Flyweight
Padrão de Projeto

Caique Pereira, Euler Carvalho

Agenda

- Padrões de Projeto
- Vantagens
- GoF - Padrões Estruturais
- Técnicas utilizadas
- O padrão Flyweight
- Implementando o Flyweight com Python
- Considerações Finais

Padrões de Projeto

"Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementá-la duas vezes da mesma forma"

Christopher Alexander

Vantagens

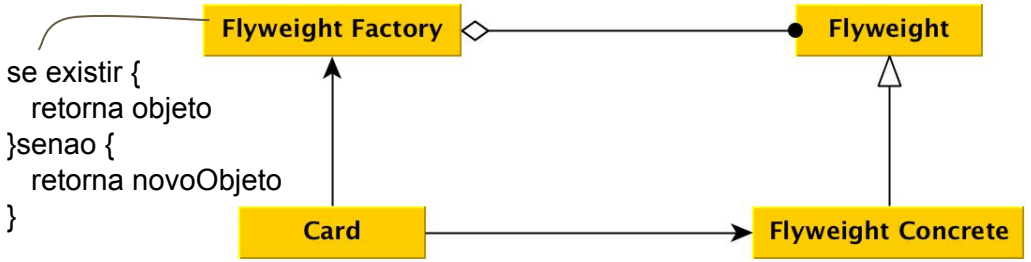
- Reutilização de Código
- Separação e implementações guiadas por interfaces
- Redução significativa na complexidade do código
- Código com maior qualidade
- Modularização
- [...]

GoF - Padrões Estruturais

- Se preocupam com a composição estrutural das classes e objetos
- Facilitam a comunicação entre todas as entidades do projeto
- Geralmente faz uso do conceito de herança para suas implementações
- Descrevem determinadas formas de compor objetos para criação de novas funcionalidades, ao invés da utilização de composição de interfaces
- Especificam os métodos que conectam os objetos, ao invés de suas referências
- Definem como cada componente ou entidade deve ser estruturado (modularização)

GoF - Padrões Estruturais

- Adapter
- Decorator
- Proxy
- **Flyweight**
- Bridge
- Facade
- Composite



O Padrão Flyweight

Nome: Flyweight

Problema: Duplicação de objetos idênticos

Solução: Utilizar o compartilhamento para suportar eficientemente grandes quantidades de objetos

Intenção: Criar uma estrutura de compartilhamento de objetos

O Padrão Flyweight

- Padrão de design utilizado para minimizar o uso de recursos
- Reduz o consumo de memória quando é necessário criar muitos objetos **idênticos**, que podem tranquilamente **ser compartilhados em diferentes contextos**
- Ou seja, procura definir um ponto de compartilhamento e manipulação de objetos reutilizáveis

Implementando o Flyweight com Python

https://github.com/tiagorc/Flyweight/

Sem utilizar o padrão

```
values = ('2', '3', '4', '5', '6', '7', '8', '9', 'J', 'Q', 'K', 'A') # cartas
suits = ('h', 'c', 'd', 's') #naipes

class Card:
    def __init__(self, value, suit):
        self.value, self.suit = value, suit

if __name__ == '__main__':
    card1 = Card('J', 'h')
    card2 = Card('J', 'h')
    print ("Card 1: %s" % id(card1))
    print ("Card 2: %s" % id(card2))
```

Utilizando o Padrão

```
import weakref
    _CardPool = weakref.WeakValueDictionary()

    def __new__(cls, value, suit):
        obj = Card._CardPool.get(value + suit, None)

        if not obj:
            obj = object.__new__(cls)
            Card._CardPool[value + suit] = obj
            obj.value, obj.suit = value, suit

        return obj
```

Considerações Finais

- Ponto fraco: Dependendo da quantidade e da organização dos objetos a serem compartilhados, o custo pela busca de objetos por sair alto
  - ++ Espaço == -- Consumo de memória *versus* ++ Tempo de execução
- A combinação de vários padrões de projeto é sempre a melhor alternativa para um bom código

Obrigado :)