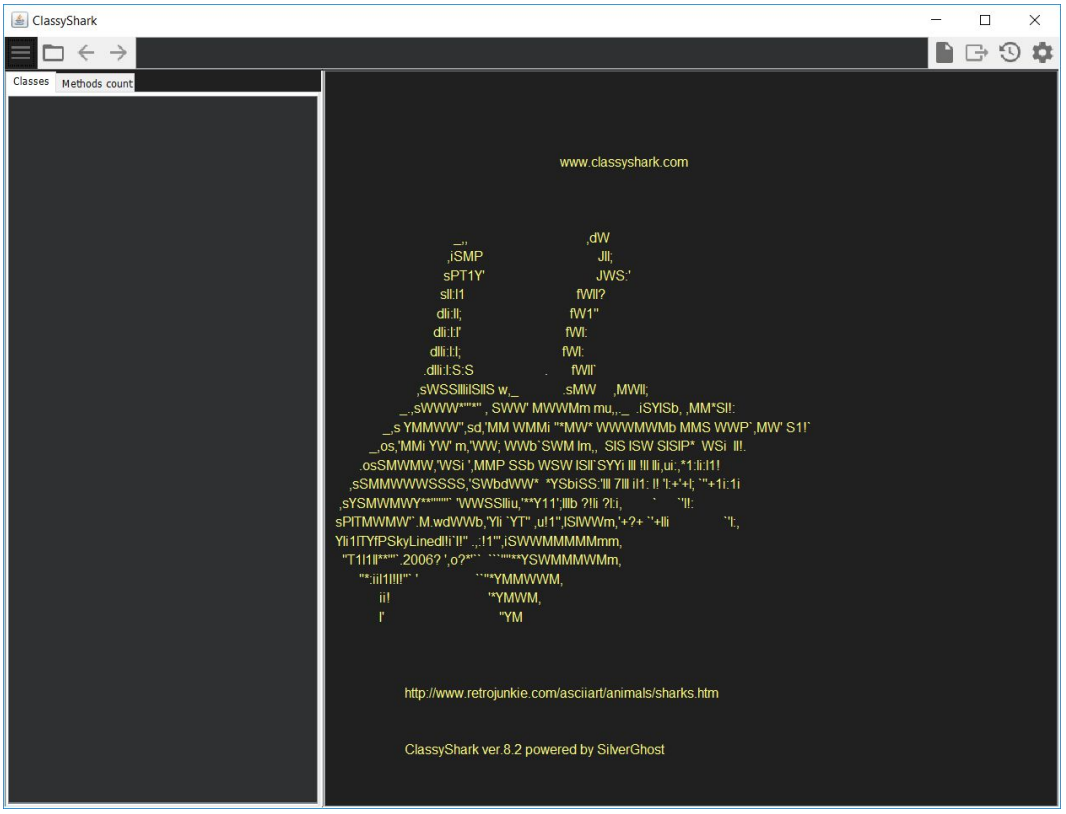
Para utilizá-lo, faça o *download* da última versão disponível na aba *'Releases'* do projeto no *GitHub*, que pode ser acessada através do link:

https://github.com/google/android-classyshark/releases

Execute o arquivo *.jar*, talvez seja necessário fazer via linha de comando:

```
java -jar ClassyShark.jar
```

Desenvolvido por [Paulo Salvatore]



Para iniciar, clique no botão de abrir e selecione o *apk* gerado anteriormente. A hierarquia da aplicação será exibida ao lado esquerdo. Rolando a barra para o final, deverá haver uma pasta chamada *'classes'* e dentro uma outra pasta chamada *'classes.dex'*, com todos os arquivos da aplicação. Abra e procure pelo pacote da sua aplicação. Dentro do pacote, pegaremos como exemplo a classe *MainActivity*, dê um clique no arquivo e será possível observar seu conteúdo ao lado direito.

Apesar do conteúdo dos métodos estarem ocultos, é possível notar que o nome da cada classe e método está exatamente do mesmo jeito que escrevemos, expondo completamente nosso código fonte.

Como o *ClassyShark* exibe apenas as assinaturas dos métodos, devemos usar algum *APK Decompiler* se quisermos exibir o conteúdo inteiro. Atualmente existem diversas opções disponíveis, tanto online, como o [JavaDecompilers.com/APK] ou algumas opções *offline*, como o *APK Easy Tool*, que faz algumas coisas adicionais.

Por questões de facilidade e irei demonstrar o processo de *decompile* utilizando o método online referenciado acima. Acesse o site informado e faça o *upload* e *decompile* do mesmo *APK* aberto no *ClassyShark*. Na tela de *result*, navegue até o pacote onde está a classe *MainActivity* e faça o download do

arquivo. Dentro do arquivo baixado, podemos verificar todo o conteúdo da *MainActivity*, que está bem próximo ao que escrevemos.

```
protected void onCreate(@Nullable Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView((int) C0185R.layout.activity_main);
  somar1(1, 2);
  System.out.println(somar2(1, 2));
  System.out.println(somar3());
}

public final void somar1(int num1, int num2) {
  System.out.println(num1 + num2);
}

public final int somar2(int num1, int num2) {
  return num1 + num2;
}

public final int somar3() {
  return 3;
}
```

Agora vamos gerar uma versão da aplicação com o *ProGuard*, para ativá-lo, basta alterar a opção *minifyEnabled* para *true* e o *Android Studio* irá utilizar as regras do *ProGuard* desativar nos arquivos referenciados em *proguardFiles* para realização do processo, que por padrão já vem com algumas regras do *ProGuard* escritas.

Além disso, para que o processo esteja visível na *build* de *debug*, é necessário adicionar uma configuração extra no *build types*.

```
buildTypes {
   debug {
       minifyEnabled true
```

```
        proguardFiles
getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
    }
    // ...
}
```

Antes de gerar o novo *APK*, vá no menu "Build | Clean Project" para certificar que nenhum *cache* irá atrapalhar o *APK* gerado. Gere uma nova *build* e mova o arquivo gerado para a mesma pasta onde colocou o anterior, alterando o nome para identificá-lo como tendo o *ProGuard* ativado. Inicialmente a diferença é nítida, o *APK* gerado possui 1.050 KB, praticamente metade do tamanho do anterior.

Decompilando a aplicação podemos observar que o código da *MainActivity* recebeu várias alterações, principalmente no que se refere aos nomes de variáveis e métodos.

```
public final void m3032a(int i, int i2) {
  System.out.println(i + i2);
}

public final int m3033b(int i, int i2) {
  return i + i2;
}

public final int m3034k() {
  return 3;
}

protected void onCreate(Bundle bundle) {
  super.onCreate(bundle);
  setContentView((int) R.layout.activity_main);
  m3032a(1, 2);
  System.out.println(m3033b(1, 2));
  System.out.println(m3034k());
}
```

Um outro detalhe importante, para que o processo de otimização aconteça, é necessário referenciar um outro arquivo, chamado *'proguard-android-optimize.txt'*, que possui algumas linhas adicionais de configuração, removendo a linha *'-dontoptimize'*. Uma recomendação do *Google* é que realizemos o processo de *build* inicialmente sem otimização, assim que certificar que tudo está funcionando como esperamos, aplicamos o passo de otimização.

Para realizar a configuração, basta alterar o arquivo a ser utilizado pelo *ProGuard*.

```
buildTypes {
   debug {
       minifyEnabled true
       proguardFiles
getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
   }
   // ...
}
```

As alterações que o arquivo faz podem ser visualizadas no trecho a seguir:

```
-optimizations
!code/simplification/arithmetic,!code/simplification/cast,!fi
eld/*,!class/merging/*
-optimizationpasses 5
-allowaccessmodification
```

Realizando essa modificação e gerando um novo *apk* realizando o mesmo processo feito anteriormente é possível ver que o tamanho da aplicação caiu um pouco mais, indo para 1.029 KB.

Abrindo o arquivo da *MainActivity* podemos visualizar que tivemos uma redução ainda mais completa do código, eliminando os métodos e colocando o conteúdo deles direto *inline*, removendo a chamada das operações que eram executadas.

```
protected final void onCreate(Bundle bundle) {
  super.onCreate(bundle);
```

```
  setContentView((int) R.layout.activity_main);
  System.out.println(3);
  System.out.println(3);
  System.out.println(3);
}
```

6.5. Keep Rules

Como podemos perceber o *ProGuard* irá modificar diversas partes do código, baseado em sua análise do que é utilizado ou não e de como as coisas são feitas. No entanto, diversos problemas podem ocorrer durante esse processo, principalmente quando estamos utilizando alguma biblioteca em que não temos controle do está sendo realizado; quando algum método utiliza *Reflection* que acessa o nome de classes e métodos em tempo de compilação e que o *ProGuard* não tem como prever; quando o *app* faz referência a uma classe apenas no *AndroidManifest*; quando o *app* chama um método da *JNI (Java Native Interface)*; e alguns outros motivos.

Para contornar esses problemas existem as *Keep Rules*, que consistem em um conjunto de regras de declaração que nos permitem informar ao *ProGuard* que determinado código deve ser mantido exatamente como está.

6.5.1. @Keep

A primeira maneira é adicionar uma *annotation* **@Keep** no método ou na classe que queremos manter.

Adicionando essa anotação na classe *MainActivity*, podemos gerar uma nova *build* e verificar o resultado:

```
@Keep
public final class MainActivity extends C0597d {
  public static final C0400a Companion = new C0400a();
  private static final int NUM_1 = 1;
  private static final int NUM_2 = 2;
  private HashMap _$_findViewCache;

  public static final class C0400a {
     private C0400a() {
```

```
    }
  }

  public final void _$_clearFindViewByIdCache() {
     if (this._$_findViewCache != null) {
        this._$_findViewCache.clear();
     }
  }

  public final View _$_findCachedViewById(int i) {
     if (this._$_findViewCache == null) {
        this._$_findViewCache = new HashMap();
     }
     View view = (View)
this._$_findViewCache.get(Integer.valueOf(i));
     if (view != null) {
        return view;
     }
     view = findViewById(i);
     this._$_findViewCache.put(Integer.valueOf(i), view);
     return view;
  }

  protected final void onCreate(Bundle bundle) {
     super.onCreate(bundle);
     setContentView((int) R.layout.activity_main);
     somar1(1, 2);
     System.out.println(somar2(1, 2));
     System.out.println(somar3());
  }

  public final void somar1(int i, int i2) {
     System.out.println(i + i2);
  }

  public final int somar2(int i, int i2) {
     return i + i2;
  }
```

```
  public final int somar3() {
     return 3;
  }
}
```

Note que apesar de algumas otimizações na classe, grande parte da estrutura foi mantida. Quando anotamos métodos ou campos, também podemos ter um resultado interessante, onde o *ProGuard* irá trabalhar em todos os aspectos da classe com exceção dos que estão anotados.

```
public final class MainActivity extends C0564c {
  public static final C0400a f2552m = new C0400a();

  public static final class C0400a {
     private C0400a() {
     }
  }

  protected final void onCreate(Bundle bundle) {
     super.onCreate(bundle);
     setContentView((int) R.layout.activity_main);
     System.out.println(3);
     System.out.println(somar2(1, 2));
     System.out.println(somar3());
  }

  @Keep
  public final int somar2(int i, int i2) {
     return i + i2;
  }

  @Keep
  public final int somar3() {
     return 3;
  }
}
```

6.5.2. Configuração do ProGuard

O mesmo resultado também seria possível se utilizarmos as declarações direto no arquivo de configurações do *ProGuard*. Quando criamos um projeto, além dos arquivos *default* de configuração do projeto, temos também um arquivo na raiz chamado *'proguard-rules.pro'*, que é onde devemos inserir todas as informações customizadas da aplicação.

No nosso exemplo, para mantermos a *MainActivity* sem utilizar a *annotation* @Keep, podemos utilizar o seguinte código:

```
-keep public class MainActivity
```

6.6. Arquivos gerados

Acessando a mesma pasta onde os *outputs* foram gerados, podemos observar que além da pasta *apk*, há também a pasta *mapping*, que possui 4 arquivos em formato *.txt* com informações muito relevantes sobre o processo que o *ProGuard* aplicou. Cada arquivo tem a sua devida importância e vale a pena guardá-los junto com cada *build* gerada.

- **dump.txt**
  - Descreve a estrutura interna de todos os arquivos de classe no *APK*.
- **mapping.txt**
  - Fornece uma conversão entre os nomes original e ofuscado da classe, do método e do campo.
- **seeds.txt**
  - Lista as classes e os membros que não foram ofuscados.
- **usage.txt**
  - Lista os códigos que foram removidos do *APK*.

6.7. Testando a aplicação

Com a *build* em mãos, para testar nossa aplicação no emulador basta arrastar o arquivo *apk* para dentro do emulador, isso fará com que ele seja instalado, porém, não será executado automaticamente. Como nossa aplicação é bem simples e não possui nenhuma biblioteca ou funcionalidade adicional, não há

muito o que testar, o funcionamento será perfeito assim como esperado, mesmo na aplicação com o passo *optimize* realizado.

6.8. Stack Trace do código ofuscado

Dentro os arquivos gerados, um dos mais importantes é o *mapping*, pois ele nos ajudará na hora de fazer o *debugging* de erros ocorridos em nossa aplicação, principalmente quando ela estiver publicada na loja e os erros ocorrem no dispositivo dos usuários.

Quando um erro ocorre, recebemos a informação completa do caminho do arquivo onde a falha ocorreu, geralmente contendo o pacote e a classe em questão. Como estamos trabalhando com um arquivo ofuscado, essa mensagem de erro também bem ofuscada, informando por exemplo que o erro aconteceu no método **j** da classe **a**, em uma variável **x**, o que não faz muito sentido.

6.8.1. Stack Trace Local

Para realização da Stack Trace local de um código ofuscado, é necessário utilizar um *plugin* que realiza a desofuscação. A documentação do *Google* recomenda a utilização do arquivo *'retrace.bat'* para *Windows* e *'retrace.sh'* para *MAC* ou *Linux*, no entanto, junto com a *SDK* do *Android*, na mesma pasta do arquivo de *retrace*, existe um outro arquivo chamado *proguardgui* que executa uma interface de usuário que dentre outras funcionalidades possui uma função de *retrace*. O arquivo pode ser localizado no diretório informado abaixo e deverá ter uma interface semelhante à figura 10.

Diretório: '...\Android\Sdk\tools\proguard\bin\proguardgui.bat'

Para realizar a *stack trace* de maneira local, copie o código do erro causado que contenha os nomes ofuscados; no *ProGuard GUI*, clique na opção *'Retrace'* no menu esquerdo; cole o código no campo *'Obfuscated stack trace'*; selecione o local do arquivo *mapping.txt* do *APK* em questão; e clique em *'ReTrace!'*.

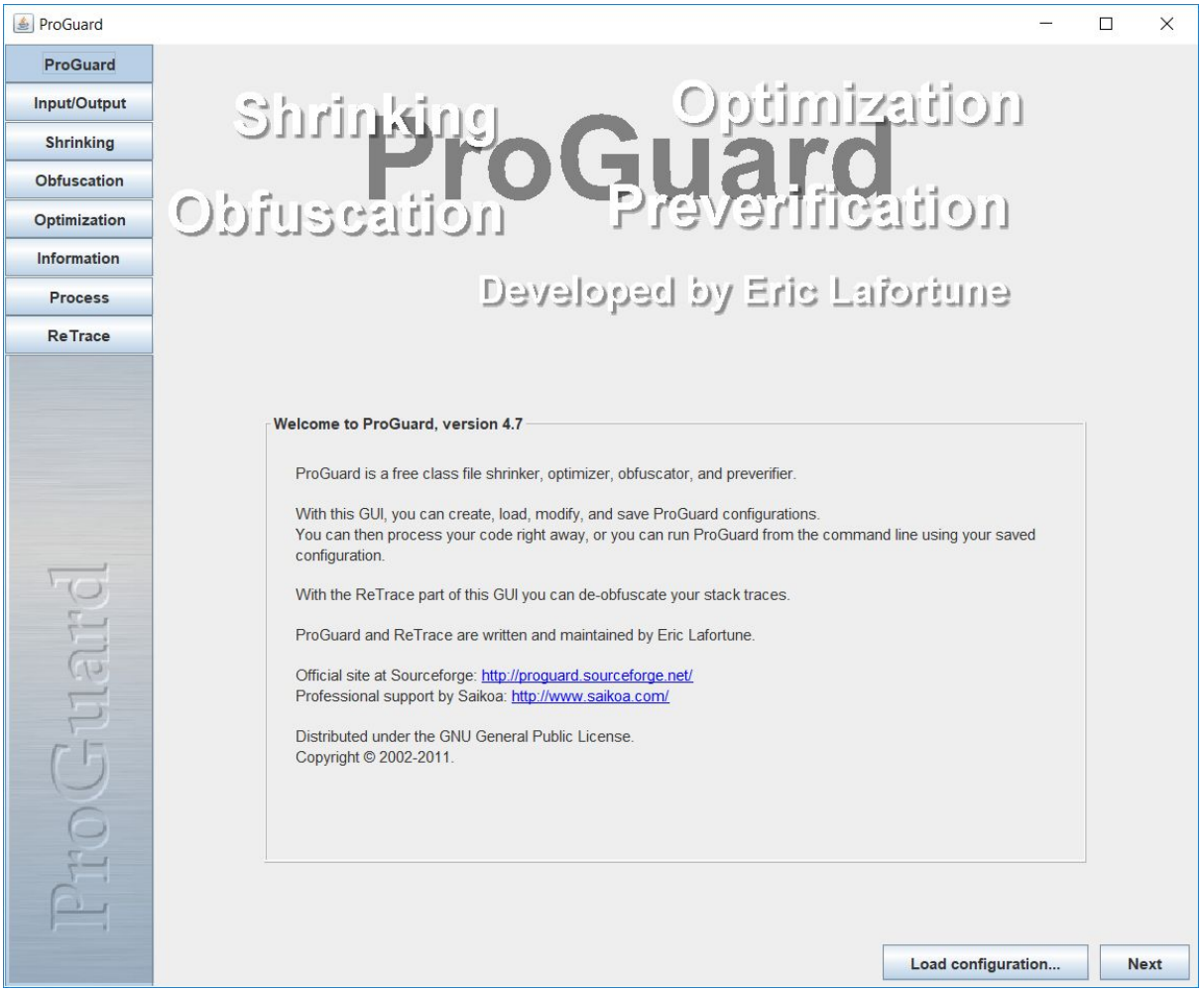


Figura 10: *ProGuard GUI*
Fonte: autor

6.8.2. Stack Trace Play Store

Para contornar isso, podemos enviar o arquivo de *mapping* da versão específica para a *Play Store*, isso nos permitirá acessar os registros de erros já com as nomenclaturas desofuscadas. Para fazer isso, basta acessar o painel *Google Play Console* e selecionar o aplicativo em questão. Depois, basta entrar na opção "Android Vitals | Arquivos de desofuscação" e realizar o *upload* do arquivo *mapping* correspondente, conforme mostrado na figura 11.

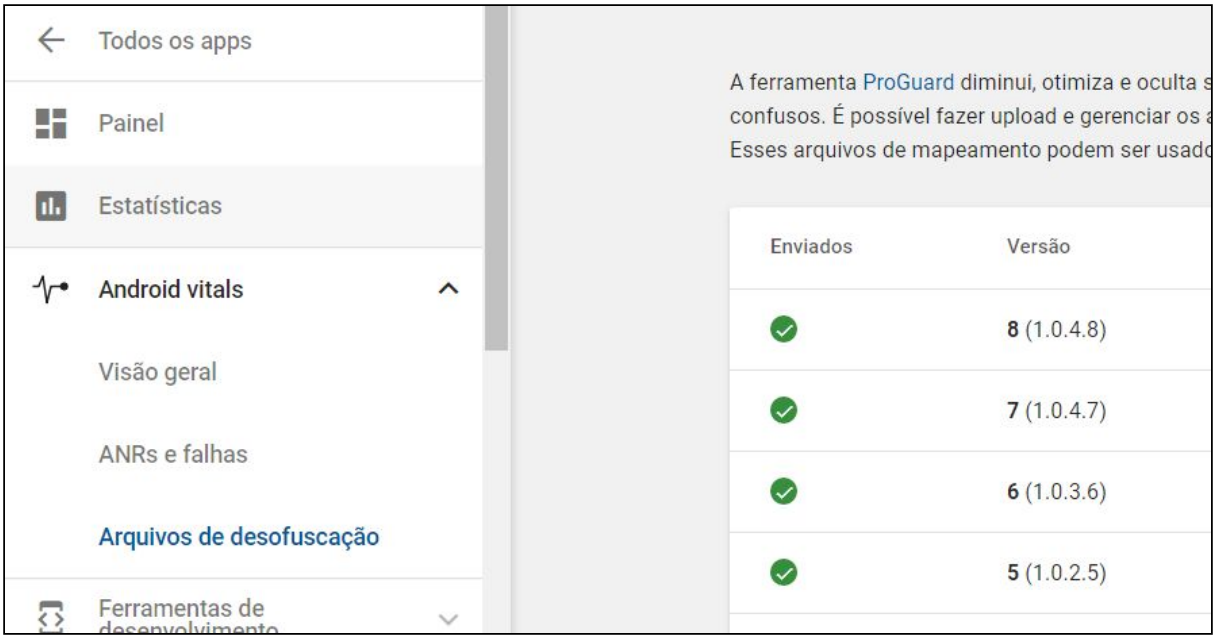


Figura 11: Página de envio de arquivos de desofuscação
Fonte: autor - *Google Play Console*

6.9. Redução de Recursos

Além de reduzir os códigos, também é possível aplicar o *ProGuard* para a redução de recursos, como imagens, vídeos, *layouts*, etc. Para isso, basta acessar o arquivo *build.gradle* do *app* e adicionar a opção *'shrinkResources true'*. Um detalhe é que para essa opção funcionar, é necessário que o *minifyEnabled* também esteja como *true*.

```
buildTypes {
   debug {
     shrinkResources true
     minifyEnabled true
     proguardFiles
getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
   }
}
```

É recomendado realizar essa alteração apenas após ter feito um primeiro *build* com o *ProGuard* ativado e a aplicação estar devidamente testada. Somente

após isso deve-se ativar o *shrink resources*, visto que alguns *bugs* adicionais podem ocorrer nesse passo.

Note que um novo arquivo *.txt* de *log* será gerado juntamente com os outros quatro arquivos informados anteriormente, contendo informações de todos os recursos ofuscados.

6.10. Integrando bibliotecas

Como já citado anteriormente, o vídeo 3 fala sobre como utilizar o *ProGuard* da maneira correta para manter aplicações cada vez menores e mais eficientes. Durante a apresentação, um exemplo de um *app* bem simples foi bem interessante e mostrou como podemos fazer para entender de fato com o *ProGuard* funciona e quais as principais abordagens que encontramos na *web* quando estamos integrando bibliotecas em nossos projetos, um dos principais motivos de preocupação quando vamos compilar nosso *APK*.

Utilizarei esse exemplo para mostrar algumas características do desenvolvimento de uma aplicação real utilizando o *ProGuard* integrado com bibliotecas.

Manteremos o mesmo projeto utilizado até o momento neste capítulo, apenas alteraremos algumas configurações para iniciar nossos testes.

Primeiro desativaremos o *ProGuard*.

```
buildTypes {
   debug {
     shrinkResources false
     minifyEnabled false
     proguardFiles
getDefaultProguardFile("proguard-android.txt"),
"proguard-rules.pro"
   }
   // ...
}
```

Depois, adicionaremos as duas bibliotecas que utilizaremos nesse projeto, a biblioteca *Guava* (apenas para ilustrar) e a biblioteca *AndroidPlot*, uma entre tantas disponíveis para a exibição de gráficos simples.

```
// Guava
```

```
implementation "com.google.guava:guava:26.0-android"

// Android Plot
implementation "com.androidplot:androidplot-core:1.5.5"
```

Após realizar o *sync*, criaremos nosso *layout* básico no arquivo de *layout* da *MainActivity*.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
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"
  tools:context=".MainActivity">

  <TextView
     android:id="@+id/textView"
     android:layout_width="0dp"
     android:layout_height="wrap_content"
     android:layout_marginEnd="8dp"
     android:layout_marginLeft="8dp"
     android:layout_marginRight="8dp"
     android:layout_marginStart="8dp"
     android:layout_marginTop="8dp"
     android:gravity="center"
     android:text="TextView"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />

  <com.androidplot.xy.XYPlot
     android:id="@+id/plot"
     style="@style/APDefacto.Light"
     android:layout_width="0dp"
     android:layout_height="0dp"
```

```
    android:layout_marginTop="8dp"
    app:domainTitle="domain"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:lineLabelRotationBottom="-45"
    app:lineLabels="left|bottom"
    app:rangeTitle="range"
    app:title="A Simple XY Plot" />

</android.support.constraint.ConstraintLayout>
```

E por fim, adicionaremos o código da nossa *Main Activity*, que irá carregar alguns dados no gráfico construído.

```
import android.graphics.Color
import android.graphics.DashPathEffect
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.view.View
import com.androidplot.util.PixelUtils
import com.androidplot.xy.*
import com.google.common.base.Joiner
import kotlinx.android.synthetic.main.activity_main.*
import java.text.FieldPosition
import java.text.Format
import java.text.ParsePosition

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    // Apenas um exemplo de utilização do Joiner do Guava
    val joiner = Joiner.on(" ").skipNulls()
```

```
    textView.text = joiner.join("Movile", null, "Next",
"Android")

    val plot = findViewById<View>(R.id.plot) as XYPlot

    val domainLabels = listOf<Number>(1, 2, 3, 6, 7, 8, 9,
10, 13, 14)
    val series1Numbers = listOf<Number>(1, 4, 2, 8, 4, 16,
8, 32, 16, 64)
    val series2Numbers = listOf<Number>(5, 2, 10, 5, 20, 10,
40, 20, 80, 40)

    val series1 = SimpleXYSeries(series1Numbers,
SimpleXYSeries.ArrayFormat.Y_VALS_ONLY, "Series1")
    val series2 = SimpleXYSeries(series2Numbers,
SimpleXYSeries.ArrayFormat.Y_VALS_ONLY, "Series2")

    val series1Format = LineAndPointFormatter(Color.BLACK,
Color.WHITE, Color.CYAN, null)
    val series2Format = LineAndPointFormatter(Color.BLACK,
Color.WHITE, Color.CYAN, null)

    series2Format.linePaint.pathEffect =
DashPathEffect(floatArrayOf(PixelUtils.dpToPix(20f),
PixelUtils.dpToPix(15f)), 0f)

    series1Format.interpolationParams =
CatmullRomInterpolator.Params(10,
CatmullRomInterpolator.Type.Centripetal)
    series2Format.interpolationParams =
CatmullRomInterpolator.Params(10,
CatmullRomInterpolator.Type.Centripetal)

    plot.addSeries(series1, series1Format)
    plot.addSeries(series2, series2Format)
```

```
plot.graph.getLineLabelStyle(XYGraphWidget.Edge.BOTTOM).format = object : Format() {
        override fun format(obj: Any, toAppendTo: StringBuffer, pos: FieldPosition): StringBuffer {
            val i = Math.round((obj as Number).toFloat())
            return toAppendTo.append(domainLabels[i])
        }

        override fun parseObject(source: String, pos: ParsePosition): Any? {
            return null
        }
    }
  }
}
```

Execute a aplicação no emulador para garantir que tudo está funcionando conforme a figura 12. Com tudo testado, realize também o *build* do *APK* sem redução de código e note o tamanho do arquivo. No meu caso, o arquivo ficou com 3.040 KB. Um ganho de 1.023 KB em relação a última compilação sem a redução do *ProGuard*.

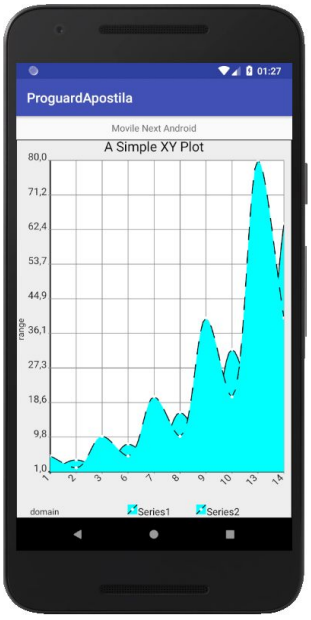


Figura 12: Aplicação simples para demonstração do *ProGuard*
Fonte: autor - *Android Emulator (API 27)*

Podemos realizar uma análise do nosso *APK* utilizando a ferramenta *APK Analyzer* presente no próprio *Android Studio*, selecionando através do menu "Build | Analyze APK...", conforme é possível verificar na figura 13.

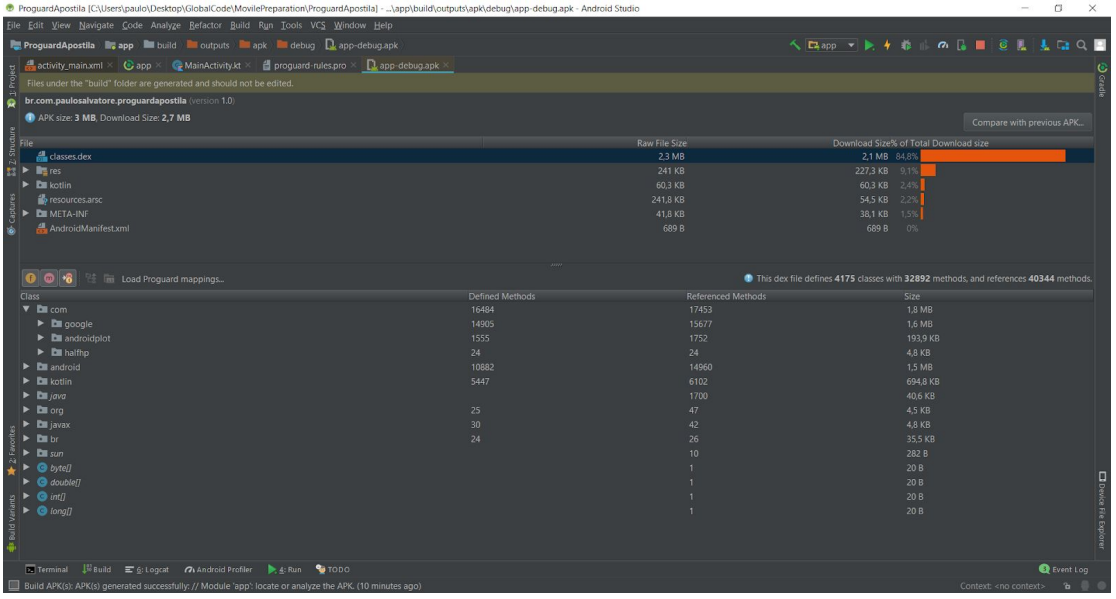


Figura 13: Análise do *APK* sem alteração do *ProGuard*
Fonte: autor - *Android Studio - APK Analyzer*

Olhando para os resultados, podemos ver que a biblioteca *Guava* ocupa cerca de 1,6 MB de tamanho do *APK*, enquanto a biblioteca *AndroidPlot* ocupa quase 200 KB e nossos códigos declarados na *MainActivity* ocupam 35,5 KB.

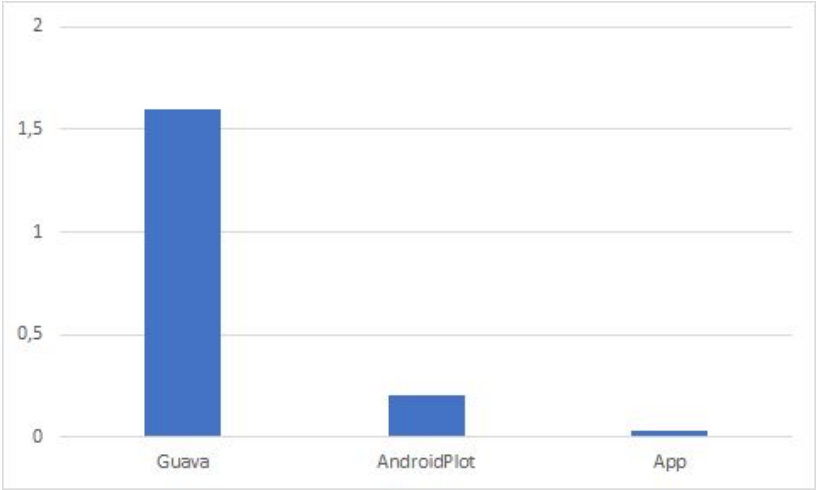


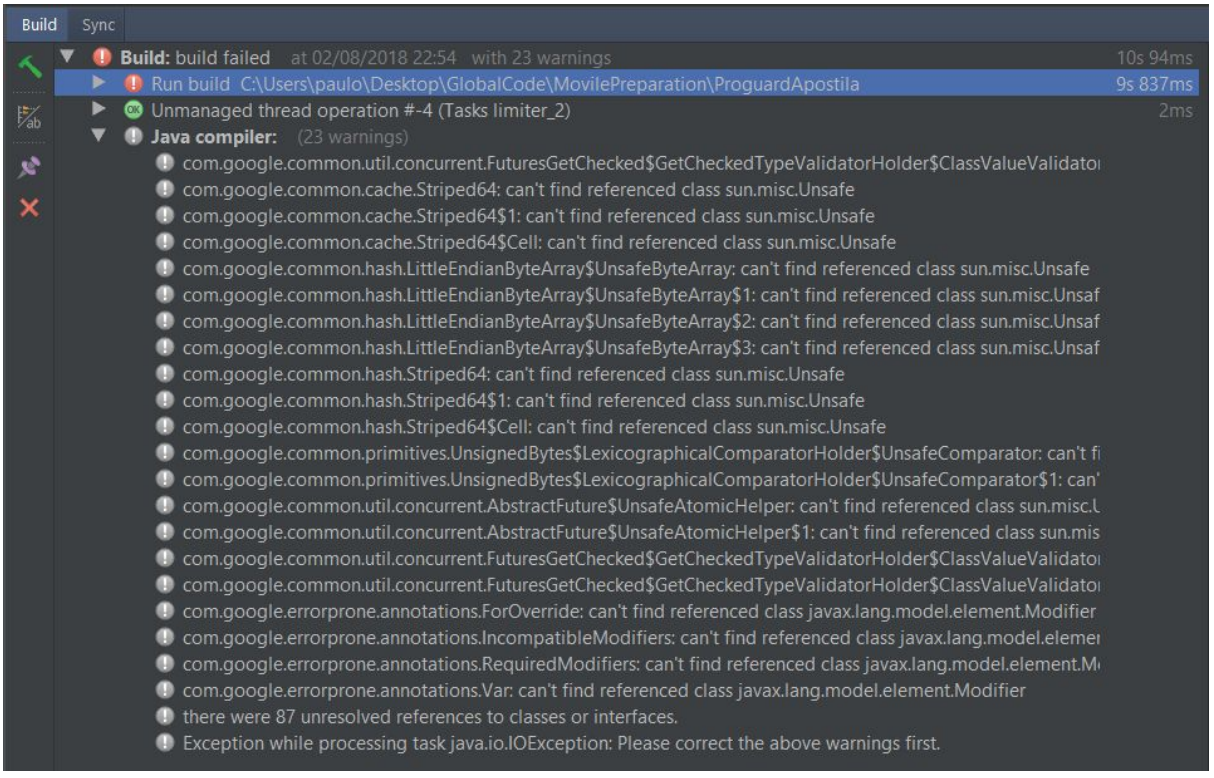
Figura 14: Comparação da ocupação de tamanho no *app*
Fonte: autor

Mesmo que no nosso exemplo o *Guava* esteja apenas servidor para fornecer uma pequena ferramenta de teste, ele acaba dominando a maior parte da aplicação, pois carrega todo seu conteúdo junto, mesmo que não sendo utilizado. É aí que o *ProGuard* entra em ação.

Para reduzir esse tamanho, vamos ativar o *ProGuard*.

```
buildTypes {
  debug {
     shrinkResources false
     minifyEnabled true
     proguardFiles
getDefaultProguardFile("proguard-android.txt"),
"proguard-rules.pro"
  }
  /// ...
}
```

Após fazer o *sync*, podemos realizar o *build* do nosso *app* e esperar a mágica acontecer...



Como é possível notar, recebemos uma série de *warnings* e diversas referências não resolvidas, sem contar os possíveis erros em tempo de execução que nem sempre são capturados durante a compilação.

Em uma pesquisa rápida na *internet* em como resolver isso a solução parece muito simples.

```
-dontwarn **
```

Escrevemos isso na configuração e magicamente os erros desaparecem e nosso *APK* compila. Executamos a aplicação e recebemos um *crash* antes mesmo de entrar. O que basicamente fizemos foi falar para o *ProGuard* fingir que nada aconteceu e realizar o processo sem dizer nada pra gente.

6.11. Entendendo o funcionamento do ProGuard

Para utilizar o *ProGuard* da melhor maneira possível, precisamos entender o seu funcionamento. Começamos por analisar como o *ProGuard* sabe o que está ou não sendo utilizado. Em uma aplicação *Android* começamos pelo manifesto, todas as referências feitas no manifesto automaticamente irão gerar *keep rules* para que suas informações sejam mantidas, buscando sempre o seu construtor. Apesar do *ProGuard* não conseguir entender a informação presente no manifesto, o próprio *Android* irá construir as referências automaticamente, informando para o *ProGuard* o que ele deve manter, através do *AAPT (Android Asset Packaging Tool)*.

Iniciando a análise na *MainActivity*, percebemos que ela não possui um construtor, sabemos que ela na verdade é construída no método *onCreate()*. Analisando as classes que a *MainActivity* estender, eventualmente iremos encontrar a classe *Activity*. Essa classe em questão pertence a *Android System Library*, ou seja, qualquer classe pertencente à biblioteca principal será considerada, fazendo com que todos os seus métodos, incluindo o *onCreate()* também sejam considerados, inclusive todas as referências das subclasses que sobrescrevem ele.

Analisando o método *onCreate()* da *MainActivity*, podemos observar uma chamada do método *findViewById()*.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
```

```
//...
    val plot = findViewById<View>(R.id.plot) as XYPlot
//...
}
```

O funcionamento desse método consiste em tempo de execução retorna um objeto em algum lugar de nossa *view* baseado no *layout*, algo que o *ProGuard* não consegue entender, pois afinal, esse *ID* está definido no arquivo *xml*, que assim como o manifesto da aplicação, o *ProGuard* não lê. Novamente, o *AAPT* entra em ação, fazendo com que essa classe referenciada também seja mantida, no nosso caso, *XYPlot*.

Esse é o fluxo de análise de código, consiste em entender exatamente como cada classe é solicitada e como o sistema realiza suas chamadas para sabermos quando algo é necessário ou não. Mesmo assim, ainda temos diversos erros durante o *build*.

A grande resposta para isso é que a análise que fazemos é muito diferente do que a *virtual machine* faz. A primeira diferença são as *annotations*, pois elas não são importantes para a *Android Virtual Machine* em tempo de execução, a não ser que estejamos usando *Reflection*, portanto, se uma annotation estiver faltando, o código ainda será executado, porque ele não é nem considerado. Em comparação, o *ProGuard* precisa entender as *annotations*, pois elas podem ser parte de uma *keep rule*, portanto, ela irá analisá-las para entender sua hierarquia, caso isso não dê certo, receberemos um *warning* sobre isso, sendo uma causa comum dos *warnings*.

Um outro exemplo pode ser conferido no trecho a seguir, onde parte da biblioteca *Guava* utiliza *ClassValue*, que é um conceito do *Java 7* que não está disponível na plataforma *Android*, fazendo com que esse código provoque uma falha durante a execução.

```
private static final ClassValue<Boolean> isValidClass =
   new ClassValue<Boolean>() {
     @Override
     protected Boolean computeValue(Class<?> type) {

checkExceptionClassValidity(type.asSubclass(Exception.class))
;
       return true;
```

```
    }
  };
```

Se nos perguntarmos porque esse código funciona, a resposta é que os criadores do *Guava* utilizam de um artifício para esconder as classes que estão faltando, fazendo com que a classe tente ser instanciada via *Reflection* e caso falhe, retorne uma implementação alternativa.

```
static GetCheckedTypeValidator getBestValidator() {
 try {
   Class<?> theClass =
Class.forName(CLASS_VALUE_VALIDATOR_NAME);
   return (GetCheckedTypeValidator)
theClass.getEnumConstants()[0];
 } catch (Throwable t) { // ensure we really catch
*everything*
   return weakSetValidator();
 }
}
```

Isso será óbvio para a *VM* quando o código for executado, afinal, apresentará uma falha que será pega pelo *catch* e retornará uma outra informação. No entanto, o *ProGuard* não consegue entender esse código pois é muito complicado para ser analisado de forma estática.

Finalmente, para resolver todos os erros precisamos analisar cada informação para entender de onde ela vem e porque o *ProGuard* está nos alertando sobre isso. Após realizar todo esse trabalho de análise, chegamos a uma configuração inicial de *keep rules*.

```
-dontwarn com.google.errorprone.**
-dontwarn sun.misc.Unsafe
-dontwarn java.lang.ClassValue
```

Adicionando essas regras, conseguimos fazer com que nossa aplicação seja compilada. No entanto, quando executamos a aplicação, notamos que temos um novo *crash*. A partir da nossa análise, as regras que escrevemos parecem fazer sentido, pois afinal sabemos que o que pedimos para o *ProGuard* não nos alertar

são coisas que sabemos que realmente não causarão problemas durante o tempo de execução.

No entanto, o problema típico quando utilizamos essa abordagem são as *Reflections*, que por definição utiliza um valor dinâmico durante tempo de execução para carregar um método ou uma classe, algo que não será entendido pela análise estática feita pelo *ProGuard*.

Para consertar isso, novamente vamos na *internet* e pesquisamos o que nossas bibliotecas sobre utilização junto com o *ProGuard*. Olhando na página do *GitHub* da biblioteca *AndroidPlot* [ref 22], podemos observar a seguinte regra:

```
-keep class com.androidplot.** { *; }
```

Isso significa que o *ProGuard* não irá tocar em nenhuma informação dentro do pacote *com.androidplot*. A princípio isso irá corrigir o problema, porém, não nos permitirá otimizar nossa aplicação no nível que podemos. Para contornar isso, precisamos analisar novamente o que está causando os erros, investigar o conteúdo das classes e entender quais regras devemos escrever.

O primeiro lugar que devemos olhar é o *logcat*, procurando pelo erro de execução causado na última vez que tentamos rodar a aplicação.

```
java.lang.RuntimeException: Unable to start activity
package.MainActivity
Binary XML file line #24:
Error inflating class com.androidplot.xy.XYPlot
//...
Caused by: java.lang.RuntimeException:
        at android.content.res.TypedArray.getBoolean
        at com.androidplot.a.b(Unknown Source:3)
        at com.androidplot.a.a(Unknown Source:173)
        at com.androidplot.a.a(Unknown Source:76)
        at com.androidplot.a.<init>(Unknown Source:131)
        at com.androidplot.xy.XYPlot.<init>
```

O que acontece é que durante a construção do *layout*, assim que a *view XYPlot* tenta ser iniciada via *Reflection*, o sistema não consegue encontrar a classe que teve seu nome alterado. Aplicando o *ReTrace* na *stack trace* acima é possível

verificar que a série de problemas foi causado a partir da classe *Plot* (estendida pela *XYPlot*), que por sua vez possui diversas subclasses.

Portanto, devemos manter essas classes intactas, escrevendo as seguintes *keep rules:*

```
-keep,allowshrinking class com.androidplot.Plot {}
-keep,allowshrinking class * extends com.androidplot.Plot {}
```

Note que além da declaração *-keep*, escrevemos também *allowshrinking*, que permitirá que essa classe seja removida caso ela não esteja sendo usada em nenhum lugar do código.

Com isso nossa aplicação deverá rodar, com o tamanho da aplicação reduzido consideravelmente.

Vamos supor agora que queremos melhorar a visualização do nosso gráfico segundo a documentação da biblioteca e criar arquivos *xml* para realizar essa customização. Para testar, substitua o seguinte código na *MainActivity*.

```
val series1Format = LineAndPointFormatter(Color.BLACK,
Color.WHITE, Color.CYAN, null)
val series2Format = LineAndPointFormatter(Color.BLACK,
Color.WHITE, Color.CYAN, null)

val series1Format = LineAndPointFormatter(this,
R.xml.line_point_formatter_with_labels)
val series2Format = LineAndPointFormatter(this,
R.xml.line_point_formatter_with_labels_2)
```

Crie um novo diretório na pasta *resources* chamado *xml* e dois arquivos, *'line_point_formatter_with_labels.xml'* e *'line_point_formatter_with_labels_2.xml'* com os seguintes conteúdos, respectivamente:

Arquivo 1

```
<?xml version="1.0" encoding="utf-8"?>
<config
  linePaint.strokeWidth="5dp"
  linePaint.color="#00AA00"
  vertexPaint.color="#007700"
```

```
  vertexPaint.strokeWidth="20dp"
  fillPaint.color="#00000000"
  pointLabelFormatter.textPaint.color="#CCCCCC"/>
```

Arquivo 2

```
<?xml version="1.0" encoding="utf-8"?>
<config
  linePaint.strokeWidth="5dp"
  linePaint.color="#0000AA"
  vertexPaint.strokeWidth="20dp"
  vertexPaint.color="#000099"
  fillPaint.color="#00000000"
  pointLabelFormatter.textPaint.color="#CCCCCC"/>
```

Executando nossa aplicação, percebemos um novo erro:

```
Caused by: com.a.a.b: Error while parsing key:
linePaint.strokeWidth value: 5.0dip
        at com.a.a.a.a(Unknown Source:161)
        at com.a.a.a.a(Unknown Source:26)
        at com.a.a.a.a(Unknown Source:55)
        at com.a.a.a.a(Unknown Source:8)
        at com.androidplot.b.e.a(Unknown Source:0)
        at com.androidplot.b.e.<init>(Unknown Source:6)
        at com.androidplot.xy.aj.<init>(Unknown Source:0)
        at com.androidplot.xy.o.<init>(Unknown Source:0)
        at .MainActivity.onCreate(Unknown Source:344)
// ...
Caused by: java.lang.NoSuchMethodException: No such public
method (case insensitive): getlinePaint in class
com.androidplot.xy.o
        at com.a.a.a.a(Unknown Source:51)
        at com.a.a.a.c(Unknown Source:17)
        at com.a.a.a.a(Unknown Source:52)
        at com.a.a.a.a(Unknown Source:0)
```

Novamente não diz muita coisa, mas nos dá um lugar para procurar. Buscando pela mensagem informada, encontramos um método chamado *getMethodByName()* que utiliza *Reflection* para encontrar um método em uma determinada classe.

```
private static Method getMethodByName(Class clazz, final
String methodName)
        throws NoSuchMethodException {
    final Method[] methods = clazz.getMethods();
    for (Method method : methods) {
        if (method.getName().equalsIgnoreCase(methodName)) {
            return method;
        }
    }
    throw new NoSuchMethodException("No such public method
(case insensitive): " +
            methodName + " in " + clazz);
}
```

Para contornar essa situação, podemos informar para o *ProGuard* manter os *getters* e *setters* das classes, para que caso algum seja usado pelo *xml* para carregar algum atributo, esse tipo de erro não aconteça mais.

```
-keepclassmember class com.androidplot.** {
  *** get*();
  void set*(***);
}
```

Com isso, executamos nossa aplicação e voilà! Tudo está funcionando perfeitamente e com o tamanho reduzido, conforme é possível ver na figura 15..

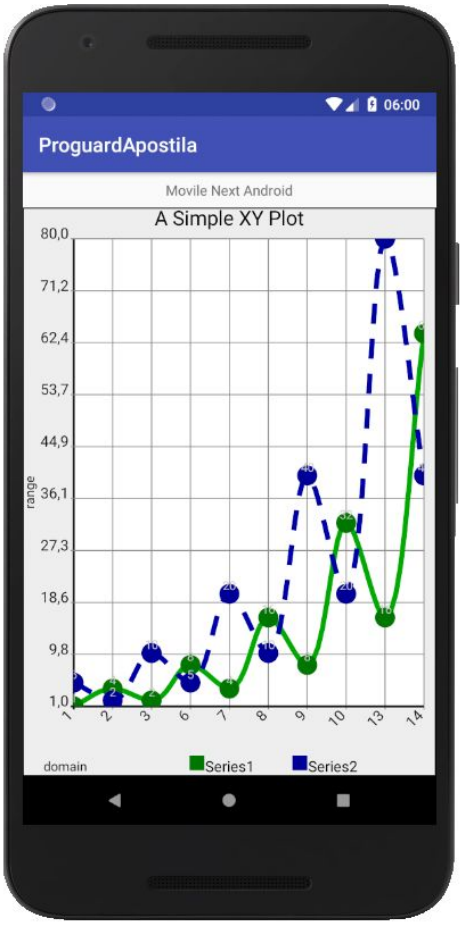


Figura 16: Aplicação funcionamento perfeitamente com *ProGuard* aplicado
Fonte: autor - *Android Emulator (API 27)*

Como é possível perceber, em um primeiro momento não parece tão óbvio o que devemos fazer, afinal, é preciso entender o que um código escrito por outra pessoa está fazendo para depois entender exatamente o que queremos manter. No entanto, o *Google* recomenda para que todos os desenvolvedores pensem nas regras do *ProGuard* sempre que estiverem construindo qualquer código, pois é muito mais fácil escrever essas regras enquanto o código está sendo produzido e você sabe exatamente o que precisa parar ele funcionar do que fazendo uma análise depois para encontrar as falhas.

Além disso, também recomenda aos desenvolvedores de bibliotecas que utilizem o *consumerProguardFiles*, que faz com que as regras do *ProGuard* sejam automaticamente referenciadas quando uma biblioteca é utilizada.

Com a nossa aplicação rodando, chegou a hora de fazer uma análise do nosso novo *APK*, conforme é possível visualizar na figura 17..

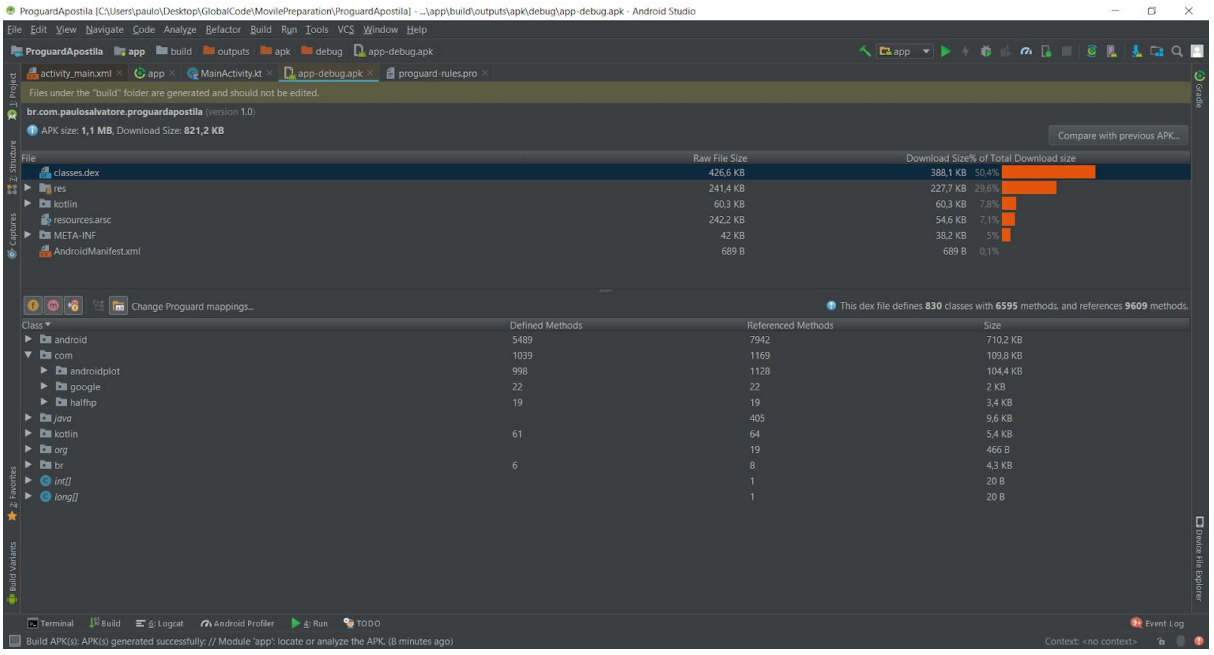
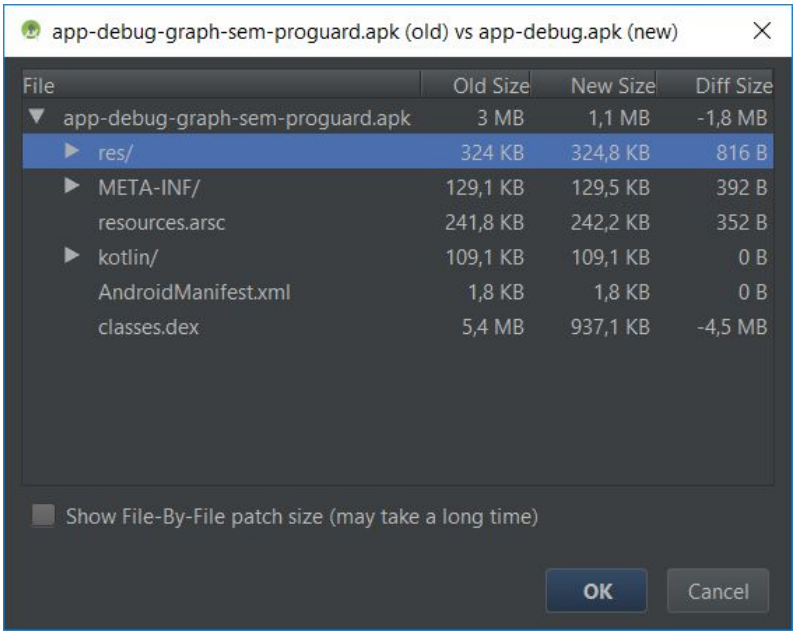


Figura 17: Análise do *APK* sem alteração do *ProGuard*
Fonte: autor - *Android Studio* - *APK Analyzer*

Comparando os resultados lado a lado, a diferença no diretório *classes.dex* é espantosa: 4.5MB!



Voltando para o nosso gráfico comparativo, podemos colocar lado a lado os valores atualizados da nossa aplicação.

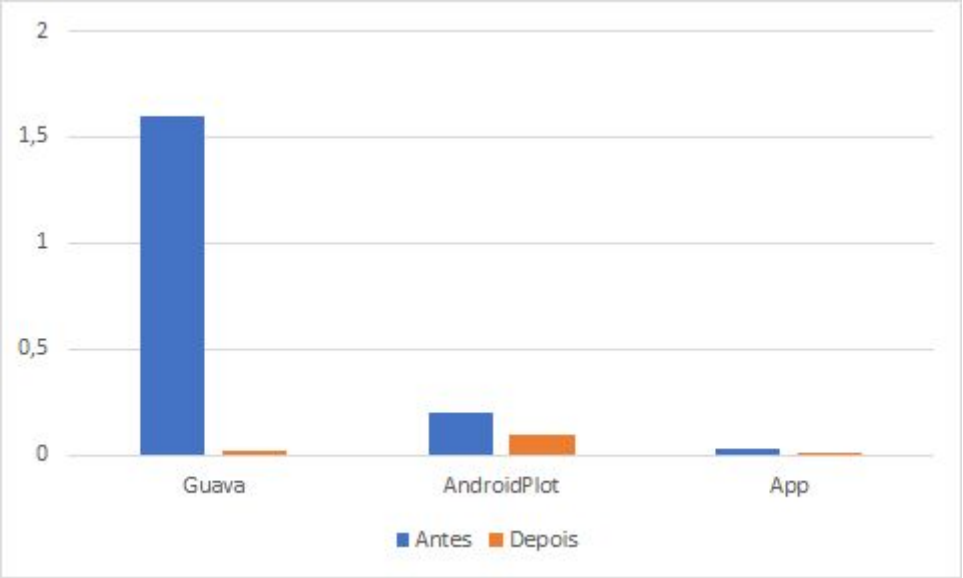
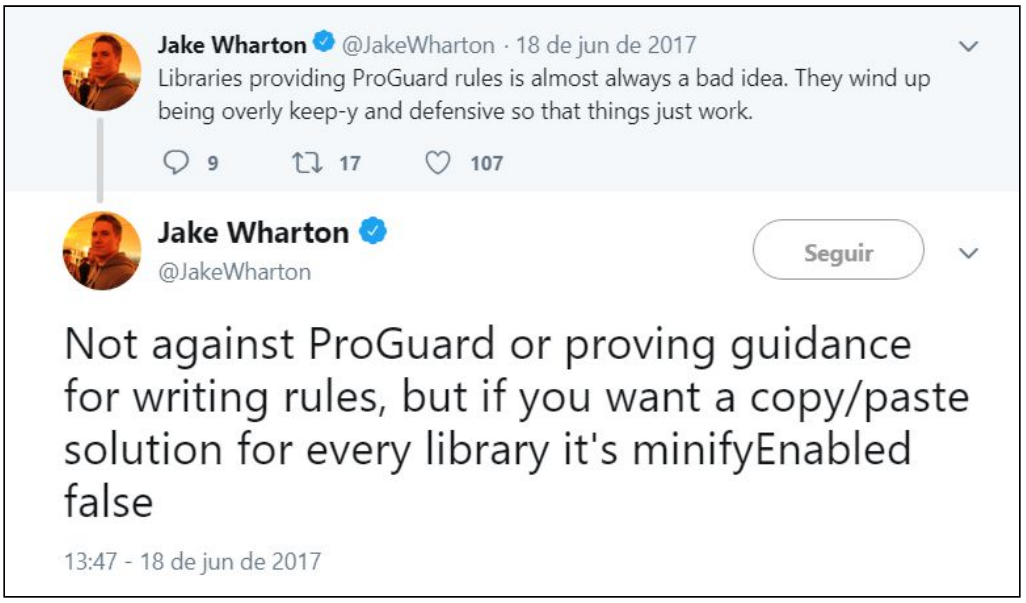


Figura 18: Comparação da ocupação de tamanho no *app*
Fonte: autor

Olhando para os resultados, podemos ver que a biblioteca *Guava* agora ocupa míseros 2 KB (afinal só estamos utilizando uma pequena declaração), enquanto a biblioteca *AndroidPlot* ocupa quase 100 KB e nossos códigos declarados na *MainActivity* caíram para 4 KB.

A princípio podemos entender esse trabalho realizado como algo complicado e trabalhoso, porém, o resultado é bastante satisfatório. Como o *Jake Wharton* publicou em seu *Twitter*, se há uma solução do *ProGuard* para qualquer biblioteca, é *'minifyEnabled false'*.



6.12. Controvérsias

O assunto de otimização de código é muito discutido no universo *Android*, enquanto alguns desenvolvedores amam e encorajam a sua utilização, outros odeiam e criticam o seu uso. No início os desenvolvedores eram mais resistentes em relação a sua utilização, porém, com o passar o tempo, acabaram tornando-se adeptos ao uso dessa tecnologia, sendo inclusive recomendada pelo *Google* em diversos *talks* e inclusive em sua documentação [ref 19].

Um caso interessante foi uma afirmação, em 2013, do desenvolvedor *Jake Wharton* que citou que nunca utilizou nem utilizaria o *ProGuard*, porém, quando questionado alguns anos depois comentou que alguns fatores como o tamanho elevado de algumas bibliotecas e o fato do *Android* passar a poder utilizar a opção de *multidex*, forçaram-no a voltar atrás em relação à declaração (figura 19).

Figura 19: *Jake Wharton*, em seu perfil no *Twitter*, sobre o *ProGuard*
Fonte: *Twitter* [ref 20]

6.13. DexGuard

Existem uma outra versão do *ProGuard*, se é que podemos chamar assim, chamada *DexGuard*, produzida pela mesma empresa e focada às aplicações *Android*. Diferente do *ProGuard* que é *open source*, a *DexGuard* é uma ferramenta paga voltada para o público empresarial de aplicativos que visa não apenas a redução e otimização do código como a proteção em maiores níveis.

Um artigo interessante publicado pela própria *GuardSquare* pode ser encontrado na referência 23.

7. Git Flow

Git é um dos assuntos mais importante para o desenvolvedor, independente de qual plataforma o projeto está sendo desenvolvido, principalmente quando estamos trabalhando com mais pessoas ou quando queremos ter um controle do que estamos fazendo no projeto.

Se quiser saber mais sobre o básico do *Git*, recomendo uma leitura adicional sobre esse funcionamento, incluindo todos os comandos e com exemplos bem interessantes. Neste capítulo falarei um pouco sobre o *git flow* [ref 24 e ref 25], uma ferramenta que visa otimizar o nosso fluxo de trabalho no *git*, padronizando algumas de nossas tarefas.

7.1. Branches

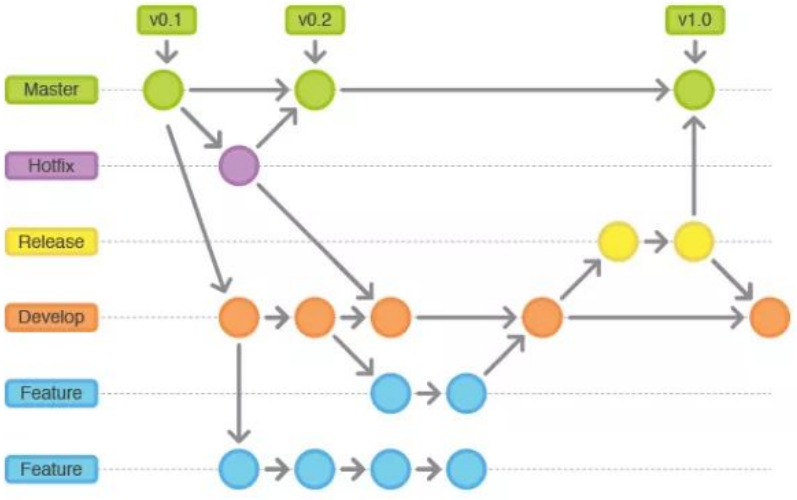


Figura 20: Branches presentes no *git flow*
Fonte: *coral.ufsm.br* [ref 25]

O *git flow* trabalha com a separação dos *branches* em primárias e secundárias (como e possível ver na figura 20):

- **Primárias**
  - **Master:** conteúdo do projeto pronto para ser utilizado em produção.
  - **Develop**: Contém o código que está pronto para entrar em produção.
- **Secundárias**
  - **Feature:** Novas funcionalidades para o sistema
  - **Release:** Novas versões do sistema
  - **Hotfix:** Conserto de *bugs* em produção

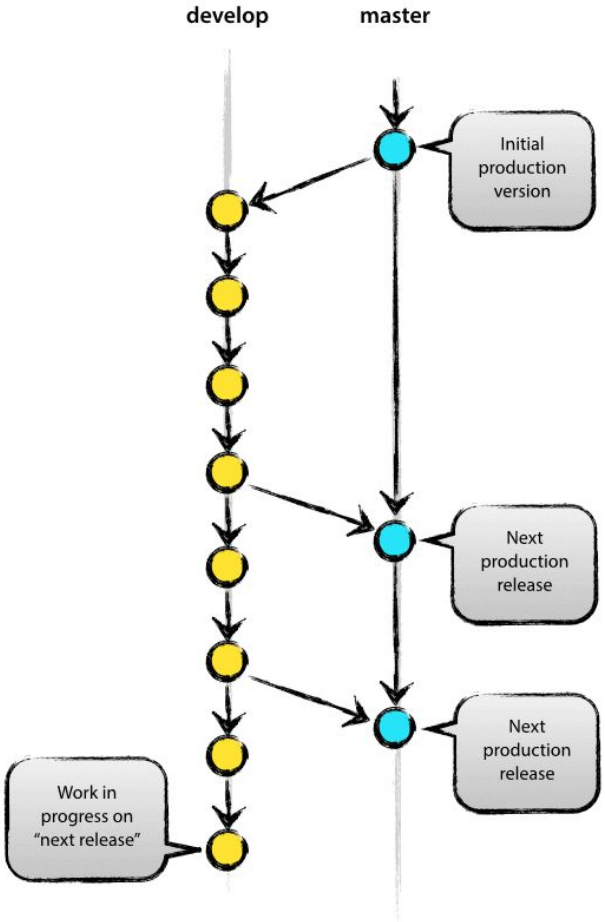


Figura 21: Relação entre o *branch develop* e o *master*
Fonte: *Vincent Driessen at nvie.com* [ref 26]

7.2. Comandos básicos

Após instalar a linha de comando do *git flow* no seu computador, podemos utilizar uma série de comandos presentes que irão realizar tarefas diretas no repositório *git* em questão.

7.2.1. Init

```
git flow init
```

O comando *git flow init* inicializará o *git flow*, construindo todos os *branches* conforme mencionados anteriormente.

```
PS C:\folder> git flow init
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [C:/folder/.git/hooks]
```

O próximo passo é criar o repositório em qualquer provedor de *git*, no meu caso, criarei no *GitHub*. Adicionamos a *URL* do *git* como *origin* e realizamos o *push* de ambos os *branches* criados anteriormente.

```
git remote add origin https://github.com/user/Repo.git
git push origin master
git push origin develop
```

```
PS C:\folder> git remote add origin
https://github.com/user/repo.git
PS C:\folder> git push origin master
Counting objects: 2, done.
Writing objects: 100% (2/2), 171 bytes | 171.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/user/Repo.git
 * [new branch]      master -> master
PS C:\folder> git push origin develop
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/user/Repo.git
 * [new branch]      develop -> develop
```

7.2.2. Feature Start

Agora nosso ambiente já está configurado e pronto para iniciar o desenvolvimento. A primeira coisa que geralmente fazemos em um projeto é implementar uma nova *feature*. Para isso, utilizaremos o comando *feature start* do *git flow*.

```
git flow feature start nova_feature
```

```
PS C:\folder> git flow feature start nova_feature
Switched to a new branch 'feature/nova_feature'

Summary of actions:
- A new branch 'feature/nova_feature' was created, based on
'develop'
- You are now on branch 'feature/nova_feature'

Now, start committing on your feature. When done, use:

     git flow feature finish nova_feature

PS C:\folder>
```

Esse comando irá criar um novo *branch* para a nova *feature* baseado no *branch develop* e automaticamente definirá como seu *branch* de trabalho. Para progredir, vamos adicionar um novo arquivo no repositório que chamarei de 'code.txt' e faremos o *commit* dele.

```
git add code.txt
git commit -m "Primeira feature"
```

```
PS C:\folder> git add code.txt
PS C:\folder> git commit -m "Primeira feature"
[feature/nova_feature 0505b17] Primeira feature
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 code.txt
PS C:\folder>
```

7.2.3. Feature Finish

Após finalizar todas as inserções de arquivos, podemos finalizar uma *feature* simplesmente utilizando o comando *feature finish* do *git flow*.

```
git flow feature finish nova_feature
```

```
PS C:\folder> git flow feature finish nova_feature
Switched to branch 'develop'
Updating 4fc0824..0505b17
Fast-forward
 code.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 code.txt
Deleted branch feature/nova_feature (was 0505b17).

Summary of actions:
- The feature branch 'feature/nova_feature' was merged into
'develop'
- Feature branch 'feature/nova_feature' has been locally deleted
- You are now on branch 'develop'
```

Ao realizar esse comando, é possível perceber que a nova *feature* foi mesclada *(merge)* com o *branch develop*; o *branch* da *feature* em questão foi removido localmente; e o *branch develop* passou a ser o nosso *branch* de trabalho atual.

7.2.4. Release Start

Com nossas features adicionadas e configuradas, podemos criar nosso primeiro *release* com o comando *release start* do *git flow*.

```
git flow release start 0.0.1
```

```
PS C:\folder> git flow release start 0.0.1
Branches 'develop' and 'origin/develop' have diverged.
And local branch 'develop' is ahead of 'origin/develop'.
Switched to a new branch 'release/0.0.1'

Summary of actions:
- A new branch 'release/0.0.1' was created, based on 'develop'
- You are now on branch 'release/0.0.1'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

     git flow release finish '0.0.1'
```

Um novo *branch* para o *release* será criado. Nesse momento ainda é possível realizar algumas correções ou pequenas modificações caso necessário.

7.2.5. Release Publish

Com o *release* criado, devemos publicar nosso trabalho utilizando o comando *release publish* do *git flow*. Isso irá automaticamente publicar o *release*.

```
git flow release publish 0.0.1
```

```
PS C:\folder> git flow release publish 0.0.1
Counting objects: 3, done.
Writing objects: 100% (3/3), 250 bytes | 250.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/paulosalvatore/GitTests.git
 * [new branch]      release/0.0.1 -> release/0.0.1
Branch 'release/0.0.1' set up to track remote branch
'release/0.0.1' from 'origin'.
Already on 'release/0.0.1'
Your branch is up to date with 'origin/release/0.0.1'.

Summary of actions:
- The remote branch 'release/0.0.1' was created or updated
- The local branch 'release/0.0.1' was configured to track the
remote branch
- You are now on branch 'release/0.0.1'
```

O *branch* remoto do *release* foi criado no repositório do *git*; o *branch* local foi configurado para rastrear o *branch* remoto; e nosso *branch* de trabalho passou a ser automaticamente para a *branch* recém criada.

7.2.6. Release Finish

Para finalizar nosso lançamento, devemos usar o comando *release finish* do *git flow*.

```
git flow release finish 0.0.1
```

```
PS C:\folder> git flow release finish 0.0.1
Branches 'master' and 'origin/master' have diverged.
And local branch 'master' is ahead of 'origin/master'.
Branches 'develop' and 'origin/develop' have diverged.
And local branch 'develop' is ahead of 'origin/develop'.
Already on 'master'
Switched to branch 'develop'
Already up to date!
```

```
Merge made by the 'recursive' strategy.
To https://github.com/user/Repo.git
 - [deleted]         release/0.0.1
Deleted branch release/0.0.1 (was 0505b17).

Summary of actions:
- Release branch 'release/0.0.1' has been merged into 'master'
- The release was tagged '0.0.1'
- Release tag '0.0.1' has been back-merged into 'develop'
- Release branch 'release/0.0.1' has been locally deleted; it has
been remotely deleted from 'origin'
- You are now on branch 'develop'
```

O *branch* de *release* foi mesclado no *branch* master e removido do repositório remoto e local. Para publicar as alterações, utilizamos o comando *push* nos *branches* de *develop* e *master* e finalizamos o fluxo básico do *git flow*.

```
git push origin master
git push origin develop
```

```
PS C:\folder> git push origin master
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 345 bytes | 345.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/user/Repo.git
   4fc0824..4517df6  master -> master
PS C:\folder> git push origin develop
Counting objects: 1, done.
Writing objects: 100% (1/1), 244 bytes | 244.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/user/Repo.git
   4fc0824..31502c1  develop -> develop
```

7.3. Resolvendo bugs em produção

Nem tudo é tão mágico quando estamos desenvolvendo e muitas vezes aparecem alguns *bugs* no *branch master* que devemos corrigir e publicar o mais rápido possível, sem ter que criar uma nova *feature* e um novo *release* para isso. Para suprir essa necessidade que existe o comando *hotfix* to *git flow*.

A construção desse comando é bem semelhante ao fluxo de *start*, *finish* e *publish* vistos anteriormente.

7.3.1. Hotfix Start

Para iniciar um *hotfix* é muito simples, mas realizar o comando *hotfix start* do *git flow*.

```
git flow hotfix start 0.1.1
```

```
PS C:\folder> git flow hotfix start 0.1.1
Switched to a new branch 'hotfix/0.1.1'

Summary of actions:
- A new branch 'hotfix/0.1.1' was created, based on 'master'
- You are now on branch 'hotfix/0.1.1'

Follow-up actions:
- Start committing your hot fixes
- Bump the version number now!
- When done, run:

     git flow hotfix finish '0.1.1'
```

Um novo *branch* para o *hotfix* será criado utilizando como base o *branch master;* e também será definido como nosso *branch* de trabalho.

Com o *hotfix* criado, basta inserir as modificações normalmente.

```
git add code.txt
git commit -m "Code Fixed!"
```

```
PS C:\folder> git add code.txt
PS C:\folder> git commit -m "Code Fixed!"
[hotfix/0.1.1 339c196] Code Fixed!
 1 file changed, 1 insertion(+)
```

7.3.2. Hotfix Finish

Após todas as modificações terem sido inseridas, basta finalizá-las utilizando o comando *hotfix finish* do *git flow*.

```
git flow hotfix finish 0.1.1
```

```
PS C:\folder> git flow hotfix finish 0.1.1
Branches 'master' and 'origin/master' have diverged.
And local branch 'master' is ahead of 'origin/master'.
Switched to branch 'develop'
Merge made by the 'recursive' strategy.
 code.txt | 1 +
 1 file changed, 1 insertion(+)
Deleted branch hotfix/0.1.1 (was 339c196).

Summary of actions:
- Hotfix branch 'hotfix/0.1.1' has been merged into 'master'
- The hotfix was tagged '0.1.1'
- Hotfix tag '0.1.1' has been back-merged into 'develop'
- Hotfix branch 'hotfix/0.1.1' has been locally deleted
- You are now on branch 'develop'
```

O *branch* do *hotfix* será finalizado e automaticamente mesclado no *branch master* e no *branch develop;* depois será automaticamente deletado localmente e o *branch* de trabalho será definido como *develop*. Realize o *push* para publicá-lo.

7.3.3. Hotfix Publish

Opcionalmente, antes de finalizar o *hotfix*, é possível publicar o *hotfix* antes de finalizá-lo, isso é último quando mais de um desenvolvedor está trabalhando no *hotfix* e precisa manter o repositório atualizado com as modificações de todos.

7.4. Pull

Além dos comandos específicos citados anteriormente, também é possível utilizar o comando *pull* do *git flow* para qualquer *feature*, *release* ou *hotfix*.

A funcionalidade do comando será exatamente a que estamos acostumados no *git* comum.

7.5. Resumo de Comandos

Por fim, podemos perceber que o *git flow* não possui uma quantidade gigantesca de comandos (vide figura 22) e basicamente consiste em um padrão de desenvolvimento que visa organizar melhor nossos repositórios e nosso fluxo de trabalho e facilitar a utilização de comandos *git*, evitando com que tenhamos que digitar o óbvio, como por exemplo fazer *checkout* sempre para mudar de *branch* quando a operação é sempre óbvia.

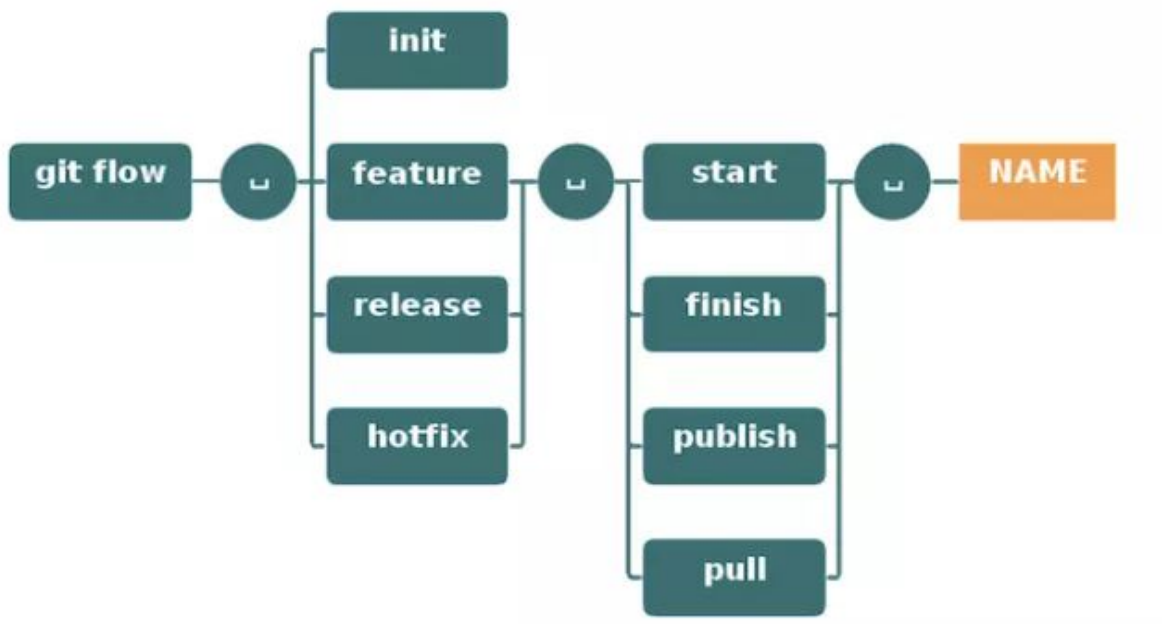


Figura 22: Resumo de Comandos do *Git Flow*
Fonte: *coral.ufsm.br* [ref 25]

8. Referências Bibliográficas

1. SOLID - https://en.wikipedia.org/wiki/SOLID (Acessado em 03 de Agosto de 2018)
2. The Clean Architecture - https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html (Acessado em 03 de Agosto de 2018)
3. Fundamentals of Testing - https://developer.android.com/training/testing/fundamentals (Acessado em 01 de Agosto de 2018)
4. Test Driven Development - https://pt.wikipedia.org/wiki/Test_Driven_Development (Acessado em 01 de Agosto de 2018)
5. JUnit4 - https://junit.org/junit4/ (Acessado em 01 de Agosto de 2018)
6. Roboletric - http://robolectric.org (Acessado em 01 de Agosto de 2018)
7. Robolectric test hangs up when started - Transferring from sonatype #3452 - https://github.com/robolectric/robolectric/issues/3452 (Acessado em 01 de Agosto de 2018)
8. Developing Android unit and instrumentation tests - Tutorial - http://www.vogella.com/tutorials/AndroidTesting/article.html (Acessado em 01 de Agosto de 2018)
9. Test from the command line - https://developer.android.com/studio/test/command-line (Acessado em 02 de Agosto de 2018)
10. Espresso - Training Section - https://developer.android.com/training/testing/espresso/ (Acessado em 02 de Agosto de 2018)
11. Depurar seu layout com o Inspetor de Layout - https://developer.android.com/studio/debug/layout-inspector (Acessado em 02 de Agosto de 2018)
12. Test UI for multiple apps - https://developer.android.com/training/testing/ui-testing/uiautomator-testing (Acessado em 02 de Agosto de 2018)
13. Mockito - http://site.mockito.org/ (Acessado em 02 de Agosto de 2018)
14. How to write good tests - https://github.com/mockito/mockito/wiki/How-to-write-good-tests (Acessado em 02 de Agosto de 2018)
15. Mockito Kotlin - https://github.com/nhaarman/mockito-kotlin/ (Acessado de 02 de Agosto de 2018)
16. Instrumentation Testing Robots - https://academy.realm.io/posts/kau-jake-wharton-testing-robots/ (Acessado em 02 de Agosto de 2018)
17. Espresso Robot Pattern in Kotlin - https://medium.com/android-bits/espresso-robot-pattern-in-kotlin-fc820ce250f7

(Acessado em 03 de Agosto de 2018)
18. ProGuard Manual - Introduction - https://www.guardsquare.com/en/products/proguard/manual/introduction (Acessado em 02 de Agosto de 2018)
19. Reduza seus códigos e recursos - https://developer.android.com/studio/build/shrink-code (Acessado em 02 de Agosto de 2018)
20. Jake Wharton's Twitter Account, Tweet about ProGuard - https://twitter.com/jakewharton/status/876541923036733442 (Acessado em 02 de Agosto de 2018)
21. ClassyShark Repo on GitHub - https://github.com/google/android-classyshark (Acessado em 02 de Agosto de 2018)
22. GitHub - Android Plot - Quick Start - https://github.com/halfhp/androidplot/blob/master/docs/quickstart.md
23. DexGuard versus ProGuard - https://www.guardsquare.com/pt-br/blog/dexguard-versus-proguard
24. Git Flow - https://github.com/nvie/gitflow (Acessado em 03 de Agosto de 2018)
25. Fluxo de Trabalho com Git – Como melhorar o trabalho em equipe - http://coral.ufsm.br/pet-si/index.php/fluxo-de-trabalho-com-git/ (Acessado em 03 de Agosto de 2018)
26. A successful Git branching model - https://nvie.com/posts/a-successful-git-branching-model/ (Acessado em 03 de Agosto de 2018)

9. Vídeos Recomendados

1. Test-Driven Development on Android with the Android Testing Support Library (Google I/O '17) - https://youtu.be/pK7W5npkhho (Acessado em 01 de Agosto de 2018)
2. Android Testing Patterns - https://www.youtube.com/playlist?list=PLWz5rJ2EKKc-6HWg_jyP0U1zrVLHn65b2 (Acessado em 02 de Agosto de 2018)
3. Effective ProGuard keep rules for smaller applications (Google I/O '18) - https://youtu.be/x9T5EYE-QWQ (Acessado em 02 de Agosto de 2018)
4. How Proguard Works - https://youtu.be/F9ymcWoDEtc (Acessado em 02 de Agosto de 2018)

10. Licença e termos de uso

11. Leitura adicional

Além de todas as referências, sugiro que leia alguns materiais adicionais para aumentar o seu conhecimento sobre alguns assuntos específicos.

11.1. SOLID

- Learning Path: SOLID Principles for Android - https://academy.realm.io/posts/learning-path-solid-principles-for-android/

11.2. Clean Architecture

- "Clean Architecture" para Android - https://medium.com/android-dev-br/clean-architecture-para-android-eb492513263e
- Real Life Clean Architecture - https://www.slideshare.net/mattiabattiston/real-life-clean-architecture-61242830

11.3. Testing

11.3.1. Unit & Instrumentation Tests

- Developing Android unit and instrumentation tests - Tutorial - http://www.vogella.com/tutorials/AndroidTesting/article.html

11.3.2. Espresso

- Android user interface testing with Espresso - Tutorial - http://www.vogella.com/tutorials/AndroidTestingEspresso/article.html

11.3.3. UI Automator

- Android cross component with the UI Automator framework - Tutorial - http://www.vogella.com/tutorials/AndroidTestingUIAutomator/article.html

11.3.4. Mockito

- Unit Testing & Mockito (I, II & III)
  - https://medium.com/@manuelvicnt/android-unit-testing-mockito-part-1-85a373c36bdf
  - https://medium.com/@manuelvicnt/android-unit-testing-mockito-part-2-cd966058a808

    - https://medium.com/@manuelvicnt/android-79549fcb2283
- Refcard: Mockito - A Simple, Intuitive Mocking Framework - https://dzone.com/refcardz/mockito?chapter=1

11.3.5. Extras

- UI/Application Exerciser Monkey - https://developer.android.com/studio/test/monkey
- Android Testing Codelab - https://codelabs.developers.google.com/codelabs/android-testing/index.html

11.4. ProGuard

- Thiengo - ProGuard Android - https://www.thiengo.com.br/proguard-android

11.5. Git

- Learn the workings of Git, not just the commands - https://www.thiengo.com.br/proguard-android
- 15 dicas para melhorar seu fluxo no Github - https://www.agatetepe.com.br/15-dicas-para-melhorar-seu-fluxo-no-github/