Cleuton Sampaio.

Rust idiomático com ADT

Rust é uma linguagem bastante expressiva

Sim, **Rust é uma linguagem expressiva**.

> **O que significa ser expressiva?** Uma linguagem expressiva permite ao programador **descrever intenções claramente** e escrever código que seja **conciso**, **legível** e **seguro**, sem sacrificar o controle sobre os detalhes.

E ela foi criada para ser "safety by design", possuindo "compile-time safety" em muitas situações. É uma das principais filosofias da linguagem, baseada na ideia de que muitos erros devem ser detectados o mais cedo possível — durante a compilação, e não em tempo de execução.

Uma das coisas interessantes de Rust são as **enums**. As enums do Rust são associadas ao conceito de "Tipos de Dados Algebricamente" (**ADT** - Algebraic Data Types) porque elas permitem modelar dados de

maneira flexível, expressiva e estrutural, utilizando a combinação de dois conceitos fundamentais: tipos soma e tipos produto.

Vamos ver um exemplo de como criar um código bem simples, mas não muito seguro, e depois vamos transformá-lo em algo mais **idiomático** em Rust.

> Algo é **idiomático** em Rust, se segue as práticas recomendadas da linguagem, aproveita seus recursos específicos, como `Option` e `Result`, e reflete a filosofia de segurança e clareza do Rust. Isso significa escrever código que parece natural para quem conhece a linguagem, é seguro e explícito sobre possíveis falhas.

Versão simples

Imagine um código simples, que calcula as parcelas a serem pagas por um cliente:

Arquivo `sem_adt.rs`:

```
struct Pagamento {
    cliente: String,
    valor_a_vista: f32,
    parcelado: bool,
    numero_parcelas: i64,
}

fn calcular_valor_parcela(pagamento: &Pagamento) -> f32 {
    if pagamento.parcelado {
        return pagamento.valor_a_vista / pagamento.numero_parcelas as f32
    }
    0.0
}

fn main() {
    let pagto = Pagamento {
        cliente: "Fulano".to_string(),
        valor_a_vista: 1000.00,
        parcelado: true,
        numero_parcelas: 0,
    };
    println!("Valor da parcela: {} - {}", pagto.cliente,
calcular_valor_parcela(&pagto));
}
```

O que há de errado nesse código? A princípio nada, certo? Só que não é muito seguro. Podemos instanciar a struct `Pagamento` sem passar todos os dados necessários, deixando à cargo do desenvolvedor validar isso. E se ele esquecer?

Nesse exemplo, temos os atributos:

- cliente: Nome do cliente
- valor_a_vista: O valor da compra se for pago à vista

- parcelado: Se a compra foi parcelada
- numero_parcelas: Se a compra foi parcelada, a quantidade de parcelas

Notou que há um "if" implícito nessa struct? O atributo `numero_parcelas` só pode ser informado (`>=0`) se o atributo `parcelado` for verdadeiro. E a função `calcular_valor_parcela` testa essa condição.

E o desenvolvedor cometeu um erro ao instanciar a estrutura informando `parcelado` como verdadeiro e `numero_parcelas` zerado. O resultado é que o programa mostrará um resultado inválido em runtime **SEM DAR ERRO ALGUM**:

```
% cargo script sem_adt.rs
Valor da parcela: Fulano – inf
```

Esse `inf` é o resultado da divisão por zero, já que o pagamento foi PARCELADO com ZERO PARCELAS.

Rust foi criada para evitar isso, mas não pode fazer a mágica sozinha.

Versão com ADT e construtor seguro

A solução é mover o máximo possível de **invariantes** para as estruturas de dados, evitando erros. Se for possível, vamos tentar evitar erros em tempo de compilação, caso contrário, precisamos evitá-los em produção, mesmo que o desenvolvedor esqueça de verificar.

Uma solução é utilizar a característica dos **enums** em Rust que nos permite adicionar propriedades às suas variantes.

```
enum FormaPagamento {
    AVista,
    Parcelado { numero_parcelas: i64 },
}
```

Agora, em vez de termos um `bool` temos uma variante dessa enum e, caso selecionemos `Parcelado` teremos que passar o número de parcelas. Isso já torna o código bem mais expressivo e idiomático. Para usar podemos criar a struct assim:

```
struct Pagamento {
    cliente: String,
    valor_a_vista: f32,
    forma_pagamento: FormaPagamento,
}
```

Mas falta algo... Precisamos mover a verificação do número de parcelas zerado (ou menor que zero) para a estrutura de dados, em vez de deixar isso a cargo do resto do código. Infelizmente, não é possível verificar em tempo de compilação se o valor do número de parcelas é maior do que zero.

Uma solução é criar um construtor seguro para a **enum**:

```
impl FormaPagamento {
    /// Construtor para pagamentos à vista
    pub fn avista() -> Self {
        FormaPagamento::AVista
    }

    /// Construtor seguro para pagamentos parcelados.
    ///
    /// Retorna `Ok(FormaPagamento::Parcelado)` se `parcelas > 0`,
    /// ou `Err(...)` se for inválido.
    pub fn parcelado(parcelas: i64) -> Result<Self, String> {
        if parcelas > 0 {
            Ok(FormaPagamento::Parcelado { numero_parcelas: parcelas })
        } else {
            Err(format!("Número de parcelas inválido: {}", parcelas))
        }
    }
}
```

Agora, se você quiser criar um pagamento parcelado, terá que informar um número de parcelas maior que zero. E, para garantir que você sempre teste o erro, o construtor seguro retorna `Result<Self, String>` obrigando o desenvolvedor a verificar se deu erro:

```
    let numero_parcelas = 0; // isso dá erro!
    //let numero_parcelas = 5;
    match FormaPagamento::parcelado(numero_parcelas) {
        Ok(forma) => {
            let pagto = Pagamento {
                cliente: "Fulano".to_string(),
                valor_a_vista: 1000.0,
                forma_pagamento: forma,
            };
            println!("Valor da parcela: {} - {}", pagto.cliente,
calcular_valor_parcela(&pagto));
        }
        Err(e) => println!("Falhou ao criar pagamento parcelado: {}", e),
    }
```

O código completo está no arquivo `com_adt.rs`.

Conclusão

Quando falamos que **Rust** é uma linguagem expressiva e segura, podemos dar a impressão de que seria IMPOSSÍVEL fazer bobagem com ela. Isso não é verdade! A linguagem oferece mecanismos para criarmos código mais seguro, mas, como vimos no exemplo, sempre é possível fazer algo que quebre esse paradigma. Nesse caso, por que usar Rust, não é mesmo?