Classificação e análise de defeitos comuns em softwares e como mitigá-los utilizando diferentes práticas de desenvolvimento

Projeto de Graduação em Computação III

Vinícius Reis

UFABC

12 de dezembro de 2023

UFABC

Introdução

- Softwares de inúmeros tipos tem feito cada vez mais parte da vida das pessoas
- Desenvolvimento de software continua sendo um processo muito dependente do fator humano
- Ter clareza e entendimento das regras de negócio e o comportamento esperado para um software é apenas uma etapa para atingir uma entrega final totalmente funcional
- Tomamos falha ou defeito como uma condição anormal de um componente. No caso de software, seriam erros de lógica presentes na aplicação
- Já erros seriam qualquer diferença entre o comportamento especificado e o real

UFABC

Objetivos

- Compreender quais são os tipos de defeitos mais comuns em aplicações em produção
- Relacionar estes defeitos com o processo de desenvolvimento
- Explorar outras abordagens que possam ser adotadas para mitigar o risco de ocorrência destes defeitos
- Buscar ferramentas que possam minimizar ainda mais as vulnerabilidades da aplicação sem interferindo minimamente na arquitetura da solução

UFABC

Conteúdos

UFABC

Busca na literatura científica

- Selecionamos o acervo digital Scopus como para realizar a pesquisa pela relevância que o acervo possui
- Foram obtidos 368 artigos. Destes, analisando os títulos de todos os resultados obtidos, selecionamos 57 deles para classificação por relevância para este trabalho

UFABC

Reclassificação dos artigos

- Dos 57 artigos que selecionamos através da análise do resumo, classificando-os pela sua relevância demonstradas anteriormente, tivemos:
  - 18 artigos pouco relevantes
  - 39 possuem alguma relevância para o estudo, porém, em alguns destes artigos não fica evidente o quão relevante este pode ser
- Através da leitura do resumo, aplicando a nova definição de relevância que definimos para cada artigo, tivemos:
  - 15 artigos de pouca relevância
  - 7 artigos de média relevância
  - 17 artigos com alta relevância
- Após leitura rápida dos artigos mais relevantes, temos:
  - 4 não puderam ser lidos, pois, não foi obtido acesso a estes
  - 11 não tiveram relevância para este estudo
  - 2 se mostraram muito relevantes para o contexto deste trabalho
    - Além disso, um outro artigo citado demonstrou muita relevância para este contexto

UFABC

Reclassificação dos artigos

Descrição	Total de artigos
Artigos encontrados	368
Serão analisados pelo título	57
Pouco relevantes	18
Relevantes	39
Após reclassificação dos relevantes	
Pouco relevantes	15
Médio relevantes	7
Relevantes	17
Após leitura rápida dos relevantes	
Não puderam ser acessados	-4
Sem relevância significativa	11
Se mostraram relevantes	2
Adicionado via *snowballing*	+1

Table: Classificações dos artigos e suas quantidades

Conteúdos

UFABC

Classificação ODC

- O primeiro artigo analisado Lopes et al. (2020) consiste na criação de um modelo de *Machine Learning* que classifique defeitos através da classificação ODC
- O artigo introduz a classificação ODC (*Orthogonal Defect Classification*) como um *framework* muito popular para classificação que considera vários atributos
- Estes atributos estão divididos entre relatório aberto e fechado.
  - Atributos de relatório aberto são aqueles referentes às informações disponíveis no momento da ocorrência do erro
  - Já os atributos de relatório fechado estão relacionados com a correção aplicada aos defeitos

UFABC

Classificação ODC

- Dentro dos atributos de relatório fechado, temos o atributo de tipo, que será o foco deste trabalho
- Ainda neste mesmo artigo, uma base de dados para treinamento foi classificada manualmente, para treinamento do modelo de ML
- Após a classificação, os três tipos mais comuns de defeitos foram:
  - Algoritmo ou método (A/M): 829 (59,47%)
  - Função, classe ou objeto (F/C/O): 221 (15,86%)
  - Verificação (C): 146 (10,47%)
- Já no artigo de Thung et al. (2012), que também trata da criação de modelo de classificação, de 500 defeitos:
  - 286 (57,2%) foram classificadas como defeitos do tipo de controle e fluxo
  - 214 (42,8%) foram classificadas como defeitos de dados

UFABC

Conclusões

- A classificação ODC é um ótimo meio de classificação de defeitos do ponto de vista de desenvolvimento de software
- Principalmente, se utilizarmos os tipos de defeitos como parâmetro para classificação
- Utilizando esta mesma taxonomia para classificar defeitos em outras aplicações, podemos validar se as ocorrências mais frequentes de defeitos encontradas em outras bases fazem sentido
- Com isso, atuamos nos itens mais comuns buscando outras abordagens que eliminem a possibilidade destes erros ocorrerem

UFABC

Conteúdos

UFABC

Introdução

- Com uma classificação consistente do ponto de vista de desenvolvimento de software, classificamos ocorrências de defeitos de uma aplicação real
- Esta aplicação é mobile, desenvolvida em Java e Kotlin e executada no Android 7.1.1 (API 25) em um hardware proprietário da empresa que mantém a aplicação
- Ela conta com um serviço de monitoramento, o Crashlytics, que reporta falhas que podem ser ou não capturadas, assim como podem ou não ser fatais para a execução da aplicação
- Analisar este tipo de dado facilita a aplicação da classificação ODC e definição do tipo de erro

UFABC

Considerações

- Os erros de comunicação não foram considerados na classificação por serem esperados, dada a natureza de uma aplicação móvel
- O monitoramento de erros captura apenas o tipo de exceção lançada nestes casos. Logo, os classificamos através da ODC baseado no tipo de exceção capturada
- Com isso, comparamos estas ocorrências com o que foi observado nos artigos pesquisados anteriormente

UFABC

Dados classificados

Tipo	**ODC**	**Classes**	**%**
Variável não inicializada	A/I	25	55,77%
Acesso inválido a variável		3	
Erro na inicialização		1	
Erro de fluxo	A/M	8	15,38%
Erro em aplicação externa	I/OOM	2	5,77%
Erro de banco de dados		1	

Table: Três tipos de erros mais frequentes em classes

Conclusões

- Considerando as ocorrências em classes, a maior quantidade de ocorrências está concentrada nos tipos:
  - Atribuição e inicialização (A/I) - 55,77%
  - Algoritmo ou método (A/M) - 15,38%
  - Interface ou mensageria (I/OOM) - 5,77%
- Estes tipos somam aproximadamente 70% das ocorrências e todos podem ser tratados utilizando outras abordagens corretas, do ponto de vista de desenvolvimento
- Isso reforça a hipótese deste trabalho, portanto, analisamos a causa e formas de mitigação destes erros utilizando as linguagens de programação utilizadas nesta aplicação

UFABC

Conteúdos

UFABC

Introdução

- Analisamos algumas abordagens dos erros mais frequentes encontrados em nossa aplicação
- A ideia é demonstrar que, utilizando alguns recursos da própria linguagem, não é complexo construir códigos que eliminam ou mitigam a possibilidade de ocorrências destes defeitos
- Vamos fazer uma comparação entre as versões utilizadas na aplicação, que são Java 8 e Kotlin 1.5.10
- Existem muitas semelhanças entre as duas linguagens, inclusive, Kotlin possui interoperabilidade com Java, por isso a comparação entre elas
- Vamos exemplificar um dos tipos de erros mais comuns que fora encontrado

Erros de fluxo

- Erros de fluxo podem ser causados por erros na implementação das regras de negócio ou no tratamento indevido de falhas que podem desviar o fluxo conforme o esperado para a aplicação
- Fatores externos podem interferir no fluxo planejado para aplicação, como uma falha física no hardware ou falha em algum serviço externo a aplicação, entre outros
- Exploramos a seguir algumas formas de tratar estes eventos inesperados

UFABC

Exceções

- Tanto em Java quanto em Kotlin, exceções são eventos que ocorrem durante a execução de uma aplicação por conta da disrupção do fluxo normal ou esperado de suas instruções
- Porém, o controle do fluxo da aplicação, em caso de eventos não esperados não precisam ser feitos apenas utilizando a captura/declaração de exceções

UFABC

Resultados orientado a tipos

- Existe uma outra abordagem utilizando Orientação a Objetos e Orientação a Tipos que pode ser utilizada para garantir, em tempo de compilação, que todos os cenários serão cobertos
- Tome, para um exemplo, que existe uma interface de resultado, chamada *Result* e duas implementações dela, uma de sucesso, chamada *Success* e uma de falha, chamada *Fail*
- Note que é simples verificarmos se o resultado de uma operação é sucesso ou falha verificando de qual classe um resultado é uma instância

Resultados orientado a tipos

- Imagine que um novo tipo *NetworkFail* implementa *Result*, mas contendo outros dados sobre a falha
- Em Java, não existe um meio de garantir que todos os casos sejam cobertos
- Seu compilador não obriga o desenvolvedor a tratar este novo tipo de resultado

Resultados orientado a tipos + Pattern Matching

- Em Kotlin, utilizando um tipo específico de classes (ou interfaces, em versões mais novas) chamadas *sealed*, é possível que o compilador saiba quais são todas as implementações possíveis de uma classe (ou interface)
- Utilizando a expressão *when*, é possível implementar uma espécie de *pattern matching*, que nos obriga é a cobrir todas as implementações possíveis de um determinado valor
- Complementando o exemplo em Java, tome para um exemplo uma classe *sealed* chanda *Result* com as implementações que representam um resultado de sucesso e uma de erro
- Quando uma nova implementação, o compilador falha em todas as verificações da expressão *when*, pois, estas não serão exaustivas (isto é, não cobrem todas possibilidades)

Conclusões

- Podemos detectar em tempo de compilação alguns erros mais triviais e de grande risco, como os erros de variável não inicializada
- A possibilidade de não captura ou tratamento de eventos inesperados diminuem empregando alguns destes tipos de abordagens
- Além disso, o risco de durante o ciclo de vida aplicação de se propagar novos erros é reduzido
- Com isso, o custo de manutenção e incremento do software é reduzido

UFABC

Conteúdos

Contextualização

- *Linters*, como são comumente conhecidas estas ferramentas, fazem uma análise estática do código fonte da aplicação em busca de possíveis erros, adoção de má práticas de programação, problemas de formatação, entre outros
- Estas ferramentas estão integradas diretamente em grande parte das IDEs, ou, podem ser executadas como plugins do gerenciadores de *build* das aplicações
- Possui um conjunto de regras de análise de código para avaliar diferentes segmentos de vulnerabilidades, sendo altamente configuráveis e personalizáveis
- Vão além da detecção de erros, mantendo também a qualidade, estética e legibilidade do código em si

Implementação

- Para realizar a implementação de um *linter*, utilizamos um projeto inicialmente desenvolvido em Java, baseado no funcionamento da ferramenta Apache ZooKeeper
- Convertemos este projeto para Kotlin para possibilitar as melhorias
- Além disso, com a conversão, muitos dos padrões inseguros de Java se mantem e são detectados pela ferramenta
- O plugin do *linter detekt* foi implementado e a verificação foi executada no projeto como prova de conceito para este trabalho

UFABC

Relatório exportado

- Veja um trecho do relatório exportado em HTML após a primeira execução da ferramente em um dos módulos, como exemplo:

Findings

Total: 15

exceptions: 8

▶ **SwallowedException: 2** The caught exception is swallowed. The original exception could be lost.

▶ **TooGenericExceptionCaught: 5** The caught exception is too generic. Prefer catching specific exceptions to the case that is currently handled.

▶ **TooGenericExceptionThrown: 1** The thrown exception is too generic. Prefer throwing project specific exceptions to handle error cases.

potential-bugs: 2

▶ **UnsafeCallOnNullableType: 2** Unsafe calls on nullable types detected. These calls will throw a NullPointerException in case the nullable value is null.

style: 5

▶ **MagicNumber: 1** Report magic numbers. Magic number is a numeric literal that is not defined as a constant and hence it's unclear what the purpose of this number is. It's better to declare such numbers as constants and give them a proper name. By default, -1, 0, 1, and 2 are not considered to be magic numbers.

Injeção do *dispatcher* de corrotinas

- Abordaremos um exemplo de alerta apontado durante a análise de um dos módulos do projeto em código não conforme a regra:

```
1  class ControllerImpl(
2      override val port: Int,
3      debug: Boolean
4  ) : Controller {
5      private val coroutineScope = CoroutineScope(SupervisorJob() + Dispatchers.IO)
6      ...
7      override fun start() {
8          timestampJob = coroutineScope.launch { timestampRepository.run() }
9          dispatcherJob = coroutineScope.launch { dispatcher.run() }
10     }
11     ...
12 }
```

Figure: *ControllerImpl.kt* — Trecho não conforme com a regra de injeção de *dispatch* de corrotinas

Injeção do *dispatcher* de corrotinas

- Código conforme a prática recomendada

```
1  class ControllerImpl(
2      override val port: Int,
3      debug: Boolean,
4      private val coroutineScope: CoroutineScope
5  ) : Controller {
6      ...
7      override fun start() {
8          timestampJob = coroutineScope.launch { timestampRepository.run() }
9          dispatcherJob = coroutineScope.launch { dispatcher.run() }
10     }
11     ...
12 }
```

Figure: *ControllerImpl.kt* — Trecho conforme com a regra de injeção de *dispatcher* d
corrotinas

Conclusão

- A execução do processo de *lint* dectectou vulnerabilidades não apenas no funcionamento de aplicação, mas na observação de práticas de risco ou desencorajadas, má formatação ou estilização de código, entre dificultade de criação e manutenção de teste e outros pontos
- Note que este processo também é facilmente empregado no ciclo de vida do desenvolvimento da aplicação, integrando-se no processo de testes, *build* e deploy de aplicações que utilizam o Gradle

Considerações finais

- Com uma revisão estruturada na literatura, encontramos um tipo de classificação consolidada e popular de defeitos em softwares, a classificação ODC
- Aplicando esta classificação numa base conhecida, confirmamos que os erros mais comuns tem relação com o processo de desenvolvimento
- Analisando estes erros, podemos diminuir ou eliminar as ocorrências destes erros a curto e médio prazo
- Utilizando outras ferramentas como os *linters*, podemos detectar e tratar vulnerabilidades de diversos tipos, através das regras nele empregadas
- O ganho da implementação deste tipo de ferramenta se estende ao longo de todo o tempo de vida da aplicação, uma vez que o processo de *lint* pode ser executado a cada passo do processo de integração e publicação contínua

UFABC

Próximos passos

- Ampliar a pesquisa para busca em repositórios públicos de quais são defeitos mais comuns nestas aplicações e quais os tipos de correções aplicadas
  - Com isso, consolidamos a hipótese do trabalho sobre a relação dos defeitos mais comuns com o processo de desenvolvimento
- Podemos propor outras abordagens de controle de defeitos, inclusive utilizando outros paradigmas de programação, explorando opções como linguagens de programação puramente funcionais ou desenvolvimento orientado a tipos

UFABC

Fim

- Dúvidas ou sugestões?

Referências I

Candido, M. G. (2022). Testes automatizados em aplicações client-side em javascript: uma análise compreensiva.

Liu, C., Zhao, Y., Yang, Y., Lu, H., Zhou, Y., and Xu, B. (2015). An ast-based approach to classifying defects. In *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*, pages 14–21.

Lopes, F., Agnelo, J., Teixeira, C. A., Laranjeiro, N., and Bernardino, J. (2020). Automating orthogonal defect classification using machine learning algorithms. *Future Generation Computer Systems*, 102:932–947.

Thung, F., Lo, D., and Jiang, L. (2012). Automatic defect categorization. In *2012 19th Working Conference on Reverse Engineering*, pages 205–214.

Conteúdos

UFABC

String de busca

- Iniciamos a construção da *string* de busca pela área e subárea de pesquisa.
- Tomamos a área principal de pesquisa como *software*
- Partimos então para as subáreas, sendo elas
  - *development* (ou desenvolvimento) e *engineering* (engenharia)
  - buscamos por *defeito* ou *defeitos*. Como eles são popularmente referidos como *bugs*, essa será nossa segunda subárea
  - como nosso objetivo inicial é classificar as ocorrências encontradas, teremos *classification* como terceira e última subárea

UFABC

String de busca

- Logo, teremos a *string* composta por:

Área de pesquisa	Subáreas		
	1	2	3
software	development engineering	bug*	classification

Table: Divisão das áreas de pesquisa

String de busca

- Para compor a *string* final, relacionamos as colunas numa lógica *E* e as linhas numa lógica *OU*. Deste modo, temos a seguinte *string*:

```
TITLE-ABS-KEY ( software ) AND ( TITLE-ABS-KEY ( development ) OR TITLE-ABS-KEY ( engineering ) )
      AND TITLE-ABS-KEY ( bug* ) AND TITLE-ABS-KEY ( classification )
```

Figure: *String* de busca nos acervos científicos

Etapas da revisão

- Criação de uma *string* de busca para pesquisa em ao menos um acervo digital
- Classificação dos resultados por relevância para este trabalho[1] de acordo com o título
- Leitura dos resumos dos artigos de maior relevância para reclassificação
- Leitura rápida dos artigos de maior relevância para reclassificação
- Compreensão mais detalhada dos artigos mais relevantes

[1] Note que esta classificação pode ser refinada entre qualquer um dos passos, conforme necessário

UFABC

Classificação inicial

- Tomando como base a classificação da Candido (2022), criamos definimos a relevância dos resultados para este estudo de acordo com os critérios listados abaixo:

Relevância		Critério
0	Não relevante	Não deveria ser considerado na pré-análise
1	Pouco relevante	Faz algum tipo de análise da causa raiz ou predição de defeitos
2	Relevante	Faz alguma classificação entre tipos de defeitos
3	Muito relevante	Classifica e ordena os defeitos encontrados com mais frequência

Table: Relevâncias utilizadas para classificação dos artigos

Classificação dos resultados obtidos

- Para separar estes casos que dificultam a classificação, teremos uma nova relevância de valor 2, sendo de *média relevância*, veja:

Relevância		Critério
0	Não relevante	Não deveria ser considerado na pré-análise
1	Pouco relevante	Faz algum tipo de análise da causa raiz ou predição de defeitos
2	Média relevância	Classifica defeitos de qualquer forma que não seja do ponto de vista de desenvolvimento de software
3	Relevante	Faz alguma classificação entre tipos de defeitos do ponto de vista de desenvolvimento
4	Muito relevante	Classifica e ordena os defeitos encontrados do ponto de vista de desenvolvimento ranqueando-os pela frequência

Table: Relevâncias utilizadas para classificação dos artigos

Conteúdos

UFABC

Classificação ODC

- O primeiro artigo analisado Lopes et al. (2020) consiste na criação de um modelo de *Machine Learning* que classifique defeitos através da classificação ODC
- O artigo introduz a classificação ODC (*Orthogonal Defect Classification*) como um *framework* muito popular para classificação que considera vários atributos
- Estes atributos estão divididos entre relatório aberto e fechado.
- Atributos de relatório aberto são aqueles referentes às informações disponíveis no momento da ocorrência do erro. Sendo elas:
  - Atividade: atividade sendo executada no momento em que o defeito ocorre
  - Gatilho: causador do defeito
  - Impacto: impacto causado ao usuário quando o defeito ocorreu

Classificação ODC

- Já os atributos de relatório fechado estão relacionados com a correção aplicada aos defeitos, que são:
  - Alvo: objeto que foi alvo da correção
  - Tipo: tipo de alteração realizada no código
  - Qualificador: característica do código anterior a alteração
  - Idade: intervalo de tempo entre o surgimento do defeito e a correção
  - Origem: origem do defeito

UFABC

Classificação ODC

- Já os atributos de relatório fechado estão relacionados com a correção aplicada aos defeitos, que são:
  - Alvo: objeto que foi alvo da correção
  - Tipo: tipo de alteração realizada no código
  - Qualificador: característica do código anterior a alteração
  - Idade: intervalo de tempo entre o surgimento do defeito e a correção
  - Origem: origem do defeito

Classificação ODC

- Note que o tipo é o atributo mais importante para estudo, pois, é o impacto do ponto de vista de desenvolvimento que analisaremos neste trabalho
- As possibilidades deste atributos são sete tipos, agrupados em duas categorias:
  - Defeitos de fluxo e controle de dados
    - Atribuição ou inicialização (A/I)
    - Verificação (C)
    - Algoritmo ou método (A/M)
    - Temporização ou serialização (T/S)
  - Defeitos estruturais
    - Função, classe ou objeto (F/C/O)
    - Interface ou mensagens O-O (I/OOM)
    - Relacionamento (R)

UFABC

Artigos remanescentes

- O artigo de Thung et al. (2012) também propõe uma classificação automática de defeitos utilizando a classificação ODC e modelos de *Machine Learning*
- Porém, os tipos de defeitos utilizados são apenas as duas categorias principais de tipos, que são defeitos de controle de controle e fluxo e de dados
- A base utilizada para classificação também é menor, contando com apenas 500 *issues* utilizadas para treinamento do modelo
- Destas, 286 (57,2%) foram classificadas como defeitos de controle e fluxo
- Enquanto as outras 214 (42,8%) foram classificadas como defeitos de dados

Artigos remanescentes

- O estudo de Liu et al. (2015) propõe uma classificação por tipos diferentes do ODC
- A classificação proposta é entre os tipos:
  - Dados
  - Computacionais
  - Interface
  - Controle/lógica
- Esta classificação não atende tão bem ao propósito deste artigo, por isso, não foi levado em consideração durante os próximos passos

UFABC

Conteúdos

Dados coletados

Tipo	**Ocorrências**		**Usuários**
	Classes	Únicas	
Erro de comunicação com o servidor	8	1.120.357	137,126
Variável não inicializada	25	16,104	6,914
Acesso inválido a vetor	4	15,898	5,363
Erro de parseamento	2	11,700	2,347
Erro de fluxo	8	1009	589
Erro de tipo recebido	1	499	434
Erro em aplicação externa	2	363	282

Table: Exceções lançadas classificadas por tipo e suas quantidades de ocorrências

Dados coletados

Tipo	**Ocorrências**		**Usuários**
	Classes	Únicas	
Falha na leitura de arquivo	1	135	135
Erro na inicialização	1	122	107
Estouro de memória	3	117	110
Acesso inválido a variável	3	86	73
Erro de permissão	1	35	4
Erro de banco de dados	1	25	16

Table: Exceções lançadas classificadas por tipo e suas quantidades de ocorrências

Dados classificados

Tipo	**ODC**	**Ocorrências**	**%**
Variável não inicializada	A/I	16,104	35,39%
Acesso inválido a variável		86	
Erro na inicialização		122	
Acesso inválido a vetor	C	15,898	34,57%
Erro de permissão		35	
Erro de parseamento	T/S	11,700	25,38%

Table: Três tipos de erros mais frequentes em ocorrências

Considerações

- Note que erros como de comunicação, *parseamento*, fluxo e em aplicações externas podem ser tratados utilizando abordagens de controle de exceções ou eventos não esperados
- Portanto, estes tipos de erros não tem relação direta com a pilha de tecnologias selecionada
- O monitoramento de erros captura apenas o tipo de exceção lançada nestes casos. Logo, classificá-los através da ODC baseado no tipo de exceção capturada
- Com isso, comparamos estas ocorrências com o que foi observado nos artigos pesquisados anteriormente

UFABC

Conteúdos

Variável não inicializada - Java

- Este defeito consiste no acesso ao valor de uma variável antes da sua correta inicialização

```
1  class NullPointerExceptionSimulation {
2      public static void main(String[] args) {
3          Result test = null;
4
5          System.out.print(test.data().toString())
6      }
7  }
```

Figure: Exemplo de código que lança uma exceção do tipo *NullPointerException*

Variável não inicializada - Kotlin

- Observe no código abaixo que, em Kotlin, não podemos declarar uma variável diretamente sem que seu valor seja atribuído
- Então, este não é um código funcional

```kotlin
class UninitializedVariableError {
    private val nonChangeableValue: Int
    private var changeableVariable: Int = null

    fun execute(){
        print("Non changeable value is $nonChangeableValue")
        print("Changeable value is $changeableVariable")
    }
}
```

Figure: Exemplo de código incorreto de variável não inicializada

Variável não inicializada - Kotlin

- Para que o código seja funcional, baste que inicializemos as duas variáveis

```
1  class UninitializedVariableError {
2      private val nonChangeableValue: Int = 0
3      private var changeableVariable: Int = 1
4
5      fun execute(){
6          print("Non changeable value is $nonChangeableValue")
7          print("Changeable value is $changeableVariable")
8      }
9  }
```

Figure: Exemplo de código funcional após inicializar variáveis

Variável não inicializada - Kotlin

- Porém, este comportamento pode ser sobreposto
- Existe um tipo de variável que pode ser inicializada de forma tardia, utilizando o operador *lateinit*
- Este operador é aplicado apenas a variáveis mutáveis e permite que o valor seja atribuido em um momento após a declaração
- O compilador espera que a variável estará inicializada quando seu valor for acessado

```
1  class UninitializedVariableError {
2      private lateinit var lateInitVariable: List<Int>
3
4      fun execute(){
5          lateInitVariable = listOf(0, 1, 2)
6
7          print("Non nullable value is ${lateInitVariable.first()}")
8      }
9  }
```

Figure: Exemplo de variável do tipo *lateinit* utilizada de forma válida

Variável não inicializada - Kotlin

- Até o momento, todos os tipos de variáveis analisados requeriam um valor não-nulo
- Ainda assim, esta regra também pode ser sobreposta
- Utilizando o operador *?*, quando precedido de uma classe, ele indica que este tipo pode ser anulável, isto é, pode adotar um valor nulo

UFABC

Variável não inicializada - Kotlin

- De todo modo, mesmo numa variável, o compilador tem maneiras de tentar garantir que um valor inválido não seja acessado
- Não é permitido que o acesso a uma variável anulável seja feita diretamente
- Precisamos fazer uma chamada segura (*safe call*) a estes valores, utilizando o operador *?* após a variável acessada

```
1  class UninitializedVariableError {
2      private val nullableValue: Int? = null
3      private var nonNullableVariable: Int = 0
4
5      fun execute(){
6          print("Non nullable value is ${nullableValue?.toFloat()}")
7          print("Nullable value is $nonNullableVariable")
8      }
9  }
```

Figure: Exemplo de chamada de uma variável nula com *safe call*

Variável não inicializada - Kotlin

- Entretanto, este é mais um exemplo de regra que pode ser ignorada
- Existe em Kotlin o operador *!!*, que indica ao compilador que o desenvolvedor garante que a variável ou valor que precede este operador não será nulo no momento do seu acesso

```kotlin
class UninitializedVariableError {
    private var nullableVariable: Int? = null
    private var nonNullableVariable: Int = 0

    fun execute(){
        nullableVariable = Random(1000).nextInt()

        print("Non nullable value is ${nullableVariable!!.toFloat()}")
        print("Nullable value is $nonNullableVariable")
    }
}
```

Figure: Exemplo de chamada válida de uma variável anulável com *null assert*

Variável não inicializada - Kotlin

- O Kotlin tem o propósito de eliminar erros causados pela leitura inválida de um valor nulo
- Portanto, o uso dos recursos de sobreposição demonstrados anteriormente é recomendado apenas quando se faz extremamente necessário
- Como por exemplo, durante a utilização de uma dependência externa que não garante o retorno de um valor válido

UFABC

Acesso inválido a vetor - Java

- Em Java, teremos um erro de acesso ao item de um vetor se tentarmos acessar um índice de valor maior ou igual ao tamanho do vetor

```java
public class ArrayOutOfBoundsError {
    public static void execute(){
        int[] list = new int[] { 0, 1, 2 };

        System.console().printf("%d", list[4]);
    }
}
```

Figure: Exemplo de um acesso inválido ao vetor em Java

Acesso inválido a vetor - Kotlin

- Quando utilizamos Kotlin, estamos sujeitos ao mesmo erro, caso acessemos o valor da mesma forma que em Java
- Utilizando o operador *get* (*[indice]*) em um índice inválido

```
1  class ArrayOutOfBoundsError {
2      fun execute(){
3          val array = arrayOf(0, 1, 2)
4
5          System.console().printf(array[4].toString())
6      }
7  }
```

Figure: Exemplo de um acesso inválido ao vetor em Kotlin

- Porém, o Kotlin também dispõe algumas funções que tratam os casos onde um índice inválido é acessado

Acesso inválido a vetor - Kotlin

- Com o uso da função *getOrNull*, caso um indíce inválido seja acessado, o valor nulo é retornado
- Porém, como a melhor prática é evitar a utilização de valores nulos, existe uma outra opção

Acesso inválido a vetor - Kotlin

- A função *getOrElse* recebe uma função *lambda* de que tem um valor inteiro como entrada, referente ao índice acessado, e deve retornar o tipo dos itens da lista acessada

```kotlin
class Fibonacci {
    fun preCalculated(i: Int): Int =
        intArrayOf(1, 1, 2, 3, 5, 8, 13, 21).getOrElse(i) { calculate(it) }

    companion object {
        fun calculate(i: Int): Int =
            if (i <= 2) 1 else calculate(i - 1) + calculate(i - 2)
    }
}
```

Figure: Exemplo de um utilização da função *getOrElse* em Kotlin

Exceções

- Tanto em Java quanto em Kotlin, exceções são eventos que ocorrem durante a execução de uma aplicação por conta da disrupção do fluxo normal ou esperado de suas instruções
- Existem três tipos de exceções
  - *Checked exceptions*: eventos excepcionais das quais uma aplicação bem escrita deveria ser capaz de se recuperar
  - Erros *errors*: eventos *externos* a qual a aplicação pode não se antecipar ou se recuperar
  - *Runtime Exceptions*: eventos *internos* a qual a aplicação pode não se antecipar ou se recuperar

UFABC

Exceções

- Em Java, existe um princípio chamado *Catch or Specify Requirement*
- Este princípio exige um dos dois seguintes pontos

Exceções

- Deve haver um bloco *try* que capture este tipo de exceção em torno da chamada

```
1  public class CatchingExample {
2      public static void catching() {
3          try {
4              throwsException();
5          } catch(Exception e) {
6              // Do something with the exception
7          }
8      }
9  }
```

Figure: Exemplo da captura de exceção Java lançada por um método

Exceções

- Deve declarar que agora, o método onde a chamada é realizada pode lançar este tipo de exceção

```
1  public class SpecifyingExample {
2      public static void specified() throws IOException {
3          if (true) throw new IOException();
4      }
5  }
```

Figure: Exemplo de declaração de exceção em Java

Resultados orientado a tipos

- Existe uma outra abordagem utilizando Orientação a Objetos e Orientação a Tipos que pode ser utilizada para garantir, em tempo de compilação, que todos os cenários serão cobertos
- Tome como exemplo a interface *Result*

```
1  public interface Result {
2      boolean isSuccess();
3  }
```

Figure: Exemplo de interface de resultados em Java

Resultados orientado a tipos

- Observe também, duas implementações diferentes desta interface
- Uma que representa um resultado de sucesso

```
1  public class Success<T> implements Result {
2      private final T data;
3
4      public Success(T data) {
5          this.data = data;
6      }
7
8      @Override
9      public boolean isSuccess() {
10         return true;
11     }
12
13     public T getData() {
14         return data;
15     }
16 }
```

Figure: Exemplo de implementação de uma interface de resultado que representa o resultado de sucesso

Resultados orientado a tipos

- Outra que representa um resultado de falha

```
1  public class Fail implements Result {
2      private final Error error;
3
4      public Fail(Error error) {
5          this.error = error;
6      }
7
8      @Override
9      public boolean isSuccess() {
10         return false;
11     }
12
13     public Error getError() {
14         return error;
15     }
16 }
```

Figure: Exemplo de declaração de exceção em Java

Resultados orientado a tipos

```java
public class Main {
    public static void main(String[] args) {
        Result result = SomeComplexOperation.execute();

        if(result instanceof Success<Data>) {
            // Do something with the success result
        } else if(result instanceof Fail) {
            // Do something with the error
        } else {
            throw new IllegalStateException("Result type not implemented")
        }
    }
}
```

Figure: Exemplo de como tratar implementações de *Result* em Java

Resultados orientado a tipos

```java
public class Main {
    public static void main(String[] args) {
        Result result = SomeComplexOperation.execute();

        if(result instanceof Success<Data>) {
            // Do something with the success result
        } else if(result instanceof NetworkFail) {
            // Do something with the error
        } else if(result instanceof Fail) {
            // Do something with the error
        } else {
            throw new IllegalStateException("Result type not implemented")
        }
    }
}
```

Figure: Exemplo de como tratar implementações de *Result* em Java

Resultados orientado a tipos + Pattern Matching

- Tome exemplo a *sealed class* chamada *Result*

```
1  sealed class Result {
2      data class Success<T>(val data: T) : Result()
3      data class Fail(val exception: Exception) : Result()
4  }
```

Figure: Exemplo de implementação de classe *sealed* em Kotlin como resultado

Resultados orientado a tipos + Pattern Matching

- Note como, ao salvarmos o resultado de uma operação que retorna uma instância de resultado, utilizando uma expressão when, o compilador nos obriga é verificar todos os cenários

```
1  class Program {
2      fun main(args: String[]) {
3          val result = SomeComplexOperation.execute()
4
5          when(result) {
6              is Result.Success<*> -> {
7                  /* Do something with the data inside the success object */
8              }
9              is Result.Error -> {
10                 /* Do something with the error */
11             }
12         }
13     }
14 }
```

Figure: Exemplo de utilização de classes de resultado em Kotlin

Resultados orientado a tipos + Pattern Matching

- Num exemplo como demonstrado em Java, onde um novo tipo de erro é adicionado, o código não compila até que este novo tipo de resultado seja implementado nas expressões *when*

```
1  class Program {
2      fun main(args: String[]) {
3          val result = SomeComplexOperation.execute()
4
5          when(result) {
6              is Result.Success<*> -> {
7                  /* Do something with the data inside the success object */
8              }
9              is Result.Error -> {
10                 /* Do something with the error */
11             }
12             is Result.NetworkFail -> {
13                 /* We have to handle the new type here */
14             }
15         }
16     }
17 }
```

Figure: Exemplo de adição de um novo tipo de falha como um resultado

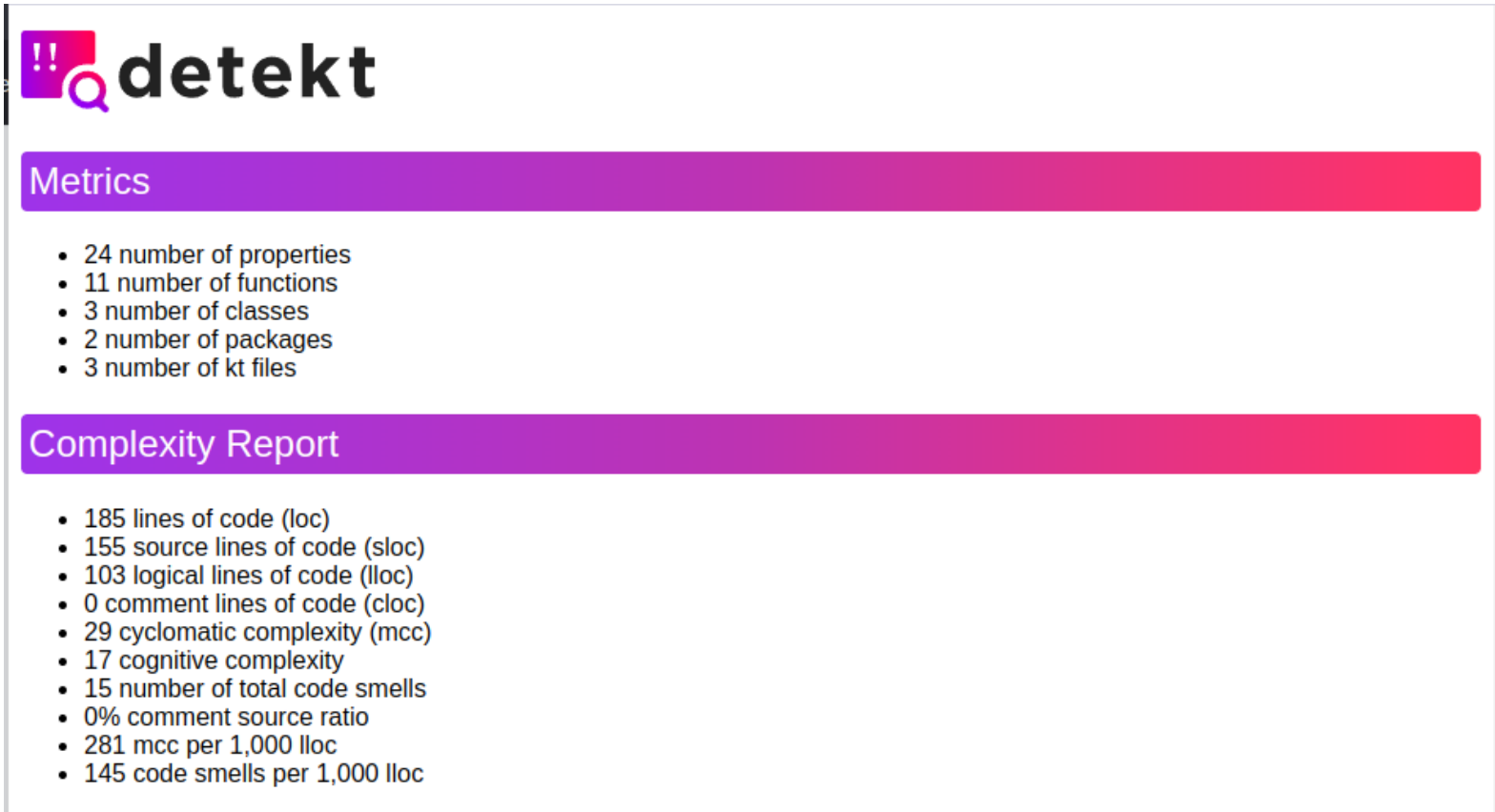
Conteúdos

Implementação

- A versão do Kotlin será a mais recente, até o momento da conversão, a 1.9.20 e a versão do Gradle a 8.4
- Implementamos a versão 1.23.3 do plugin do linter *detekt*, como prova de conceito
- Com o plugin configurado, executamos a primeira vez para coletar um relatório de todas os alertas de código fora das regras configuradas
- É importante destacar que a execução da análise passa a ser dependência de comandos de verificação, testes e *build* do pacote executável da aplicação
- O ganho disto é a implementação desta análise em fluxos de CI/CD, por exemplo

Relatório exportado

- Veja um trecho do relatório exportado em HTML, como exemplo:



detekt

Metrics

- 24 number of properties
- 11 number of functions
- 3 number of classes
- 2 number of packages
- 3 number of kt files

Complexity Report

- 185 lines of code (loc)
- 155 source lines of code (sloc)
- 103 logical lines of code (lloc)
- 0 comment lines of code (cloc)
- 29 cyclomatic complexity (mcc)
- 17 cognitive complexity
- 15 number of total code smells
- 0% comment source ratio
- 281 mcc per 1,000 lloc
- 145 code smells per 1,000 lloc

UFABC

Relatório exportado

Findings

Total: 15

exceptions: 8

- **SwallowedException: 2** The caught exception is swallowed. The original exception could be lost.
- **TooGenericExceptionCaught: 5** The caught exception is too generic. Prefer catching specific exceptions to the case that is currently handled.
- **TooGenericExceptionThrown: 1** The thrown exception is too generic. Prefer throwing project specific exceptions to handle error cases.

potential-bugs: 2

- **UnsafeCallOnNullableType: 2** Unsafe calls on nullable types detected. These calls will throw a NullPointerException in case the nullable value is null.

style: 5

- **MagicNumber: 1** Report magic numbers. Magic number is a numeric literal that is not defined as a constant and hence it's unclear what the purpose of this number is. It's better to declare such numbers as constants and give them a proper name. By default, -1, 0, 1, and 2 are not considered to be magic numbers.
- **MaxLineLength: 2** Line detected, which is longer than the defined maximum line length in the code style.
- **VarCouldBeVal: 1** Var declaration could be val.
- **WildcardImport: 1** Wildcard imports should be replaced with imports using fully qualified class names. Wildcard imports can lead to naming conflicts. A library update can introduce naming clashes with your classes which results in compilation errors.

generated with detekt version 1.23.3 on 2023-12-09 16:56:46 UTC.

Condicional complexa

- Código não conforme com a regra

```
1   class KeyValueRepository {
2       ...
3       fun find(key: String, timestamp: Long?): Entry? {
4           val result = data.getOrDefault(key, null)
5
6           if (timestamp != null && timestamp > 0 && result != null && result.timestamp < timestamp)
7               throw OutdatedEntryException(key, result.timestamp)
8
9           return result
10      }
11      ...
12  }
```

Figure: *KeyValueRepository.kt* - Trecho não conforme com a regra de condicional complexa

Condicional complexa

- Código corrigido

```
1  class KeyValueRepository {
2      ...
3      fun find(key: String, timestamp: Long?): Entry? {
4          return data.getOrDefault(key, null)?.also { entry ->
5              timestamp?.let {
6                  if(entry.timestamp < it) error("Outdated entry found!")
7              } ?: entry
8          }
9      }
10     ...
11 }
```

Figure: *KeyValueRepository.kt* - Exemplo de não violação da regra de condicional complexa

Captura de exceção genérica

- Código não conforme com a regra

```
1   class ControllerImpl(
2       override val port: Int,
3       debug: Boolean,
4       private val coroutineScope: CoroutineScope
5   ) : Controller {
6       ...
7       override fun put(request: PutRequest): PutResponse {
8           return try {
9               ...
10          } catch (e: Throwable) {
11              ...
12          }
13      }
14      ...
15  }
```

Figure: *ControllerImpl.kt* - Exemplo de captura de exceção muito genérica na classe *ControllerImpl*

Captura de exceção genérica

- Código corrigido

```
1   class ControllerImpl(
2       override val port: Int,
3       debug: Boolean,
4       private val coroutineScope: CoroutineScope
5   ) : Controller {
6       ...
7       override fun put(request: PutRequest): PutResponse {
8           return try {
9               ...
10          } catch (e: IllegalStateException) {
11              ...
12          }
13      }
14      ...
15  }
```

Figure: *ControllerImpl.kt* - Exemplo de captura de exceção específica na classe *ControllerImpl*

Declaração de classe e nome de arquivo incompatíveis

- A classe abaixo estava contida no arquivo *Result.kt*

```
1  enum class OperationResult {
2      OK,
3      NOT_FOUND,
4      ERROR,
5      TRY_AGAIN_ON_OTHER_SERVER;
6  }
```

Figure: *Result.kt* - Exemplo de declaração de arquivo incompatível com a classe declarada

- Isso gera uma imcompatibilidade entre a classe declarada e o nome de seu arquivo
- Após renomear o arquivo para *OperationResult.kt* o alerta não é mais disparada

Chamada inegura a tipo anulável

- Código não conforme com a regra

```
1  class ClientImpl(
2      private val port: Int,
3      private val serverHost: String,
4      private val serverPorts: List<Int>,
5      debug: Boolean
6  ) : Client {
7      ...
8      override fun get(key: String) {
9          try {
10             val serverPort = serverPorts.getAnyOrNull()!!
11             ...
12         } catch (e: IOException) {
13             log.e("Failed to process GET request", e)
14         }
15     }
16
17     private fun <T> List<T>.getAnyOrNull(): T? = when {
18         isEmpty() -> null
19         else -> get(kotlin.random.Random.nextInt(0, size))
20     }
21     ...
22 }
```

Figure: *ClientImpl.kt* - Exemplo de chamada insegura num objeto anulável

Chamada inegura a tipo anulável

- Código corrigido

```
1  class ClientImpl(
2      private val port: Int,
3      private val serverHost: String,
4      private val serverPorts: List<Int>,
5      debug: Boolean
6  ) : Client {
7      ...
8      override fun get(key: String) {
9          try {
10             val serverPort = serverPorts.getAnyOrNull()
11             ...
12         } catch (e: IOException) {
13             log.e("Failed to process GET request", e)
14         } catch (e: IllegalStateException) {
15             log.e("No server port found to send requests", e)
16         }
17     }
18
19     private fun <T> List<T>.getAnyOrNull(): T = get(Random.nextInt(lastIndex))
20     ...
21 }
```

Figure: *ClientImpl.kt* - Segundo exemplo delegando o acesso ao elemento a função de extensão de *List*

Utilize funções *check* ou *error*

- Código não conforme com a regra

```
1  class ClientImpl(
2      private val port: Int,
3      private val serverHost: String,
4      private val serverPorts: List<Int>,
5      debug: Boolean
6  ) : Client {
7      ...
8      override fun get(key: String) {
9          try {
10             val serverPort = serverPorts.getAnyOrNull()
11                 ?: throw IllegalStateException("No server port found to send request")
12             ...
13         } catch (e: IOException) {
14             log.e("Failed to process GET request", e)
15         } catch (e: IllegalStateException) {
16             log.e("No server port found to send requests", e)
17         }
18     }
19
20     private fun <T> List<T>.getAnyOrNull(): T? = when {
21         isEmpty() -> null
22         else -> get(Random.nextInt(0, lastIndex))
23     }
24     ...
25 }
```

Figure: *ClientImpl.kt* - Exemplo não conforme de lançamento "manual" de exceção pa

Utilize funções *check* ou *error*

- Código corrigido

```
1  class ClientImpl(
2      private val port: Int,
3      private val serverHost: String,
4      private val serverPorts: List<Int>,
5      debug: Boolean
6  ) : Client {
7      ...
8      override fun get(key: String) {
9          try {
10             val serverPort = serverPorts.getAnyOrNull()
11
12             checkNotNull(serverPort) { "No server port found to send request" }
13             ...
14         } catch (e: IOException) {
15             log.e("Failed to process GET request", e)
16         } catch (e: IllegalStateException) {
17             log.e("No server port found to send requests", e)
18         }
19     }
20
21     private fun <T> List<T>.getAnyOrNull(): T? = when {
22         isEmpty() -> null
23         else -> get(Random.nextInt(0, lastIndex))
24     }
25     ...
26 }
```

Figure: *ClientImpl.kt* - Exemplo de tratamento de pré-condição utilizando *checkNotN*

Utilização de *imports* globais

- Código não conforme com a regra

```
1  import kotlinx.coroutines.*
```

Figure: *Dispatcher.kt* - Exemplo *import* global no contexto do arquivo

Utilização de *imports* globais

- Código corrigido

```
1  import kotlinx.coroutines.withContext
2  import kotlinx.coroutines.Dispatchers
```

Figure: *Dispatcher.kt* - Exemplo de *imports* totalmente qualificados

Classe abstrata descenessária

- Código não conforme com a regra

```kotlin
abstract class Response(
    val result: OperationResult,
    val message: String?
)
```

Figure: *Response.kt* - Exemplo de classe abstrata que não possui membros concreto

Classe abstrata descenessária

- Código corrigido

```
1  interface Response {
2      val result: OperationResult
3      val message: String?
4  }
```

Figure: *Response.kt* - Exemplo de interface equivalente a classe abstrata

Exceção não utilizada

- Código não conforme com a regra

```
1  class Worker(private val client: Client, debug: Boolean) {
2      ...
3      fun run() {
4          while (running) {
5              try {
6                  ...
7              } catch (e: IllegalArgumentException) {
8                  println("Opcao invalida! Tente novamente ou pressione Ctrl+D para finalizar.")
9              }
10         }
11     }
12 }
```

Figure: *Worker.kt* - Exemplo de exceção ignorada dentro do bloco *catch*

Exceção não utilizada

- Código corrigido

```
1  class Worker(private val client: Client, debug: Boolean) {
2      ...
3      fun run() {
4          while (running) {
5              try {
6                  ...
7              } catch (e: IllegalArgumentException) {
8                  log.e("Invalid option entered!", e)
9                  println("Opcao invalida! Tente novamente ou pressione Ctrl+D para finalizar.")
10             }
11         }
12     }
13 }
```

Figure: *Worker.kt* - Exemplo de utilização de exceção capturada no bloco *catch*

Números “mágicos“

- Código não conforme com a regra

```
1  interface Controller {
2      ...
3      companion object {
4          private const val TAG = "ControllerMain"
5
6          @JvmStatic
7          fun main(args: Array<String>) {
8              try {
9                  ...
10                 val port = readIntWithDefault("Digite o valor da porta do servidor", 10097)
11                 ...
12             } catch (e: Throwable) {
13                 handleException(TAG, "Failed start Controller...", e)
14             }
15         }
16     }
17 }
```

Figure: *Controller.kt* - Exemplo de utilização de *magic number* (valores *hard-coded*

Números “mágicos“

- Código corrigido

```
1  interface Controller {
2      ...
3      companion object {
4          private const val TAG = "ControllerMain"
5          private const val DEFAULT_PORT = 10097
6
7          @JvmStatic
8          fun main(args: Array<String>) {
9              try {
10                 ...
11                 val port = readIntWithDefault("Digite o valor da porta do servidor", DEFAULT_PORT)
12                 ...
13             } catch (e: Throwable) {
14                 handleException(TAG, "Failed start Controller...", e)
15             }
16         }
17     }
18 }
```

Figure: *Controller.kt* - Exemplo de alteração de um *magic number* para uma variável constante mais descritiva

Variáveis mutáveis podem ser imutáveis

- Código não conforme com a regra

```
1  class Dispatcher(private val server: Server) {
2      ...
3      private var running = true
4
5      suspend fun run() = withContext(Dispatchers.IO) {
6          try {
7              while (running) {
8                  val socket = serverSocket.accept()
9                  ...
10             }
11         } catch (e: SocketException) {
12             log.e("Socket closed!", e)
13         } finally {
14             serverSocket.close()
15         }
16     }
17
18     companion object {
19         private const val TAG = "DispatcherThread"
20     }
21 }
```

Figure: *Dispatcher.kt* - Exemplo de utilização de uma variável mutável que não tem s
valor alterado

Variáveis mutáveis podem ser imutáveis

- Código corrigido

```
1  class Dispatcher(private val server: Server) {
2      ...
3      private val running = true
4
5      suspend fun run() = withContext(Dispatchers.IO) {
6          try {
7              while (running) {
8                  val socket = serverSocket.accept()
9                  ...
10             }
11         } catch (e: SocketException) {
12             log.e("Socket closed!", e)
13         } finally {
14             serverSocket.close()
15         }
16     }
17
18     companion object {
19         private const val TAG = "DispatcherThread"
20     }
21 }
```

Figure: *Dispatcher.kt* - Exemplo da troca de variável mutável por valor apenas de leitu