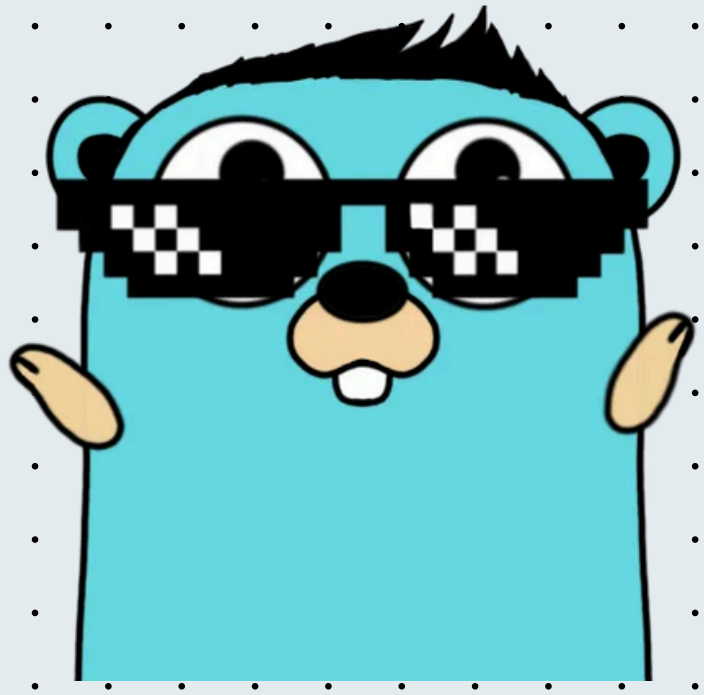
SOLID na prática

LISKOV SUBSTITUTION PRINCIPLE



Inspirado nos aprendizados adquiridos no curso Full Cycle e LeetCode.

/evelyncristinioliveira

SOLID na prática

LISKOV SUBSTITUTION PRINCIPLE

Temos uma classe base chamada de **Carro** e uma subclasse **Carro Esportivo** que herda todas as características de um carro, mas que pode possuir atributos e comportamentos adicionais. Existem diversas abordagens para aplicarmos LSP e um dos caminhos possíveis seria utilizar a função que recebe um objeto do tipo **Carro** responsável por verificar se possui 4 rodas passando num objeto do tipo **Carro Esportivo**, uma vez que o carro esportivo também possui 4 rodas.

A partir desse momento, entendemos que:

- Uma subclasse é um tipo específico de uma classe base;
- É possível usar a classe base, assim como a subclasse e;
- As subclasses não devem adicionar comportamentos que possam quebrar o contrato da classe base.

SOLID na prática

LISKOV SUBSTITUTION PRINCIPLE

No Go, não possuímos o conceito de herança que vemos nas linguagens orientadas à objetos. Ao invés disso, é possível trabalhar com composição e interfaces para alcançar o polimorfismo (objetos distintos sçao tratados como um tipo comum) e a reutilização de código.

De modo geral, o princípio afirma que se continuar a usar generalizações como as interfaces, não deverá ter herança ou não deverá ter implementadores dessas generalizações que quebrem algumas das suposições estabelecidas em um nível mais alto.

Será implementado a seguir um algoritmo para identificar se uma string *s* possui uma correspondência completa, de modo que haja uma relação entre uma letra no padrão e uma palavra não vazia em *s* sem aplicar boas práticas e um outro algoritmo aplicando o LSP do SOLID, respectivamente.

SOLID na prática

LISKOV SUBSTITUTION PRINCIPLE

EXEMPLO 1

Entrada: *pattern* = “abba”, *s* = “dog cat cat dog”

Saída: *true*

EXEMPLO 2

Entrada: *pattern* = “abba”, *s* = “dog cat cat fish”

Saída: *false*

EXEMPLO 3

Entrada: *pattern* = “aaaa”, *s* = “dog cat cat dog”

Saída: *false*

PADRÃO DE PALAVRAS

```go
package main

import (
	"fmt"
	“strings”
)

func WordPattern(pattern, s string) bool {
	words := strings.Fields(s)
	if len(words) != len(pattern) {
		return false
	}
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS

```
patternToWord := make(map[rune]string)
wordToPattern := make(map[string]rune)
for i, char := range pattern {
    word := words[i]
    if wordPattern, ok := patternWord[char]; ok {
        if wordPattern != word {
            return false
        }
    } else {
        patternToWord[char] = word
    }
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS

```
            if charPattern, ok := wordPattern[word]; ok {
                if charPattern != char {
                    return false
                }
            } else {
                wordToPattern[word] = char
            }
        }
        return true
    }
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS

```
func main() {
	s := "dog cat cat dog"
	pattern := "abba"
	if WordPattern(pattern, s) {
		fmt.Println("the string " + s + " follows the pattern
" + pattern + ".")
	} else {
		fmt.Println("the string " + s + " does not follow the
pattern " + pattern + ".")
	}
}
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```go
package main

import (
	“fmt”
	"strings"
)

type WordProcessor interface {
	Split(s, delimiter string) []string
}

type DefaultWordProcessor struct{}
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
func (d *DefaultWordProcessor) Split(s string) []string {
        return strings.Fields(s)
}

type CustomDelimiterWordProcessor struct {
        delimiter string
}

type DefaultWordProcessorAdapter struct {
        defaultWordProcessor *DefaultWordProcessor
}
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
func (a *DefaultWordProcessorAdapter) Split(s, delimiter
string) []string {
        return a.defaultWordProcessor.Split(s)
}

func WordPattern(pattern, s string, wordProcessor
WordProcessor) bool {
        words := wordProcessor.Split(s, "")
        if len(words) != len(pattern) {
            return false
        }
        patternToWord := make(map[rune]string)
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
wordToPattern := make(map[string]rune)
for i, char := range pattern {
    word := words[i]
    if wordPattern, ok := patternToWord[char]; ok {
        if wordPattern != word {
            return false
        }
    } else {
        patternToWord[char] = word
    }
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
        if charPattern, ok := wordToPattern[word]; ok {
            if charPattern != char {
                return false
            }
        } else {
            wordToPattern[word] = char
        }
    }
    return true
}
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
func main() {
	s := "dog cat cat dog"
	pattern := "abba"
	defaultProcessor := &DefaultWordProcessor{}
	adapter := &DefaultWordProcessorAdapter{defaultWordProcessor: defaultProcessor}
	if WordPattern(pattern, s, adapter) {
		fmt.Println("the string " + s + " follows the pattern " + pattern + ".")
	} else {
```

/evelyncristinioliveira

PADRÃO DE PALAVRAS COM LSP

```
            fmt.Println("the string " + s + " does not follow the
pattern " + pattern + ".")
        }
}
```

/evelyncristinioliveira