TÓPICO 09 - TRATAMENTO DE ERROS

Clean Code - Professor Ramon Venson - SATC 2025

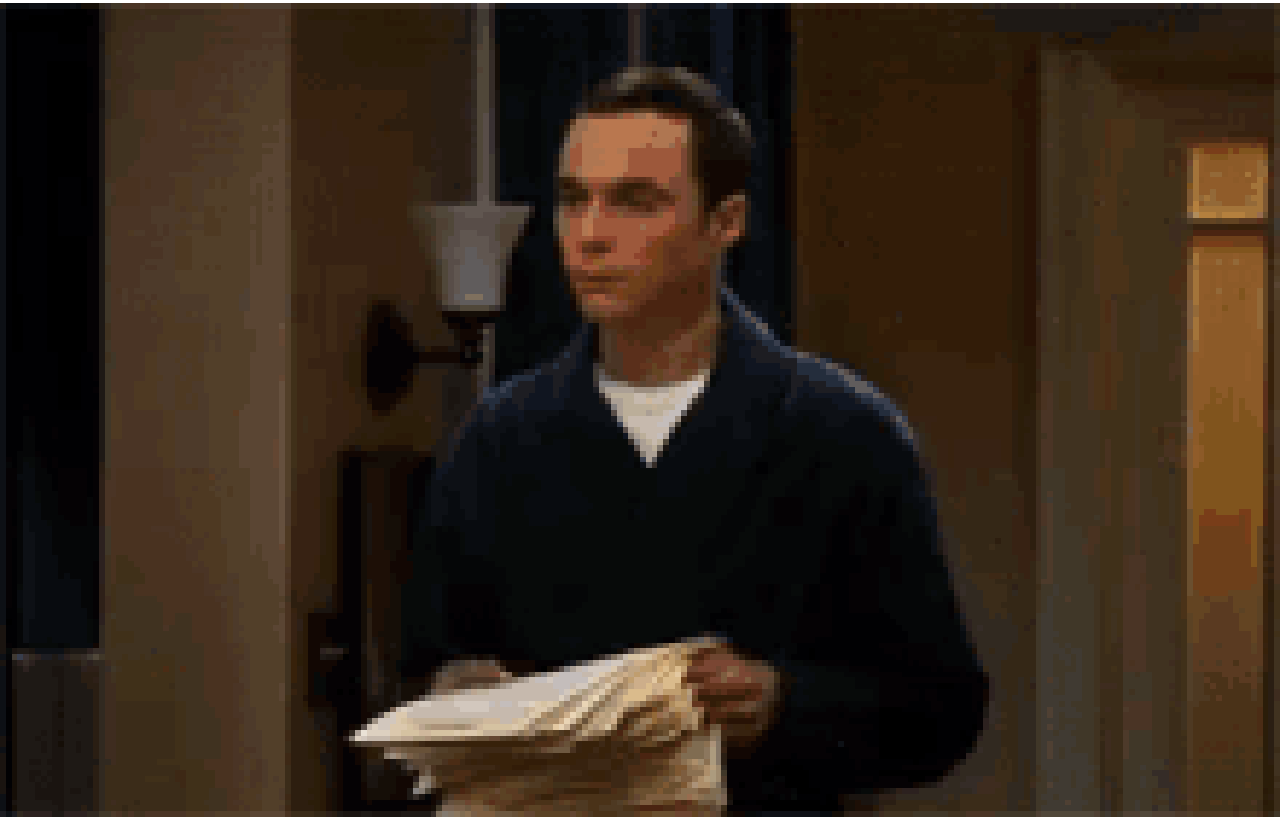
O que são exceções

Exceções são um mecanismo de tratamento de erros que permite a recuperação de uma situação de erro.

Exemplo de Tratamento de Exceção

A maioria das linguagens utiliza uma sintaxe parecida para tratar exceções.

```
try {
    // Código que pode lançar uma exceção
}
catch (Exception e) {
    // Código para lidar com a exceção
}
finally {
    // Código que sempre será executado ao final, mesmo em caso de exceção
}
```



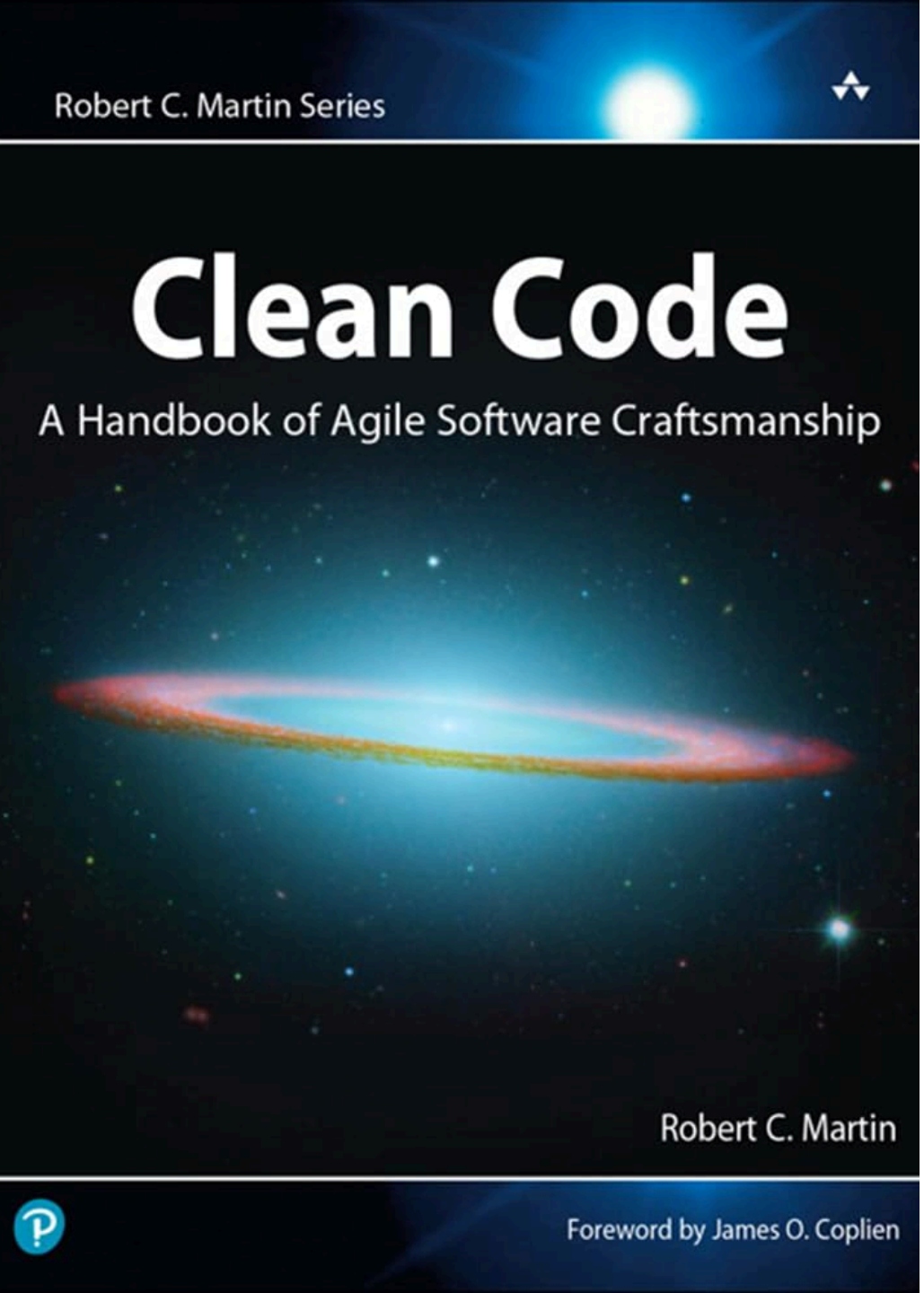
Lançamento de Exceções

Exceções podem ser lançadas quando uma condição excepcional ocorre. Isso geralmente é definido pelo uso da palavra chave `throw` ou `raise`.

Let It Crash

Algumas linguagens, como Elixir e Erlang, não possuem exceções.

Isso é uma **filosofia** de programação que diz que o programa **deve falhar rápido e de forma explícita**, permitindo a rápida recuperação.



Sumário do Livro

- Use exceções ao invés de códigos
- Crie primeiro o `try-catch-finally`
- Use exceções não verificadas
- Forneça contexto
- Defina o fluxo normal
- Não retorne `null`
- Não passe `null`

Use exceções ao invés de códigos

Retornar código de erros dificulta a manutenção.

Além disso, exceções permitem que o programa **responda mais rapidamente a erros**, sem necessidade de aguardar o retorno de cada função.

Error Code	Description
100	Loader available waiting time out
101	Unloader available waiting time out
102	Product input waiting time out
103	Press unit action waiting time out
104	Product out waiting time out
105	Bypass error
106	Door is opened
107	Motor low speed waiting time out
108	Product is unlocated
109	Stopper action waiting time out
110	Conveyer action waiting time out
111	Repeat cycle end
112	Error Flag set
113	Two boards error
114	Unknown error
115	Next station error
116	No Hardware Detected
117	Product Position Error
118	Barcode Error

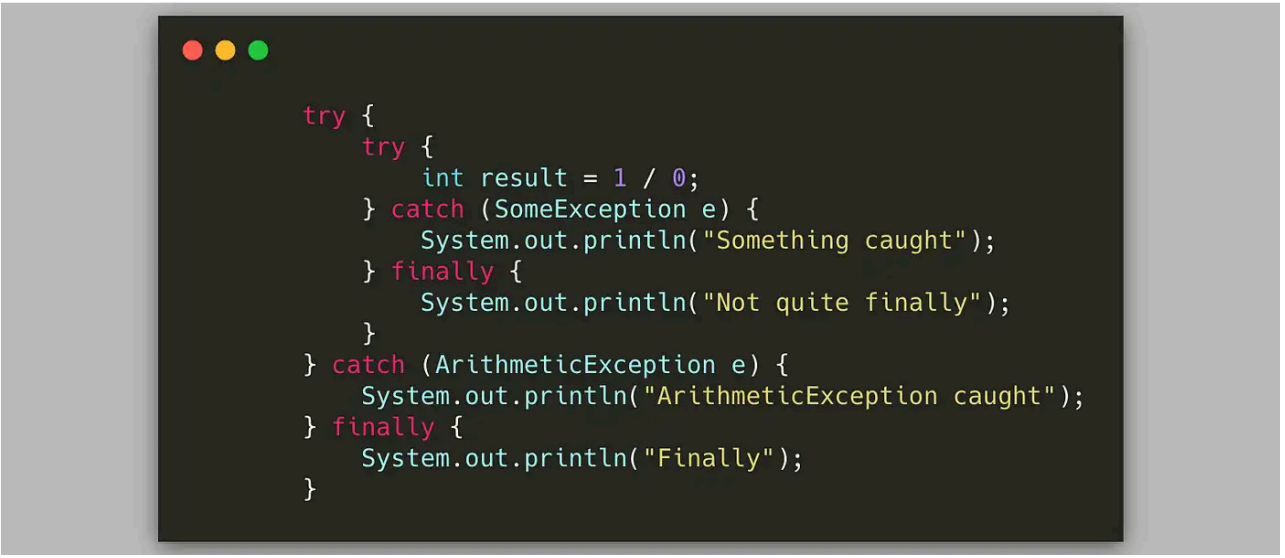
Ao invés de:

```
int dividir(int a, int b, int &resultado) {
    if (b == 0) {
        return -1; // Código de erro para divisão por zero
    }
    resultado = a / b;
    return 0; // Sucesso
}

int main() {
    int resultado;
    if (dividir(10, 0, resultado) == -1) {
        std::cerr << "Erro: divisão por zero!" << std::endl;
    } else {
        std::cout << "Resultado: " << resultado << std::endl;
    }
    return 0;
}
```

Use:

```
int dividir(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Divisão por zero não permitida.");
    }
    return a / b;
}

int main() {
    try {
        int resultado = dividir(10, 0); // Retorna o resultado ao invés do erro
        std::cout << "Resultado: " << resultado << std::endl;
    } catch (const std::runtime_error &e) {
        std::cerr << "Erro: " << e.what() << std::endl;
    }
    return 0;
}
```

Crie primeiro o `try-catch-finally`



```java
try {
    try {
        int result = 1 / 0;
    } catch (SomeException e) {
        System.out.println("Something caught");
    } finally {
        System.out.println("Not quite finally");
    }
} catch (ArithmeticException e) {
    System.out.println("ArithmeticException caught");
} finally {
    System.out.println("Finally");
}
```

Ao criar um método que pode lançar exceção, criar a estrutura `try-catch-finally` primeiro ajuda a definir o objetivo do método e qual será o estado final do programa.

O `finally` é usado para garantir que o programa sempre saia do método de forma consistente, mesmo que uma exceção seja lançada.

Use exceções não verificadas

Exceções verificadas são aquelas que devem ser tratadas explicitamente pelo programador, como `IOException` e `SQLException`.

Exceções não verificadas são aquelas que não precisam ser tratadas explicitamente pelo programador, como `NullPointerException` e `ArrayIndexOutOfBoundsException`.

Se lançarmos uma exceção verificada a partir de um método e o `catch` estiver três níveis acima, será necessário declarar a exceção na assinatura de cada um dos métodos entre você e o `catch`.

Java é uma das linguagens que implementa exceções verificadas.

```java
public class MissaoRebelde {
    public static void main(String[] args) {
        try {
            baseRebelde(); // A missão começa na base rebelde
        } catch (IOException e) {
            System.out.println("⚠️ Base Rebelde: Perdemos contato com Luke! " + e.getMessage());
        }
    }

    static void baseRebelde() throws IOException {
        System.out.println("📡 Base Rebelde: Enviando ordens para a Aliança...");
        aliancaRebelde();
    }

    static void aliancaRebelde() throws IOException {
        System.out.println("🚀 Aliança Rebelde: Chamando Luke Skywalker para atacar!");
        lukeSkywalker();
    }

    static void lukeSkywalker() throws IOException {
        System.out.println("🛸 Luke Skywalker: Mirando no reator da Estrela da Morte...");
        throw new IOException("🔥 Erro crítico! O alvo foi perdido!"); // Lançando exceção
    }
}
```

Forneça contexto

Cada exceção lançada deve forncer informações suficientes para entender o que acontece e onde aconteceu.

A mensagem deve incluir informações como:

- O que estava acontecendo no momento da exceção;
- Onde a exceção ocorreu;
- Qual foi a causa da exceção;

Defina o fluxo normal

Uma boa prática é definir o fluxo normal do programa antes de tratar exceções.

Isso garante que a lógica de negócios permanecerá clara e fácil de entender, enquanto casos especiais serão tratados por objetos de exceção.

Ao invés de:

```
public void processOrder(Order order) {
    if (order == null) {
        System.out.println("Error: Order is null");
        return;
    }
    if (!order.isValid()) {
        System.out.println("Error: Invalid order");
        return;
    }
    try {
        orderProcessor.process(order);
    } catch (Exception e) {
        System.out.println("Processing failed: " + e.getMessage());
    }
}
```

Use:

```
public void processOrder(Order order) {
    validateOrder(order);
    try {
        orderProcessor.process(order);
    } catch (ProcessingException e) {
        logError(e);
        throw new OrderProcessingException("O processamento da ordem de serviço falhou", e);
    }
}

private void validateOrder(Order order) {
    if (order == null || !order.isValid()) {
        throw new InvalidOrderException("A ordem de serviço é invalida");
    }
}
```

Não retorne `null`

Acessar um objeto nulo cria automaticamente um comportamento inesperado.

Evite utilizar `null` como valor de retorno, pois isso faz com que o cliente tenha que verificar se o objeto é nulo antes de acessar seus métodos.

Não passe `null`

Repassar `null` como parâmetro tambem é um problema.

Realize o tratamento de exceção quando o parâmetro é nulo, ou utilize a pattern `Optional`.

Outras Dicas

Padrão Objeto Nulo

O padrão Objeto Nulo (*Null Object Pattern*) é uma técnica de projeto que permite tratar objetos nulos de forma segura e consistente.

Considere como exemplo a seguinte classe:

```
class Cliente {
    private String nome;
    private String email;
    // ... getters e setters
}
```

```
public class Main {
    public static void main(String[] args) {
        // Não existe cliente com id 500, retorna NULL
        Cliente cliente = ClienteRepository.getClienteById(500);

        // Imprimindo os dados do cliente
        System.out.println("Nome: " + cliente.getNome()); // Erro: NullPointerException
        System.out.println("Email: " + cliente.getEmail()); // Erro: NullPointerException
    }
}
```

Dessa forma podemos implementar:

```
class NullCliente extends Cliente {
    public NullCliente() {
        super("", "");
    }
}

public class Main {
    public static void main(String[] args) {
        // Não existe cliente com id 500
        Cliente cliente = ClienteRepository.getClienteById(500);

        // Imprimindo os dados do cliente
        System.out.println("Nome: " + cliente.getNome()); // Retorna ""
        System.out.println("Email: " + cliente.getEmail()); // Retorna ""
    }
}
```

Optional

O `Optional` é uma classe que representa um valor opcional. Ele pode conter um valor ou estar vazio, mas nunca será nulo, de forma que possa causar exceções.

```java
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Usuário: ");
        String usuario = scanner.nextLine();
        System.out.print("Senha: ");
        String senha = scanner.nextLine();

        Optional<String> resultado = Autenticacao.autenticarUsuario(usuario, senha);

        if (resultado.isPresent()) { // Verifica se há um valor
            System.out.println("✅ Login bem-sucedido! Bem-vindo, " + resultado.get() + "!");
        } else {
            System.out.println("❌ Erro: Usuário ou senha incorretos.");
        }

        scanner.close();
    }
}
```

Separe a lógica de negócios

O tratamento de erros deve acontecer fora de métodos que manipulam a lógica de negócios.

Alguns padrões podem auxiliar na separação de responsabilidades.

```
// Classe de serviço responsável pela lógica de negócios
class PedidoService {
    public void processarPedido(int pedidoId) {
        if (pedidoId <= 0) {
            throw new PedidoException("ID do pedido inválido.");
        }
        System.out.println("✅ Pedido #" + pedidoId + " processado com sucesso!");
    }
}
```

```
// Controlador responsável por chamar o serviço e tratar erros
class PedidoController {
    public void realizarPedido(int pedidoId) {
        try {
            pedidoService.processarPedido(pedidoId);
        } catch (PedidoException e) {
            System.err.println("❌ Erro ao processar o pedido: " + e.getMessage());
        }
    }
}
```

Capture só o necessário

Evite capturar exceções que não são necessárias, usando o `catch(Exception e)`.

Especialize as chamadas para garantir um contexto mais significativo para o tratamento.

Exemplo:

```
public class LeitorDeArquivo {
    public static void lerArquivo(String caminho) {
        try (BufferedReader reader = new BufferedReader(new FileReader(caminho))) {
            reader.lines().forEach(System.out::println);
        } catch (FileNotFoundException e) {
            System.err.println("❌ Arquivo não encontrado: " + caminho);
        } catch (IOException e) {
            System.err.println("❌ Erro ao ler o arquivo: " + caminho);
        }
    }

    public static void main(String[] args) {
        lerArquivo("arquivo_inexistente.txt");
        lerArquivo("arquivo_corrompido.txt");
    }
}
```

Use finally para limpeza

Sempre feche os recursos abertos, como arquivos e conexões de banco de dados, quando ocorrerem exceções.

Isso evita vazamentos de recursos e garante que o estado do programa seja consistente.

```
public void listarPartidas(String casa, String visitante) {
    Connection conexao = DriverManager.getConnection();
    try {
        Partidas partidas = Partidas.carregarPartidas(casa, visitante);
        partidas.forEach(System.out::println);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    } finally {
        conexao.close(); // Fecha a conexão
    }
}
```

Evite falhas silenciosas

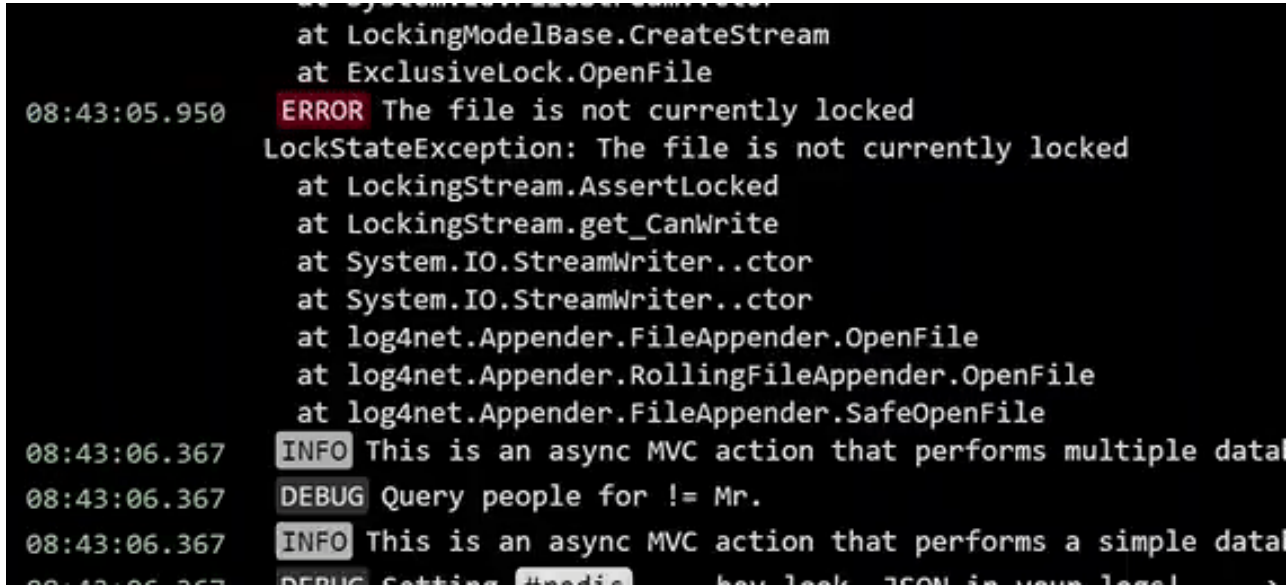
Não capture exceções sem que elas sejam tratadas ou registradas.

Exemplo:

```
public void listarPartidas(String timeCasa, String timeVisitante) {
    Connection conexao = DriverManager.getConnection();
    try {
        Partidas partidas = Partidas.carregarPartidas(timeCasa, timeVisitante);
        partidas.forEach(System.out::println);
    } catch (SQLException e) {
        // Não faz nada
    } finally {
        conexao.close();
    }
}
```

Logging

Use *logs* estruturados ao invés de printar mensagens de erro no console.



```
public void listarPartidas(String casa, String visitante) {
    Connection conexao = DriverManager.getConnection();
    try {
        Partidas partidas = Partidas.carregarPartidas(casa, visitante);
        partidas.forEach(System.out::println);
    } catch (SQLException e) {
        log.error("Erro ao listar partidas: {}", e.getMessage());
    } finally {
        conexao.close(); // Fecha a conexão
    }
}
```

Não use exceções para fluxo

Exceções não devem ser confundidas com `if-else`. O tratamento de exceção é para situações excepcionais, não para fluxo de controle.

Além disso, exceções são mais lentas que estruturas de controle.

```
public void getNumeroCamisa(String nome) {
    Jogador jogador = jogadores.containsKey(nome)
    int numero = jogador.getNumeroCamisa();
    if (numero < 12) {
        throw new TitularException();
    } else {
        throw new ReservaException();
    }
}
```

Mensagens de erro significativas

Uma mensagem de erro significativa é aquela que fornece informações suficientes para entender o que aconteceu e onde aconteceu.

Evite expor informações sensíveis (ex.: credenciais) ou detalhes de implementação.

Use classes personalizadas

Classes que são específicas do domínio auxiliam na compreensão do código e na categorização de erros.

`Exception`, `RuntimeException` e `Error` são classes genéricas e não devem ser usadas para criar exceções personalizadas.

Falhe alto e rápido

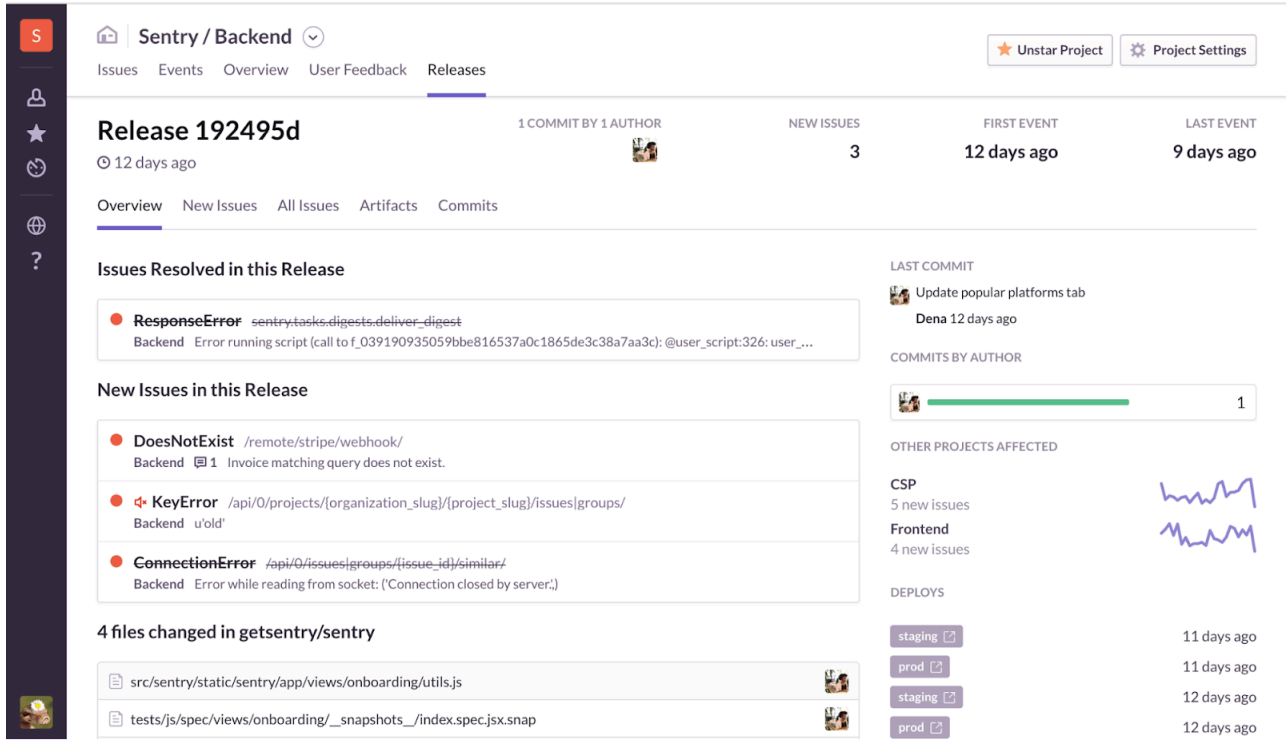
Detecte e corrija erros o mais cedo possível.

Evite que o erro se propague para outros pontos do código.

Ferramentas de Monitoramento

Ferramentas de monitoramento e registro de erros podem ajudar a identificar e resolver problemas:

- Sentry
- Loggly
- New Relic
- Datadog
- Rollbar



Material de Apoio

- Toptal
- omar.saibaa
- Daniel Wisky