Validador com Expressões Regulares

Davison Logan dos Santos Cardoso[1]

[1]ICEN – Universidade Federal do Pará (UFPA)
R. Augusto Corrêa, 01 - Guamá, Belém - PA, 66075-110

[2]Faculdade de Computação –Universidade Federal do Pará

davison.cardoso@icen.ufpa.br

Resumo. *Este trabalho tem como objetivo desenvolver e implementar um validador utilizando expressões regulares para a verificação de entradas de dados em diferentes formatos, como nome, e-mail, senha, CPF e telefone. A metodologia consiste em criar máscaras de validação específicas para cada tipo de dado, garantindo que as entradas atendam aos padrões esperados. Para testar a eficácia do validador, foi criada uma série de exemplos de entradas válidas e inválidas, e os resultados foram analisados. A aplicação do validador contribui para garantir que os dados sejam consistentes e estejam no formato adequado, fundamental em processos como cadastro de usuários em sistemas. O código foi desenvolvido em Python, utilizando expressões regulares (regex) para realizar as validações de forma eficiente e robusta. Como resultado, o validador foi capaz de identificar corretamente entradas válidas e inválidas, evidenciando a importância das expressões regulares no controle de qualidade de dados.*

1. Introdução

A validação de dados é uma etapa crucial em muitos sistemas, especialmente em formulários de cadastro, onde informações como nome, e-mail, CPF e telefone são frequentemente coletadas. A precisão desses dados é essencial para evitar problemas futuros, como falhas de comunicação ou erros no processamento de informações. Uma das ferramentas mais poderosas para essa validação é o uso de expressões regulares (regex), que permitem criar padrões para identificar se os dados estão no formato correto. Expressões regulares são sequências de caracteres que formam padrões de busca, amplamente utilizadas para validar entradas em sistemas, realizar buscas e substituições em textos, entre outros.

O objetivo deste trabalho é construir um validador que utilize expressões regulares para garantir que os dados inseridos estejam corretos, sem erros de formatação. A criação de máscaras de validação específicas para diferentes tipos de dados é essencial para garantir que informações como e-mails, senhas, CPFs e números de telefone sejam inseridas corretamente. Além disso, a implementação de um validador robusto contribui diretamente para a melhoria da confiabilidade do sistema, evitando o processamento de dados inválidos e assegurando a integridade das informações armazenadas.

Essas técnicas são especialmente importantes em sistemas que lidam com grandes volumes de dados, onde a precisão e a conformidade das informações são fundamentais para o funcionamento adequado e seguro dos processos.

2. Materiais e Métodos

2.1. Escolha da Linguagem

A linguagem escolhida para o desenvolvimento deste trabalho foi **Python**, devido à sua simplicidade, versatilidade e vasta documentação. Python é amplamente utilizado em diversas áreas de desenvolvimento, desde scripts simples até sistemas mais complexos. A facilidade de uso e a leitura intuitiva do código tornam-no uma escolha excelente para quem busca rapidez no desenvolvimento e clareza no entendimento do código. Além disso, a linguagem oferece suporte nativo para expressões regulares através da biblioteca `re`, o que permite realizar validações de dados de maneira eficiente e sem a necessidade de bibliotecas externas complexas.

2.2. Bibliotecas

Para a implementação deste validador, foi utilizada a biblioteca `re`, que é a biblioteca padrão do Python para trabalhar com expressões regulares. A biblioteca `re` fornece funções poderosas para buscar, manipular e validar padrões de texto, o que foi fundamental para a construção das máscaras de validação neste trabalho. Além disso, foi utilizada a biblioteca `tabulate` para formatar e apresentar os resultados dos testes de maneira estruturada e legível, facilitando a visualização do desempenho do validador durante os testes experimentais.

2.3. Desenvolvimento do Código

O desenvolvimento do código foi baseado na criação de máscaras de validação utilizando expressões regulares para cada tipo de dado. As expressões regulares foram cuidadosamente desenhadas para corresponder exatamente ao formato esperado de cada tipo de dado, como nome, e-mail, CPF, senha e telefone. O processo de construção das expressões regulares envolveu:

- **Máscara para Nome**: A expressão regular foi construída para garantir que o nome seguisse o padrão de uma primeira letra maiúscula, seguida de letras minúsculas, e permitisse um sobrenome com as mesmas regras. A máscara também possibilita a presença de um segundo nome, separando-o por um espaço.

```python
@staticmethod  1 usage
def mascara_nome(nome: str) -> bool:
    match = re.search( pattern: r'[A-Z][a-z]+( [A-Z][a-z]+)? [A-Z][a-z]+', nome)
    return Mascaras.valida_regex(match, nome)
```

Figure 1. Implementação para a máscara para nome

- **Máscara para E-mail**: A expressão foi criada para validar e-mails com a estrutura básica `exemplo@dominio.com.br`. Permitindo o domínio `.com` e `.br`, mas restringindo o formato a letras minúsculas.

```python
@staticmethod  1 usage
def mascara_email(email: str) -> bool:
    match = re.search( pattern: r'[a-z]+@[a-z]+(.com)?.br', email)
    return Mascaras.valida_regex(match, email)
```

Figure 2. Implementação para a máscara para email

- **Máscara para Senha**: A senha deve ter exatamente 8 caracteres, contendo pelo menos uma letra maiúscula e um número. A expressão regular foi projetada para verificar essas condições de forma rigorosa.

```python
@staticmethod  1 usage
def mascara_senha(senha: str) -> bool:
    if len(senha) != 8:
        return False
    match = re.search( pattern: r'(?=.*[A-Z])(?=.*[0-9])[a-z|A-Z|0-9]*', senha)
    return Mascaras.valida_regex(match, senha)
```

Figure 3. Implementação para a máscara para senha

- **Máscara para CPF**: O formato do CPF foi validado para garantir que estivesse no formato `xxx.xxx.xxx-xx`, onde `x` são números.

```python
@staticmethod  1 usage
def mascara_cpf(cpf: str) -> bool:
    match = re.search( pattern: r'[0-9]{3}.[0-9]{3}.[0-9]{3}-[0-9]{2}', cpf)
    return Mascaras.valida_regex(match, cpf)
```

Figure 4. Implementação para a máscara para CPF

- **Máscara para Telefone**: A expressão foi criada para validar números de telefone com diferentes formatos: com ou sem parênteses e traços, além de permitir um número de celular com DDD e um número de 9 dígitos.

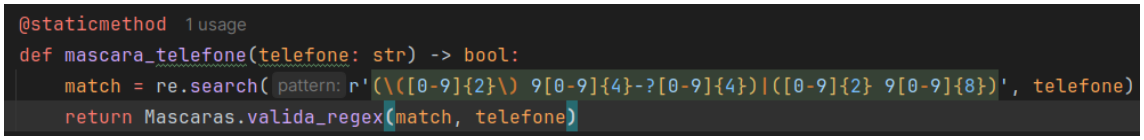

Figure 5. Implementação para a máscara para telefone

Essas expressões foram integradas à classe `Mascaras`, onde cada método corresponde a um tipo de dado a ser validado. O validador, por sua vez, recebe a entrada de dados, escolhe a máscara apropriada e utiliza as expressões regulares para determinar se a entrada está correta ou não.

2.4. Ambiente de Desenvolvimento

O código foi desenvolvido em um **notebook Avell**, equipado com um processador **Intel i7** e uma **placa de vídeo GeForce**, utilizando o sistema operacional **Pop!_OS** (baseado em Linux). A IDE escolhida foi o **PyCharm** da JetBrains, uma plataforma robusta e eficiente para o desenvolvimento em Python. O ambiente de desenvolvimento ofereceu todos os recursos necessários para a codificação, depuração e execução dos testes de forma eficiente, com recursos como auto-completação, verificação de erros em tempo real e integração com ambientes virtuais para gerenciamento de dependências.

3. Testes Experimentais

Os testes realizados neste trabalho têm o objetivo de garantir que as máscaras de validação, implementadas por meio das expressões regulares, funcionem corretamente para diferentes tipos de dados, como nome, e-mail, senha, CPF e telefone. A seguir, apresento uma descrição detalhada dos casos de teste utilizados, incluindo as entradas, saídas esperadas e os resultados obtidos.

3.1. Casos de Teste

Para cada tipo de dado, foram definidos casos de teste divididos em dois grupos: **aceitas** (entradas válidas) e **rejeitadas** (entradas inválidas). Cada entrada foi verificada contra a máscara correspondente, e o resultado foi comparado ao esperado.

3.1.1. 1. Validação de Nome

- **Entradas aceitas**: ”Alan Turing”, ”Noam Chomsky”, ”Ada Lovelace”
- **Entradas rejeitadas**: ”1Alan”, ”Alan”, ”A1an”
- **Objetivo**: Validar se o nome tem o formato correto, com a primeira letra maiúscula e o resto minúsculo, além de garantir que o nome tenha pelo menos duas palavras (primeiro nome e sobrenome).

3.1.2. 2. Validação de E-mail

- **Entradas aceitas**: ”a@a.br”, ”divulga@ufpa.br”
- **Entradas rejeitadas**: ”@”, ”a@.br”, ”T@teste.br”
- **Objetivo**: Validar o formato de e-mails, garantindo que possuam o domínio ”.br” e uma estrutura válida de nome de usuário e domínio.

3.1.3. 3. Validação de Senha

- **Entradas aceitas**: ”518R2r5e”, ”F123456A”, ”1234567T”, ”ropsSoq0”
- **Entradas rejeitadas**: ”F1234567A”, ”abcdefgH”, ”1234567HI”
- **Objetivo**: Garantir que a senha tenha pelo menos 8 caracteres, incluindo pelo menos uma letra maiúscula e um número.

3.1.4. 4. Validação de CPF

- **Entradas aceitas**: ”123.456.789-09”, ”000.000.000-00”
- **Entradas rejeitadas**: ”123.456.789-0”, ”111.111.11-11”
- **Objetivo**: Verificar se o CPF segue o formato correto de 11 dígitos, com pontuação no formato `xxx.xxx.xxx-xx`.

3.1.5. 5. Validação de Telefone

- **Entradas aceitas**: ”(91) 99999-9999”, ”(91) 999999999”, ”91 999999999”
- **Entradas rejeitadas**: ”(91) 59999-9999”, ”99 99999-9999”, ”(94)95555-5555”
- **Objetivo**: Garantir que o telefone tenha o formato correto, aceitando ou não parênteses, traços e diferentes maneiras de escrever o número.

3.2. Resultados Esperados e Resultados Obtidos

Os resultados esperados foram que todas as entradas **aceitas** passassem nas validações, enquanto todas as entradas **rejeitadas** falhassem. O código foi executado e os resultados foram comparados com as saídas esperadas.

Categoria	Cadeia	Esperado	Resultado
nome	Alan Turing	Aceita	Aceita
nome	Noam Chomsky	Aceita	Aceita
nome	Ada Lovelace	Aceita	Aceita
nome	1Alan	Rejeitada	Rejeitada
nome	Alan	Rejeitada	Rejeitada
nome	A1an	Rejeitada	Rejeitada

Figure 6. Resultados do Programa de Validação

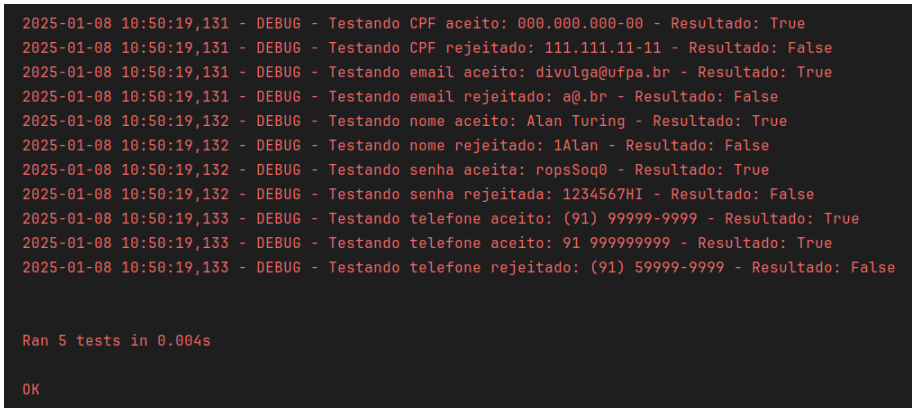

Figure 7. Resultados dos testes unitários

Esses testes ajudaram a assegurar que as expressões regulares estão funcionando corretamente, aceitando apenas entradas válidas e rejeitando as inválidas.

3.3. Como os Testes Garantem a Robustez do Código

Os testes cobrem uma ampla gama de cenários, incluindo entradas válidas e inválidas para todos os tipos de dados. Eles garantem que o código validador:

- **Acerte as entradas válidas**: Apenas as entradas que atendem aos critérios das expressões regulares são aceitas.
- **Rejeite entradas inválidas**: Qualquer entrada que não se encaixa no padrão esperado é corretamente rejeitada.

O código é capaz de lidar com diferentes tipos de dados e validar se eles atendem aos requisitos estabelecidos pelas expressões regulares de forma eficaz e eficiente.

4. Conclusão

4.1. Resultados Obtidos

Os resultados obtidos neste trabalho demonstraram que as expressões regulares implementadas para validar diferentes tipos de dados (como nome, e-mail, senha, CPF e telefone) são eficazes e funcionam conforme o esperado. O código validador foi capaz de aceitar as entradas válidas e rejeitar corretamente as entradas inválidas, garantindo que os dados seguissem os formatos estabelecidos para cada categoria. A precisão das validações foi confirmada pelos testes experimentais, que mostraram uma correspondência consistente entre os resultados esperados e os resultados obtidos.

4.2. Desafios Encontrados

Durante o desenvolvimento do código, alguns desafios foram encontrados, principalmente na criação de expressões regulares complexas. As expressões regulares precisam ser cuidadosas e específicas para cobrir todos os casos possíveis, sem falhar nas entradas válidas ou aceitar entradas inválidas. Além disso, algumas entradas imprevistas, como diferentes formatações de números de telefone ou variações de e-mails, exigiram ajustes finos nas expressões para garantir a precisão das validações. A complexidade das máscaras de validação, especialmente para tipos como CPF e senha, também foi um desafio, pois é necessário levar em consideração várias variações e regras.

5. Anexos do Código fonte

```python
import re

class Mascaras:
    @staticmethod
    def valida_regex(match: re.Match, cadeia: str) -> bool:
        if not match:
            return False

        if match.group(0) != cadeia:
            return False
        return True

    @staticmethod
    def mascara_nome(nome: str) -> bool:
        match = re.search(r'[A-Z][a-z]+( [A-Z][a-z]+)? [A-Z][a-z]+', nome)
        return Mascaras.valida_regex(match, nome)

    @staticmethod
    def mascara_email(email: str) -> bool:
        match = re.search(r'[a-z]+@[a-z]+(.com)?.br', email)
        return Mascaras.valida_regex(match, email)

    @staticmethod
    def mascara_senha(senha: str) -> bool:
        if len(senha) != 8:
            return False
        match = re.search(r'(?=.*[A-Z])(?=.*[0-9])[a-z|A-Z|0-9]*', senha)
        return Mascaras.valida_regex(match, senha)

    @staticmethod
    def mascara_cpf(cpf: str) -> bool:
        match = re.search(r'[0-9]{3}.[0-9]{3}.[0-9]{3}-[0-9]{2}', cpf)
        return Mascaras.valida_regex(match, cpf)

    @staticmethod
    def mascara_telefone(telefone: str) -> bool:
        match = re.search(r'(\([0-9]{2}\) 9[0-9]{4}-?[0-9]{4})|([0-9]{2} 9[0-9]{8})', telefone)
        return Mascaras.valida_regex(match, telefone)
```

Figure 8. Class que contém as Expressões regulares

```python
from src.validador.mascaras import Mascaras


class Validador:
    @staticmethod
    def validar_nome(nome):
        if Mascaras.mascara_nome(nome):
            return True, "Nome válido."
        return False, "Nome inválido. Deve conter nome e sobrenome com letras maiúsculas e minúsculas."

    @staticmethod
    def validar_email(email):
        if Mascaras.mascara_email(email):
            return True, "E-mail válido."
        return False, "E-mail inválido. Deve seguir o formato exemplo@exemplo.com ou
exemplo@exemplo.com.br."

    @staticmethod
    def validar_senha(senha):
        if Mascaras.mascara_senha(senha):
            return True, "Senha válida."
        return False, "Senha inválida. Deve conter pelo menos 1 letra maiúscula, 1 número e ter
exatamente 8 caracteres."

    @staticmethod
    def validar_cpf(cpf):
        if Mascaras.mascara_cpf(cpf):
            return True, "CPF válido."
        return False, "CPF inválido. Deve estar no formato xxx.xxx.xxx-xx."

    @staticmethod
    def validar_telefone(telefone):
        if Mascaras.mascara_telefone(telefone):
            return True, "Telefone válido."
        return False, "Telefone inválido. Deve seguir o formato (xx) 9xxxx-xxxx, (xx) 9xxxxxxxx ou xx
9xxxxxxxx."

    @staticmethod
    def validar(tipo, entrada):
        validadores = {
            "nome": Validador.validar_nome,
            "email": Validador.validar_email,
            "senha": Validador.validar_senha,
            "cpf": Validador.validar_cpf,
            "telefone": Validador.validar_telefone,
        }
        if tipo not in validadores:
            return False, f"Tipo de validação '{tipo}' não suportado."
        return validadores[tipo](entrada)
```

Figure 9. Class que utliza as regex para efetuar validações de dados

```python
from validador.validador import Validador
from tabulate import tabulate

def mostrar_tabela(exemplos):
    resultados = []

    for categoria, tipos in exemplos.items():
        for status, cadeias in tipos.items():
            for cadeia in cadeias:
                resultado, _ = Validador.validar(categoria, cadeia)
                resultados.append([
                    categoria,
                    cadeia,
                    "Aceita" if status == "aceitas" else "Rejeitada",
                    "Aceita" if resultado else "Rejeitada"
                ])

    print(tabulate(
        resultados,
        headers=["Categoria", "Cadeia", "Esperado", "Resultado"],
        tablefmt="grid"
    ))

def validar_dados():
    while True:
        print("\nTipos disponíveis: nome, email, senha, cpf, telefone")
        tipo = input("Escolha um tipo de dado para validar (ou digite 'voltar' para retornar ao menu):
").strip().lower()
        if tipo == "voltar":
            break

        if tipo not in ["nome", "email", "senha", "cpf", "telefone"]:
            print("Tipo inválido. Tente novamente.")
            continue

        dado = input(f"Digite o dado para validar como {tipo}: ").strip()
        resultado, mensagem = Validador.validar(tipo, dado)
        print(f"Resultado: {'Válido' if resultado else 'Inválido'} - {mensagem}")

def main():
    exemplos = {
        "nome": {
            "aceitas": ["Alan Turing", "Noam Chomsky", "Ada Lovelace"],
            "rejeitadas": ["1Alan", "Alan", "A1an"]
        },
        "email": {
            "aceitas": ["a@a.br", "divulga@ufpa.br"],
            "rejeitadas": ["@", "a@.br", "T@teste.br"]
        },
        "senha": {
            "aceitas": ["518R2r5e", "F123456A", "1234567T", "ropsSoq0"],
            "rejeitadas": ["F1234567A", "abcdefgH", "1234567HI"]
        },
        "cpf": {
            "aceitas": ["123.456.789-09", "000.000.000-00"],
            "rejeitadas": ["123.456.789-0", "111.111.11-11"]
        },
        "telefone": {
            "aceitas": ["(91) 99999-9999", "(91) 999999999", "91 999999999"],
            "rejeitadas": ["(91) 59999-9999", "99 99999-9999", "(94)95555-5555"]
        }
    }

    while True:
        print("\nMenu:")
        print("1. Mostrar tabela de exemplos")
        print("2. Validar dados manualmente")
        print("3. Sair")
        escolha = input("Escolha uma opção: ").strip()

        if escolha == "1":
            mostrar_tabela(exemplos)
        elif escolha == "2":
            validar_dados()
        elif escolha == "3":
            print("Encerrando o programa.")
            break
        else:
            print("Opção inválida. Tente novamente.")

if __name__ == "__main__":
    main()
```

Figure 10. main para executar as classes anteriores

```python
import unittest
import logging
from src.validador.mascaras import Mascaras

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger()

class TestMascaras(unittest.TestCase):

    exemplos = {
        "nome": {
            "aceitas": ["Alan Turing"],
            "rejeitadas": ["1Alan"]
        },
        "email": {
            "aceitas": ["divulga@ufpa.br"],
            "rejeitadas": ["a@.br"]
        },
        "senha": {
            "aceitas": ["ropsSoq0"],
            "rejeitadas": ["1234567HI"]
        },
        "cpf": {
            "aceitas": ["000.000.000-00"],
            "rejeitadas": ["111.111.11-11"]
        },
        "telefone": {
            "aceitas": ["(91) 99999-9999", "91 999999999"],
            "rejeitadas": ["(91) 59999-9999"]
        }
    }

    def test_mascara_nome(self):
        for item in self.exemplos['nome']['aceitas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_nome(item)
                logger.debug(f"Testando nome aceito: {item} - Resultado: {result}")
                self.assertTrue(result, f"Falha ao aceitar '{item}'")
        for item in self.exemplos['nome']['rejeitadas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_nome(item)
                logger.debug(f"Testando nome rejeitado: {item} - Resultado: {result}")
                self.assertFalse(result, f"Falha ao rejeitar '{item}'")

    def test_mascara_email(self):
        for item in self.exemplos['email']['aceitas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_email(item)
                logger.debug(f"Testando email aceito: {item} - Resultado: {result}")
                self.assertTrue(result, f"Falha ao aceitar '{item}'")
        for item in self.exemplos['email']['rejeitadas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_email(item)
                logger.debug(f"Testando email rejeitado: {item} - Resultado: {result}")
                self.assertFalse(result, f"Falha ao rejeitar '{item}'")

    def test_mascara_senha(self):
        for item in self.exemplos['senha']['aceitas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_senha(item)
                logger.debug(f"Testando senha aceita: {item} - Resultado: {result}")
                self.assertTrue(result, f"Falha ao aceitar '{item}'")
        for item in self.exemplos['senha']['rejeitadas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_senha(item)
                logger.debug(f"Testando senha rejeitada: {item} - Resultado: {result}")
                self.assertFalse(result, f"Falha ao rejeitar '{item}'")

    def test_mascara_cpf(self):
        for item in self.exemplos['cpf']['aceitas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_cpf(item)
                logger.debug(f"Testando CPF aceito: {item} - Resultado: {result}")
                self.assertTrue(result, f"Falha ao aceitar '{item}'")
        for item in self.exemplos['cpf']['rejeitadas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_cpf(item)
                logger.debug(f"Testando CPF rejeitado: {item} - Resultado: {result}")
                self.assertFalse(result, f"Falha ao rejeitar '{item}'")

    def test_mascara_telefone(self):
        for item in self.exemplos['telefone']['aceitas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_telefone(item)
                logger.debug(f"Testando telefone aceito: {item} - Resultado: {result}")
                self.assertTrue(result, f"Falha ao aceitar '{item}'")
        for item in self.exemplos['telefone']['rejeitadas']:
            with self.subTest(item=item):
                result = Mascaras.mascara_telefone(item)
                logger.debug(f"Testando telefone rejeitado: {item} - Resultado: {result}")
                self.assertFalse(result, f"Falha ao rejeitar '{item}'")

if __name__ == '__main__':
    unittest.main()
```

Figure 11. testes unitários