Desenvolvimento de Componentes Distribuídos – Programação por Contrato
Karen Christina Diz – 24/05/21

Descrição da Tarefa:
+ Como a programação por contrato exige, geralmente, uma ferramenta ou um módulo externo para ser implementado, mostre e explique em um documento como esta técnica funciona para a linguagem que você escolheu.
+ Assim explique e exemplifique como seria a criação de um contrato na linguagem que você está utilizando para desenvolver seu projeto.
+ Não é necessário implementar o projeto utilizando de contrato, apenas demonstrar como seria uma classe que utilize este Design Pattern.
+ Não esqueça de explicar, com suas palavras, o que é este Design Pattern e em quais situações você acredita que ele seja útil.

Programação por Contrato e Padrões de Projeto

A programação por contrato, que vem do termo inglês *Design by Contract (DbC),* é uma abordagem utilizada no desenvolvimento de softwares, focada na documentação e na aceitação dos direitos e responsabilidades dos módulos de software, aumentando a confiabilidade na correção dos softwares, e firmando um acordo que no software não terá nada além do que acordado previamente no contrato, ou seja, se ao realizar uma operação e ela não atender as condições pré-estabelecidas ela retorna como um erro.

Esse contrato tem algumas expectativas, e pode ser dividido em 3 partes principais:
- Pré-condições: o que deve ser verdade para a rotina ser chamada, ou seja, os requisitos necessários.
- Pós-condições: o que a rotina garante fazer após executar, e ela sempre vai ser concluída.
- Invariantes: são condições que são sempre verdadeiras da perspectiva do chamador, essas invariantes podem ser desrespeitadas durante a execução da rotina, mas depois da execução da rotina devem ser sempre verdadeiras.

Essa ideia foi implementada em 1992 na linguagem Eiffel, por Bertrand Meyer que criou tanto a linguagem com essa ideia de *Design by Contract*. Mas essa ideia só foi amplamente difundida em 1995, com a publicação do livro *Design Patterns: Elements of Reusable Object Oriented Software*, da chamada *Gang of Four* (GoF), composta por, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides.

Assim o termo *Design Patterns* ficou muito mais conhecido e difundido no mundo da programação. No livro esses quatro autores aplicaram o conceito de padrões de projeto para programação coletando 23 padrões para projetos, e dividindo em 3 grupos:

+ Padrões de criação: tratam da criação do objeto e de referência, que aumentam a flexibilidade e a reutilização de código.

+ Padrões estruturais: tratam da relação dos objetos e como eles se tratam entre si para formar grandes objetos complexos mantendo ainda as estruturas flexíveis e eficientes.

+ Padrões comportamentais: tratam da comunicação eficiente dos objetos em termos de responsabilidade e algoritmos.

E um padrão pode ser composto por 4 elementos principais:

+ Intenção: descrição breve do problema e sua solução

+ Motivação: explica melhor o problema e as possibilidades do padrão

+ Estrutura: mostra as classes e como se relacionam (UML)

+ Exemplo de código: geralmente em alguma linguagem popular

Assim a partir da criação desses contratos nos códigos é possível aumentar a confiabilidade, e a robustez do mesmo na programação orientada, além da possível e ampla reutilização dos códigos considerando que todos esses contratos serão devidamente documentados, e testados.

Python

Em Python o conceito de *Design Patterns* é o mesmo das outras linguagens, porém algo que não é muito comentado é que esses padrões eles funcionam e foram utilizados por muito tempo em linguagens estáticas, compiladas, já o python é uma linguagem dinâmica, interpretada, então nem todos os '*Patterns*' fazem sentido em serem implementados em todas as linguagens, considerando que cada linguagem tem suas peculiaridades, além de que hoje muitos padrões são *default.*

Um exemplo de padrão em python que poderíamos utilizar de forma adequada e útil seria o *Adapter.* O Adaptador é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborem entre si.

Por exemplo, se eu tenho uma pessoa que fala português, outra, inglês, outra, espanhol, considerando que todas essas pessoas só conhecem e entendem o seu próprio idioma, e esquecendo que hoje existe tradutor. Levando isso em consideração eu preciso passar uma mensagem para essas pessoas de forma que todas elas consigam entender, então eu utilizo um adaptador, faço uma adaptação no meu código para que todas essas pessoas entendam a mensagem. Para isso eu desenvolvi o seguinte código:

```python
#  CRIANDO AS CLASSES  #

class Pessoa01:
    def portugues(self):
        print('Entenda o que eu estou falando')


class Pessoa02:
    def ingles(self):
        print("Understand what i'm talking about")


class Pessoa03:
    def espanhol(self):
        print('Comprenda lo que yo estoy hablando')


#  CRIANDO O MÉTODO ADAPTADOR DE FALA  #


class Pessoa01Adapter:
    def __init__(self, pessoa01):
        self.pessoa01 = pessoa01

    def falar(self):
        self.pessoa01.portugues()


class Pessoa02Adapter:
    def __init__(self, pessoa02):
        self.pessoa02 = pessoa02

    def falar(self):
        self.pessoa02.ingles()


class Pessoa03Adapter:
    def __init__(self, pessoa03):
        self.pessoa03 = pessoa03

    def falar(self):
        self.pessoa03.espanhol()


#  CHAMANDO OS ADAPTADORES  #

pessoas = [
    Pessoa01Adapter(Pessoa01()),
    Pessoa02Adapter(Pessoa02()),
    Pessoa03Adapter(Pessoa03())
]
for pessoa in pessoas:
    pessoa.falar()
```

Mas pra facilitar ainda mais poderíamos criar um adaptador único, como se fosse ‘poliglota’ dessa forma:

```
#  CRIANDO AS CLASSES  #

class Pessoa01:
    def portugues(self):
        print('Entenda o que eu estou falando')


class Pessoa02:
    def ingles(self):
        print("Understand what i'm talking about")


class Pessoa03:
    def espanhol(self):
        print('Comprenda lo que yo estoy hablando')


#   MÉTODO ADAPTADOR ÚNICO DE FALA  #


class FalaAdapter:

    def __init__(self, pessoa, *, falar):
        self.pessoa = pessoa
        metodo_de_fala = getattr(self.pessoa, falar)
        self.__setattr__('falar', metodo_de_fala)


pessoas_adapters = [
    FalaAdapter(Pessoa01(), falar='portugues'),
    FalaAdapter(Pessoa02(), falar='ingles'),
    FalaAdapter(Pessoa03(), falar='espanhol'),
]

for adapter in pessoas_adapters:
    adapter.falar()
```

Assim eu crio um único adaptador, e se eu quiser incluir outro idioma eu só preciso colocar nesse adaptador, e consigo me comunicar e mandar a mensagem para todos uma única vez com o meu método ‘falar’.

Concluindo assim, é possível entender que o padrão foi utilizado, pois, eu já tinha um problema e precisava de uma resposta para ele, então o uso do padrão solucionou o meu problema e ainda fez com que o código fosse mais claro, conciso, eficiente e principalmente que eu poderei aproveitar ele para utilizar com outros idiomas.