FlyWeight Pattern

Peso Mosca

Usa compartilhamento para dar suporte a vários objetos de forma eficiente.

Acadêmicos:

Cleverton Hoffmann
Rodrigo Odorizzi

Descrição

Flyweight é um padrão de projeto de software apropriado quando vários objetos devem ser manipulados em memória sendo que muitos deles possuem informações repetidas. Dado que o recurso de memória é limitado, é possível segregar a informação repetida em um objeto adicional que atenda as características de imutabilidade e comparabilidade (que consiga ser comparado com outro objeto para determinar se ambos carregam a mesma informação). Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina.

Problema que resolve

O padrão de peso da mosca é usado principalmente para reduzir o número de objetos criados e para diminuir o consumo de memória e aumentar o desempenho. Este tipo de padrão de projeto vem sob padrão estrutural, pois esse padrão fornece maneiras de diminuir a contagem de objetos, melhorando assim a estrutura de objeto da aplicação. O padrão de peso-mosca tenta reutilizar objetos de tipo semelhantes já existentes armazenando-os ou cria um novo objeto quando nenhum objeto correspondente é encontrado.

Aplicabilidade

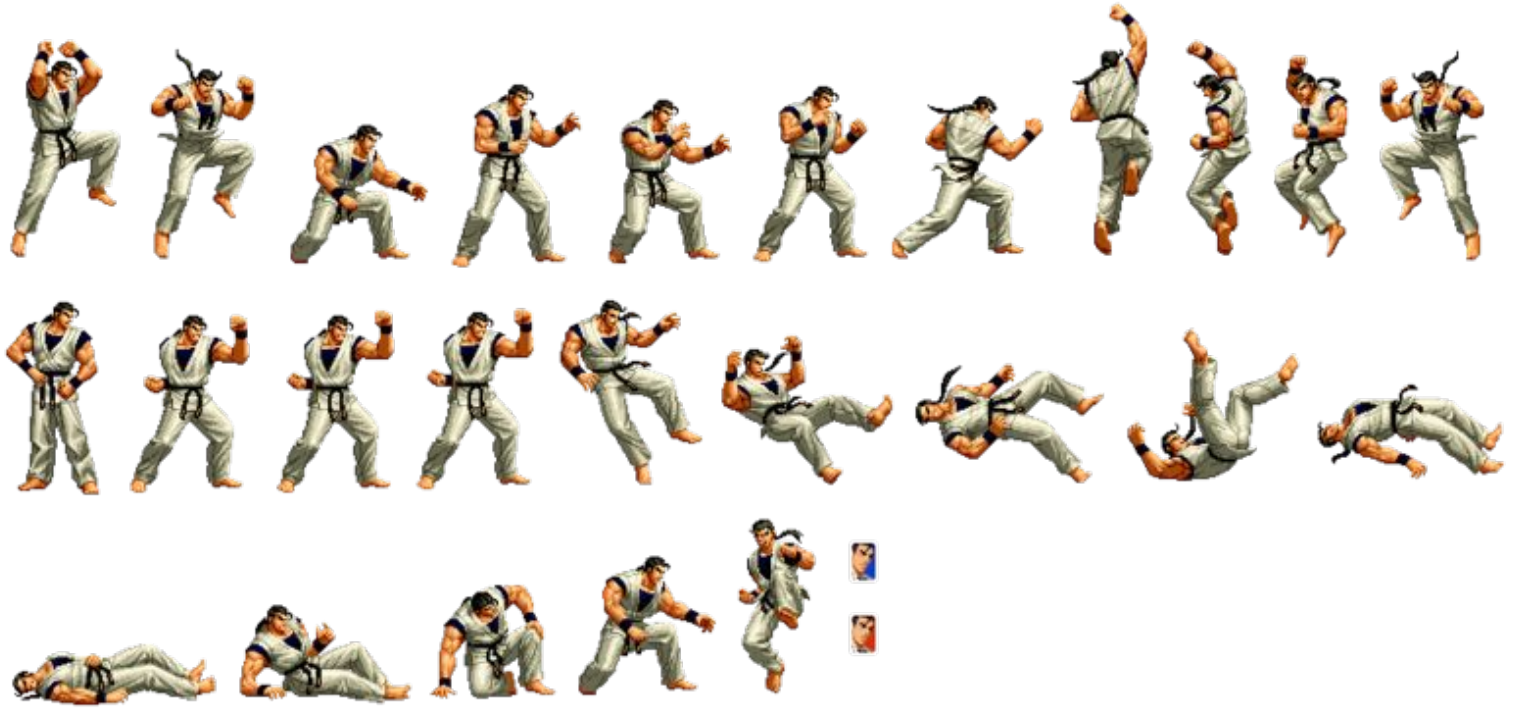
A eficiência do padrão Flyweight depende muito de como e onde ele é usado. Aplique o padrão Flyweight quando todas as condições a seguir forem verdadeiras:

- Uma aplicação utiliza um grande número de objetos;
- Os custos de armazenamento são altos por causa da grande quantidade de objetos;
- A maioria dos estados de objetos pode ser tornada extrínseca;
- Muitos grupos de objetos podem ser substituídos por relativamente poucos objetos compartilhados, uma vez que estados extrínsecos são removidos;
- A aplicação não depende da identidade dos objetos. Uma vez que objetos Flyweights podem ser compartilhados, testes de identidade produzirão o valor verdadeiro para objetos conceitualmente distintos.

Métodos Intrínseco e Extrínseco

O estado intrínseco é armazenado no flyweight; ele consiste de informações independentes do contexto do flyweight, desta forma tornando-o compartilhado.

O estado extrínseco depende de/e varia com o contexto do flyweight e, portanto, não pode ser compartilhado. Os objetos-cliente são responsáveis pela passagem de estados extrínsecos para o flyweight quando necessário.



Fonte: https://silvrback.s3.amazonaws.com/uploads/7f49efcd-34df-4193-a470-6fb2baf6536a/Combos_large.png

Atores envolvidos

• Flyweight (Glyph)
– declara uma interface através da qual flyweights podem receber e atuar sobre estados extrínsecos.

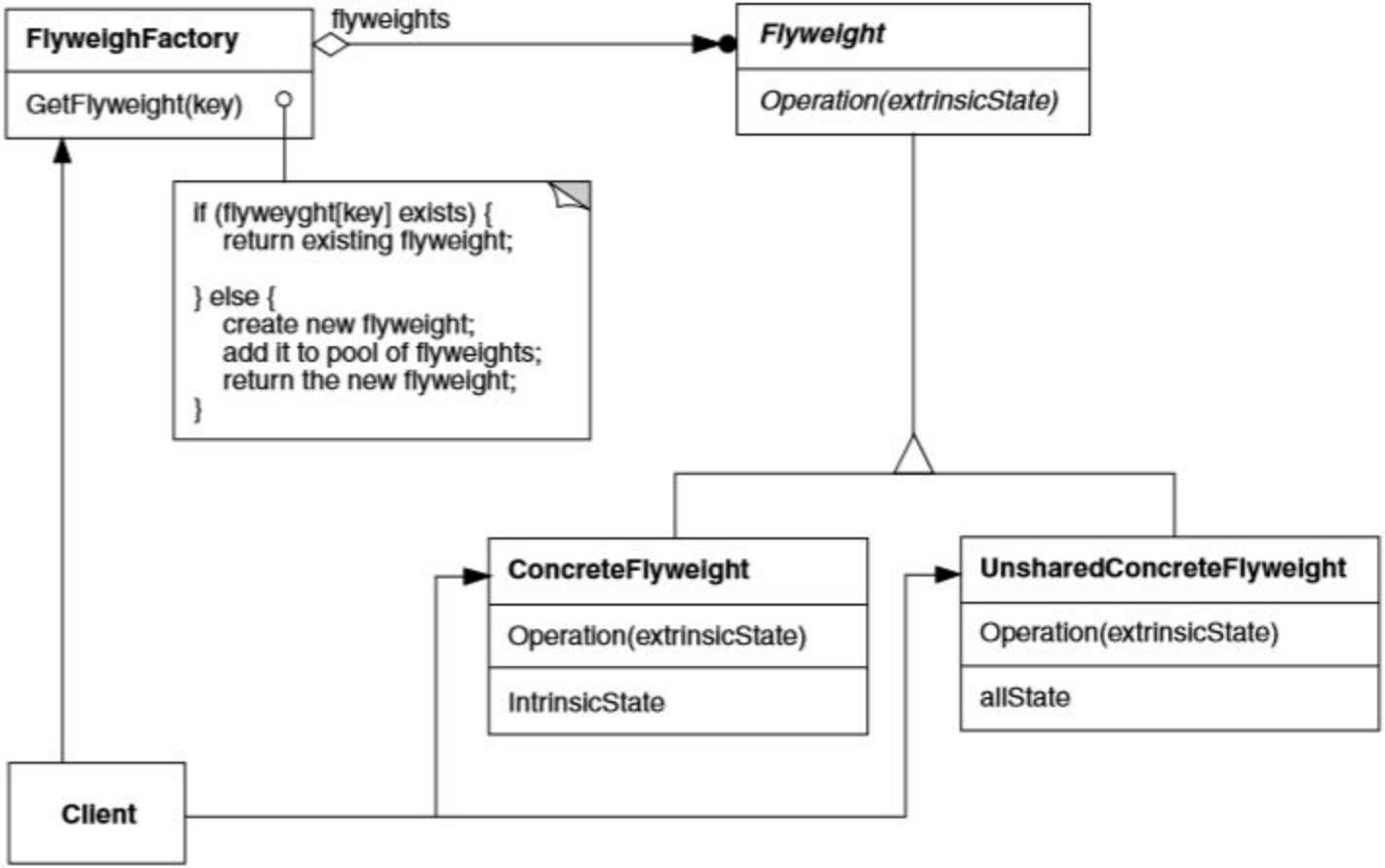
• ConcreteFlyweight (Character)
– implementa a interface de Flyweight e acrescenta armazenamento para estados intrínsecos, se houver. Um objeto ConcreteFlyweight deve ser compartilhável. Qualquer estado que ele armazene deve ser intrínseco, ou seja, independente do contexto do objeto ConcreteFlyweight.

Client
– mantém uma referência para flyweight(s);
– computa ou armazena o estado extrínseco do flyweight(s).

Atores envolvidos

• UnsharedConcreteFlyweight (Row, Column)
– nem todas as subclasses de Flyweight necessitam ser compartilhadas. A interface de Flyweight habilita o compartilhamento; ela não o força ou o garante. É comum para objetos UnsharedConcreteFlyweight não compartilhar objetos ConcreteFlyweight como filhos em algum nível da estrutura de objetos de Flyweight (tal como o fazem as classes Row e Column).

• FlyweightFactory
– cria e gerencia objetos flyweight;
– garante que os flyweights sejam compartilhados apropriadamente. Quando um cliente solicita um flyweight, um objeto FlyweightFactory fornece uma instância existente ou cria uma, se nenhuma existir.


FlyweighFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweyght[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
IntrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Exemplo 1 - Criando objetos de notas musicais

Problema: Queremos criar uma música onde teremos diversos sons/notas.

A questão é que não queremos repetir o processo de criação dos objetos das notas por que isso tornaria o programa lento ao ser executado. Como podemos desenvolver um sistema capaz de criar apenas um objeto de uma determinada nota armazená-lo no sistema e utilizá-lo quando necessário?

Lembrando que uma música é formada por uma sequência de notas executadas dentro de um período de tempo. E a nota pode ser considerada um som.

O método implementado é Flyweight intrínseco.

Classe Nota

```
public class Nota {

    protected String nota;

    public Nota(String nota) {
        this.nota = nota;
    }
}
```

Classe AbstractFlyweight

```
public abstract class AbstractFlyweight {

    public abstract void tocarNota();

}
```

Classe Flyweight

```
import org.jfugue.player.Player;

public class Flyweight extends AbstractFlyweight{

    protected Nota nota;

    public Flyweight(String notaN) {
        nota = new Nota(notaN);
    }

    @Override
    public void tocarNota() {
        Player player = new Player();
        player.play(nota.nota);
       System.out.println(nota.nota);
    }
}
```

Biblioteca de música:

http://www.jfugue.org/download.html

Classe FlyweightFactory

```
public class FlyweightFactory {

    protected ArrayList<AbstractFlyweight>
listanotas;

    public enum Posicao {
        DO, RE, MI, FA, SOL
    }

    public FlyweightFactory() {
        listanotas = new
ArrayList<AbstractFlyweight>();
        listanotas.add(new Flyweight("C"));
        listanotas.add(new Flyweight("D"));
        listanotas.add(new Flyweight("E"));
        listanotas.add(new Flyweight("F"));
        listanotas.add(new Flyweight("G"));
    }
```

Classe FlyweightFactory

```
  public AbstractFlyweight
getFlyweight(Posicao nota) {
        switch (nota) {
        case DO:
            return listanotas.get(0);
        case RE:
            return listanotas.get(1);
        case MI:
            return listanotas.get(2);
        case FA:
            return listanotas.get(3);
        default:
            return listanotas.get(4);
        }
    }

}
}
```

Classe Main

```
public static void main(String[] args) {

        FlyweightFactory factory = new FlyweightFactory();
        factory.getFlyweight(Posicao.DO).tocarNota();
        factory.getFlyweight(Posicao.RE).tocarNota();
        factory.getFlyweight(Posicao.MI).tocarNota();
        factory.getFlyweight(Posicao.FA).tocarNota();
        factory.getFlyweight(Posicao.FA).tocarNota();
        factory.getFlyweight(Posicao.DO).tocarNota();
        factory.getFlyweight(Posicao.RE).tocarNota();
        factory.getFlyweight(Posicao.DO).tocarNota();
        factory.getFlyweight(Posicao.RE).tocarNota();
        factory.getFlyweight(Posicao.SOL).tocarNota();
        factory.getFlyweight(Posicao.FA).tocarNota();
        factory.getFlyweight(Posicao.MI).tocarNota();
        factory.getFlyweight(Posicao.MI).tocarNota();
        factory.getFlyweight(Posicao.DO).tocarNota();
        factory.getFlyweight(Posicao.RE).tocarNota();
        factory.getFlyweight(Posicao.MI).tocarNota();
        factory.getFlyweight(Posicao.FA).tocarNota();
    }
```

Saída


Saída
POO (run)
Console do Depurador
run:
C
D
E
F
F
C
D
C
D
G
F
E
E
C
D
E
F
CONSTRUÍDO COM SUCESSO (tempo total: 17 segundos)

Explicação das Classes

Classe Nota: Classe que contém a nota instanciada.

Classe AbstractFlyweight: Classe abstrata onde há o método abstrato tocar nota.

Classe Flyweight: Classe que extende a classe AbstractFlyweight e faz a criação do método tocar nota.

Classe FlyweightFactory: Cria uma fábrica de classes Flyweight, e onde são instanciadas as notas.

Classe Main: Classe principal.

Exemplo 2 - Criando diversos objetos de Círculos

Problema: Queremos criar diversos objetos círculos com suas respectivas áreas, como podemos fazer isso sem repetir o processo de criação do mesmo círculo?

Considerando que podemos definir a partir da chamada de método um novo círculo e este deverá ser adicionado na lista de círculos.

O método implementado é Flyweight extrínseco.

Classe Círculo

```
public class Circulo {

    protected Double raio;
    public Double Area;

    public Circulo(Double raio) {
        this.raio = raio;
        calculaArea(raio);
    }

    public void calculaArea(Double raio){
        this.Area = raio*raio*Math.PI;
    }

    public void mostraCirculo(){
        System.out.println ("Circulo{" + "raio=" + raio + ", Area=" + Area + '}');
    }
}
```

Classe AbstractFlyweight

```
public abstract class AbstractFlyweight {

    public abstract Circulo getCirculo();

}
```

Classe Flyweight

```
public abstract class AbstractFlyweight {

    public abstract Circulo getCirculo();

}

public class Flyweight extends AbstractFlyweight{

    public Circulo circulo;

    public Flyweight(Double raio) {
        circulo = new Circulo(raio);
    }

    @Override
    public Circulo getCirculo() {
        return circulo;
    }
}
```

Classe FlyweightFactory

```
public class FlyweightFactory {

    protected final ArrayList<AbstractFlyweight> listaCirculos;

    public FlyweightFactory() {
        listaCirculos = new ArrayList<AbstractFlyweight>();
        listaCirculos.add(new Flyweight(2.0));
        listaCirculos.add(new Flyweight(3.0));
        listaCirculos.add(new Flyweight(4.0));
        listaCirculos.add(new Flyweight(5.0));
        listaCirculos.add(new Flyweight(6.0));
        listaCirculos.add(new Flyweight(7.0));
    }
```

Classe FlyweightFactory

```
public Circulo getFlyweight(Double raio) {
        for(AbstractFlyweight c: listaCirculos){
            if(c.getCirculo().raio.equals(raio)){
                return c.getCirculo();
            }
        }
        listaCirculos.add(new Flyweight(raio));
        return listaCirculos.get(listaCirculos.size()-1).getCirculo();
    }

}
```

Classe Main

```
public class ConstruirCirculos {

    public static void main(String[] args) {

        FlyweightFactory factory = new FlyweightFactory();
        factory.getFlyweight(2.0).mostraCirculo();
        factory.getFlyweight(3.0).mostraCirculo();
        factory.getFlyweight(8.0).mostraCirculo();
        factory.getFlyweight(9.0).mostraCirculo();
        factory.getFlyweight(9.0).mostraCirculo();

    }

}
```

Saída

POO (run) × Console do Depurador ×

```
run:
Circulo{raio=2.0, Area=12.566370614359172}
Circulo{raio=3.0, Area=28.274333882308138}
Circulo{raio=8.0, Area=201.06192982974676}
Circulo{raio=9.0, Area=254.46900494077323}
Circulo{raio=9.0, Area=254.46900494077323}
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Explicação das Classes

Classe Círculo: Classe que contém o raio e área do círculo instanciada.

Classe AbstractFlyweight: Classe abstrata onde há o método abstrato círculo.

Classe Flyweight: Classe que extende a classe AbstractFlyweight e retorna o círculo.

Classe FlyweightFactory: Cria uma fábrica de classes Flyweight, e onde são instanciadas os círculos e onde contém a validação dos círculos e a lista de objetos.

Classe Main: Classe principal, pode-se passar um dado possibilitando a inserção e criação de objetos ainda não existentes.

Dúvidas e Questionamentos

Referências

- GAMA, Erich; HELM, Richard;JOHNSON, Ralph; VLISSIDES,John. **Padrões de projeto** - Soluções reutilizáveis de software orientado a objetos.
- https://padroesdeprojetoifc.wordpress.com/2016/11/22/padrao-flyweight/
- https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm
- https://brizeno.wordpress.com/category/padroes-de-projeto/flyweight/
- http://www.ic.unicamp.br/~vanini/mc857/PadroesDeProjeto.pdf
- http://javaexplorer03.blogspot.com/2015/09/flyweight-pattern.html
- http://www.ybadoo.com.br/tutoriais/poo/09/
- http://www.fluffycat.com/Java-Design-Patterns/Flyweight/
- http://blog.triadworks.com.br/evitando-duplicacao-de-objetos-com-flyweight
- http://www.inf.ufes.br/~vitorsouza/wp-content/uploads/java-br-curso-padroesdeprojeto-slides03.pdf
- https://sourcemaking.com/design_patterns/flyweight
- Biblioteca de música. http://www.jfugue.org/download.html