Flyweight

Padrões Estruturais

O padrão *Flyweight* permite usar compartilhamento para suportar grandes quantidades de objetos de granularidade fina.

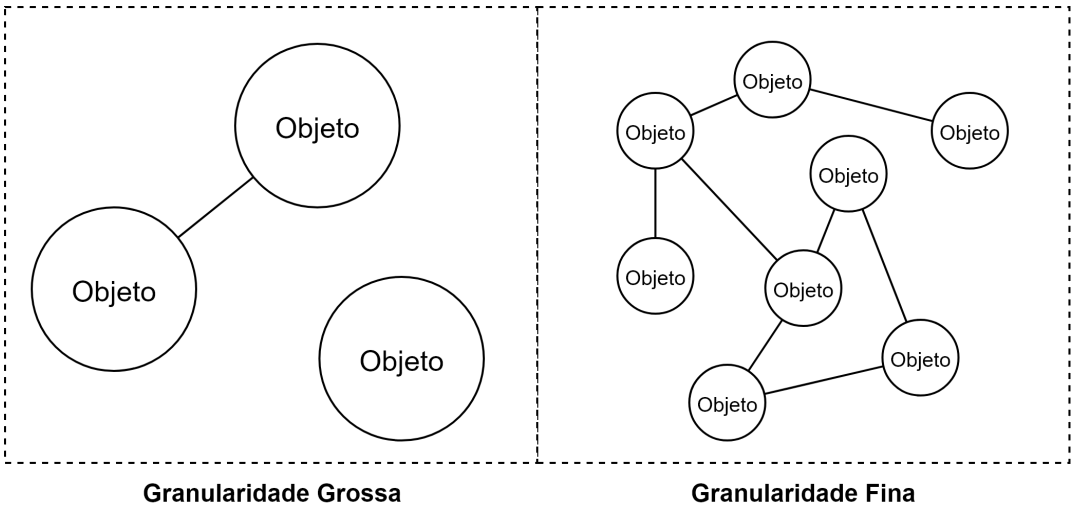
Motivação (Por que utilizar?)

Em programação orientada a objetos o nível de granularidade de um objeto é proporcional ao nível de detalhes que ele contém, ou seja, quão específico é um objeto.

A granularidade é utilizada para mensurar a profundidade de abstração de um *software*. Ela pode ser dividida em duas partes:

- **Granularidade grossa** (coarse-grained): Poucos objetos grandes, onde cada um realiza muitas operações. Os objetos são mais autossuficientes e não precisam se comunicar com muitos outros objetos para conseguir outras operações.

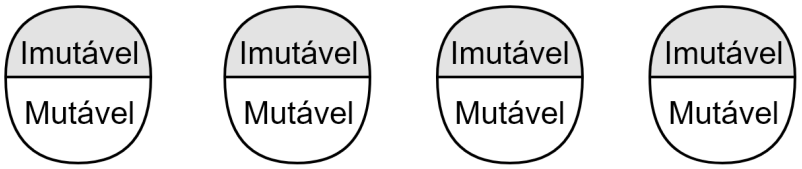
- **Granularidade fina** (fine-grained): Muitos objetos pequenos, onde cada um realiza poucas operações específicas e se comunicam com outros objetos para obter outras operações.



Esquema dos tipos de granularidade de objetos

Cada uma das abordagens acima apresenta vantagens e desvantagens. Para entender o padrão *Flyweight* apenas o conceito de granularidade já é o suficiente, por este motivo não iremos nos aprofundar neste assunto.

Em projetos de *software* podem existir situações onde muitos objetos de granularidade fina podem ser necessários em uma determinada funcionalidade. Muitos objetos instanciados ao mesmo tempo consomem muita memória RAM.



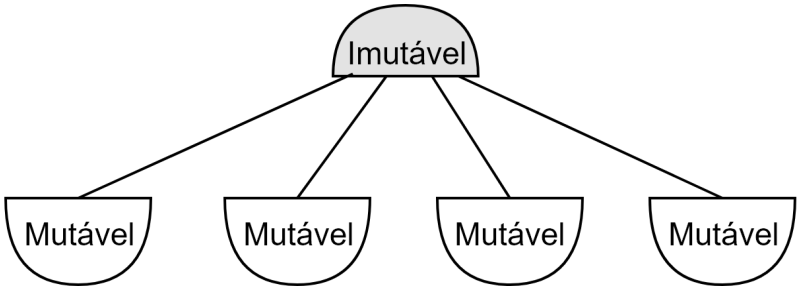
4 objetos completos em memória sem compartilhamento

Na imagem acima vemos a representação de 4 objetos em memória RAM. A parte mutável de cada objeto pode ser alterada em tempo de execução, já a parte imutável nunca muda depois que o objeto é criado. Considere que as partes imutáveis destes objetos sejam idênticas e tenham 50kB de tamanho e desconsidere o tamanho da parte mutável. Somando o tamanho das 4 partes imutáveis teríamos 200kB em memória.

O padrão *Flyweight* permite usar o compartilhamento desta parte imutável entre todos os objetos. A parte imutável possui um **estado intrínseco** e é armazenada no *Flyweight*, já a parte mutável possui um **estado extrínseco** e varia conforme o contexto do *Flyweight*, por isso não pode ser armazenado nele e também não pode ser compartilhada.

Conceito:

- **Estado Intrínseco:** os dados são imutáveis e independentes do contexto.
- **Estado Extrínseco:** dependentes ou variam conforme o contexto. Portanto, não podem ser compartilhados.



4 objetos completos em memória com compartilhamento

Na imagem acima, também considerando que a parte imutável tenha 50kB porém teríamos apenas uma destas partes em memória. Agora temos 50kB em memória utilizando compartilhamento da parte imutável dos objetos contra 200kB sem usar compartilhamento. Escalando esta situação imagine se ao invés de 4 desses objetos tivéssemos centenas ou milhares deles, seria necessário muita memória RAM.

Para ilustrar considere como exemplo o cenário onde precisamos construir o *back-end* um *software* para planejamento de projetos de reflorestamento de áreas desmatadas. Precisamos permitir que o usuário coloque uma determinada espécie de árvore em uma coordenada X, Y de um plano (duas dimensões).

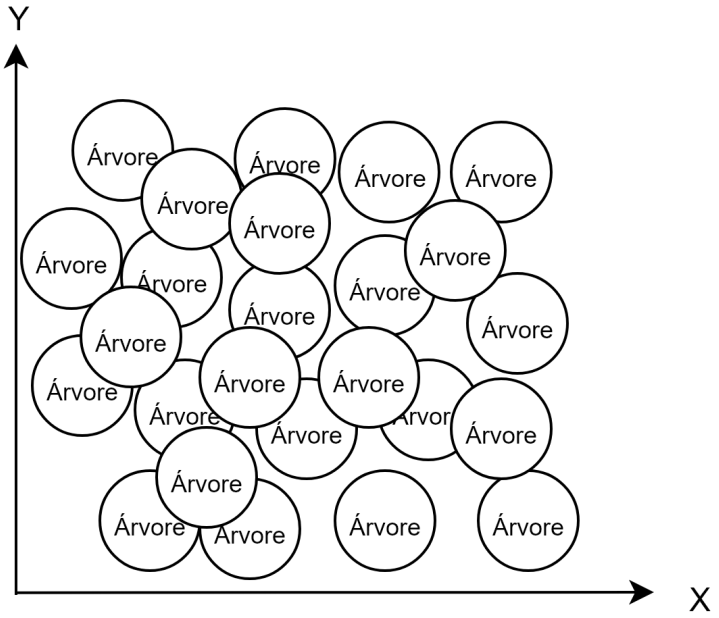


Ilustração de um projeto feito no software que precisamos criar

Cada árvore possui os seguintes atributos:

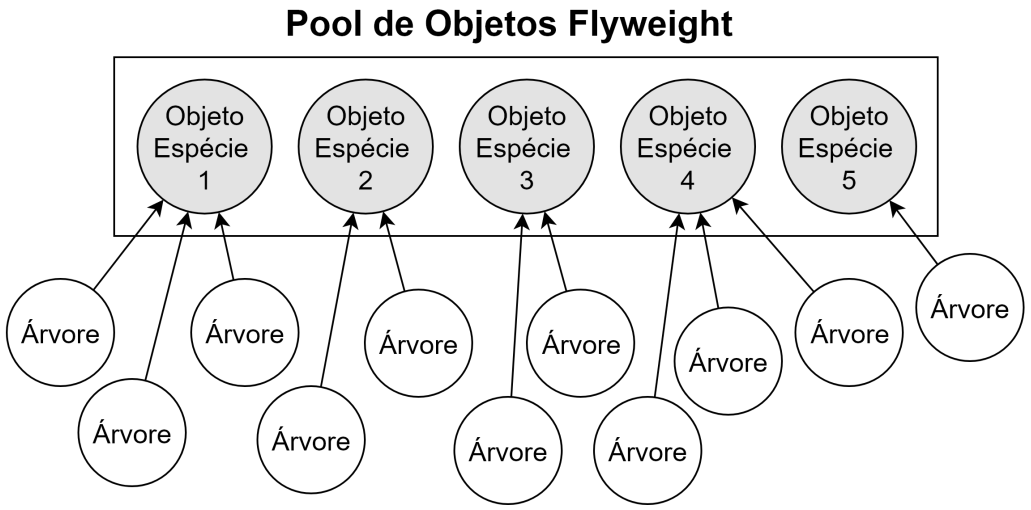
- Posição no eixo X;
- Posição no eixo y;
- Nome da espécie;
- Cor;
- Altura máxima.

Uma mesma espécie de árvore sempre terá o mesmo nome de espécie, cor e altura máxima, os únicos atributos que variam são as posições nos eixos x e y. Neste cenário podemos utilizar o padrão *Flyweight* e compartilhar um objeto espécie entre todos os objetos árvores que pertençam à espécie em questão:

Objeto Árvore separado em parte mutável e não mutável.

Uma árvore tem uma espécie, teremos várias árvores de várias espécies. Devido ao compartilhamento de estados, não é interessante que o cliente crie diretamente os *flyweights*, por este motivo iremos terceirizar essa tarefa para um objeto fábrica que pode decidir entre criar um *flyweight* ou retornar um já criado.

Os *flyweights* criados pela fábrica vão formando um conjunto de objetos compartilháveis que poderão ser reaproveitados pela fábrica de árvores quando ela encontrar uma oportunidade.



Objetos Árvore compartilhando objetos Espécie

Vejamos como ficaria o diagrama de classes do nosso projeto.

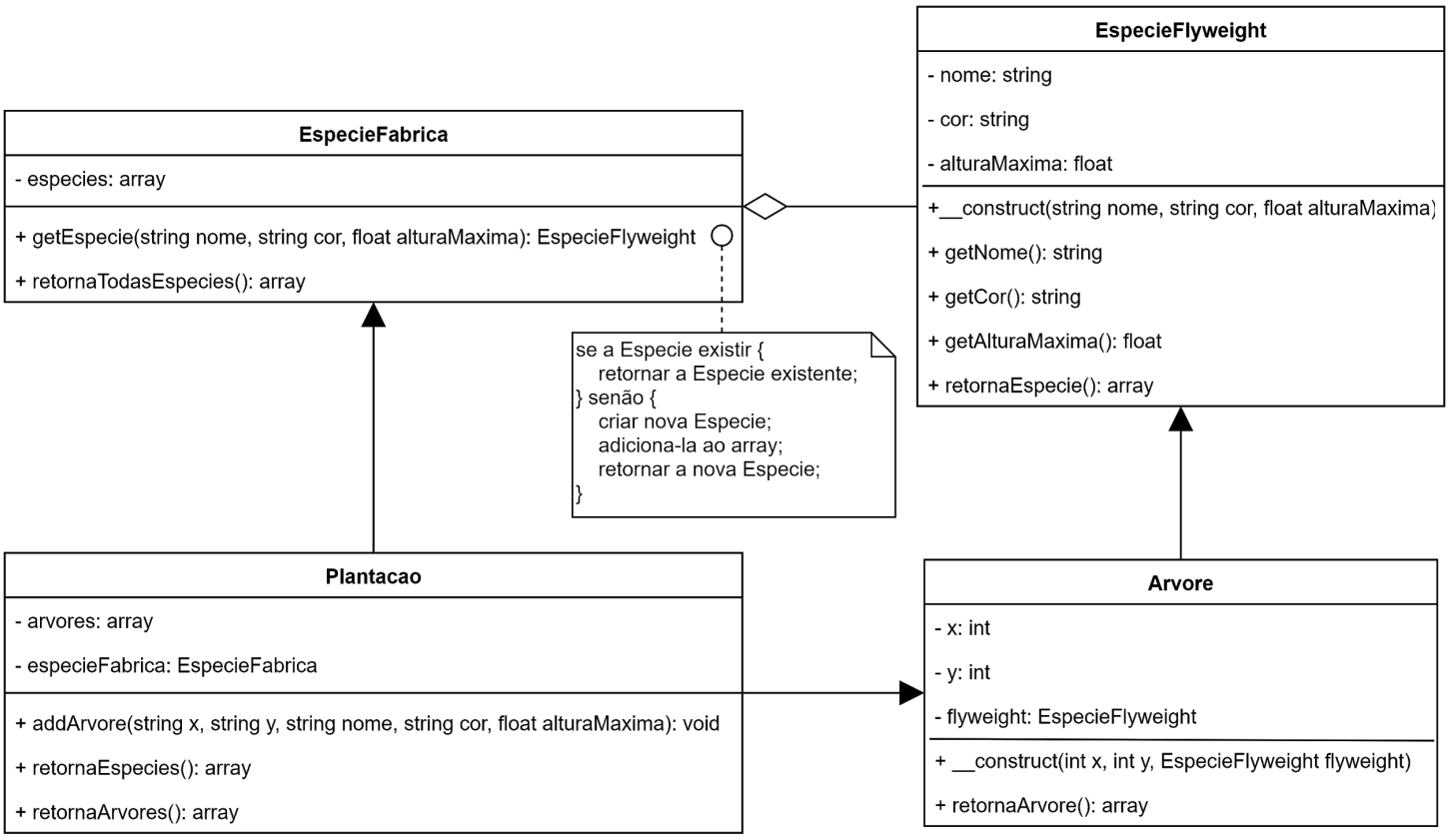


Diagrama de classes do software para projetos de reflorestamento

Interpretando o diagrama de classes podemos notar que **Arvore** pertence a uma **EspecieFlyweight,** e **EspecieFabrica** mantêm as instâncias **EspecieFlyweight** que já foram criadas.

Vamos iniciar a implementação pela classe **EspecieFlyweight.** Ela é uma classe muito simples, possui os atributos não mutáveis de uma árvore e os métodos para recuperar o valor de tais atributos.

```php
class EspecieFlyweight
{
    private string $nome;
    private string $cor;
    private float $alturaMaxima;

    //Recebe todos os valores no construtor
    public function __construct(string $nome, string $cor, float $alturaMaxima)
    {
        $this->nome = $nome;
        $this->cor = $cor;
        $this->alturaMaxima = $alturaMaxima;
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getCor(): string
    {
        return $this->cor;
    }

    public function getAlturaMaxima(): float
    {
        return $this->alturaMaxima;
    }

    //Retorna um array com os dados de EspecieFlyweight
    public function retornaEspecie(): array
    {
        return [
            'nome' => $this->nome,
            'cor' => $this->cor,
            'alturaMaxima' => $this->alturaMaxima
        ];
    }
}
```

Enquanto a classe **EspecieFlyweight** representa as informações não mutáveis de uma árvore, a classe **Arvore** mantém as informações mutáveis e também uma referência a um objeto de **EspecieFlyweight** que representa sua espécie.

```php
class Arvore
{
    private int $x;
    private int $y;
    private EspecieFlyweight $flyweight;

    //Recebe todos os valores no construtor
    public function __construct(int $x, int $y, EspecieFlyweight $flyweight)
    {
        $this->x = $x;
        $this->y = $y;
        $this->flyweight = $flyweight;
    }

    //Retorna um array com os dados de Arvore
    public function retornaArvore(): array
    {
        return [
            'x' => $this->x,
            'y' => $this->y,
            'especie' => $this->flyweight->retornaEspecie()
        ];
    }
}
```

É na classe **EspecieFabrica** que a economia de memória acontece. Antes de implementar vamos entender o que essa classe precisa fazer:

- Gerenciar a criação de objetos do tipo **EspecieFlyweight**.
- Promover o compartilhamento de objetos **EspecieFlyweight** entre árvores que pertençam a uma mesma espécie.

Para alcançar tais objetivos iremos criar um método importante, o **getEspecie()**. Este método será o responsável por verificar se um objeto de uma determinada espécie de árvore já existe. Em caso positivo ele irá retornar a instância de **EspecieFlyweight** já existente, economizando assim a alocação de informações redundantes em memória.

Nota: Talvez você esteja se perguntando se isso não se parece com um *singleton*. Sim, a idéia é a mesma, a diferença aqui é que um objeto *Flyweight* deve ser imutável, ou seja, depois de criado ele não deve mudar. O *Singleton* pode ser modificado depois de criado, inclusive essa é uma das principais causas de problemas que ocorrem devido a sua utilização.

Vejamos a implementação:

```
class EspecieFabrica
{
    private $especies = []; //Lista de espécies de árvores (Flyweights).

    //Retorna uma espécie de árvore já existente ou cria uma nova e a retorna.
    public function getEspecie (
        string $nome,
        string $cor,
        float $alturaMaxia
    ): EspecieFlyweight {
        //Para cada espécie de árvore já existente.
        foreach ($this->especies as $e) {
            //Compara seus atributos
            if (
                $e->getNome() == $nome &&
                $e->getCor() == $cor &&
                $e->getAlturaMaxima() == $alturaMaxia
            ) {
                //Se a espécie já existir a retorna sem criar uma nova.
                return $e;
            }
        }
        //Se não existir cria uma nova, a adiciona à lista de espécies e a retorna.
        $novaEspecie = new EspecieFlyweight($nome, $cor, $alturaMaxia);
        $this->especies[] = $novaEspecie;
        return $novaEspecie;
    }

    //Retorna um array contento todos os objetos EspecieFlyweight criados
    public function retornaTodasEspecies(): array
    {
        return $this->especies;
    }
}
```

O cliente do padrão Flyweight será a classe **Plantacao**. Ela irá gerenciar e manter a criação de objetos **Arvore**.

É importante darmos uma atenção especial ao método **addArvore()** presente na classe **Plantacao**. Este método recebe por parâmetro os valores x, y, nome, cor e altura máxima da árvore a ser criada. Os parâmetros **x e y são os estados extrínsecos** de **Arvore** e podem mudar conforme o contexto de cada objeto, no caso, a posição em que cada árvore se encontra. Já os parâmetros **nome, cor e altura máxima são estados intrínsecos** de **Arvore**, não variam entre árvores de uma mesma espécie e por isso podem ser compartilhados.

Quando o método **addArvore()** é chamado ele deve delegar a criação de uma espécie para a classe **EspecieFabrica** por meio do método **getEspecie()**

passando para ele uma chave que permite identificar uma espécie, no nosso caso a chave são os campos nome, cor e altura máxima. A partir deste ponto o método **getEspecie()** é quem vai se preocupar com a criação e retorno da instância de **EspecieFlyweight** que deve ser utilizada na criação do objeto **Arvore**.

```
class Plantacao
{
    //Contém todas as árvores da plantação
    private array $arvores;
    //Mantém referência a EspecieFabrica para solicitar as instâncias de EspecieFlyweight
    private EspecieFabrica $especieFabrica;

    public function __construct()
    {
        //Inicializa a fábrica de espécies
        $this->especieFabrica = new EspecieFabrica();
    }

    //Adiciona uma árvora a plantação
    public function addArvore(
        int $x,
        int $y,
        string $nome,
        string $cor,
        float $alturaMaxima
    ): void {
        //Solicita uma espécie (EspecieFlyweight) a EspecieFabrica
        $especie = $this->especieFabrica->getEspecie($nome, $cor, $alturaMaxima);
        //Cria a árvore
        $arvore = new Arvore($x, $y, $especie);
        //E a adiciona ao array de árvores.
        $this->arvores[] = $arvore;
    }

    //Retorna um array contendo todas as espécies criadas
    public function retornaEspecies(): array
    {
        $especies = $this->especieFabrica->retornaTodasEspecies();
        $especiesArray = [];
        foreach ($especies as $especie) {
            $especiesArray[] = $especie->retornaEspecie();
        }

        return $especiesArray;
    }

    //Retorna um array contendo todas as árvores criadas
    public function retornaArvores(): array
    {
        $arvoresArray = [];
        foreach ($this->arvores as $arvore) {
            $arvoresArray[] = $arvore->retornaArvore();
        }

        return $arvoresArray;
    }
}
```

Vamos fazer um pequeno teste:

```php
//Numero de ciclos de iterações do for
$iteracoes = 10;

//Criação da plantação de Árvores
$plantação = new Plantacao();

//Em cada iteração serão criadas 3 árvores, uma de cada espécie.
//As cordenadas x,y serão geradas de forma aleatória (serão um número de 0 à 500)
for ($i = 0; $i < $iteracoes; $i++) {
    $plantação->addArvore(rand(0, 500), rand(0, 500), 'Ipê', '#2caf1e', 10);
    $plantação->addArvore(rand(0, 500), rand(0, 500), 'Palmeira', '#008a29', 7);
    $plantação->addArvore(rand(0, 500), rand(0, 500), 'Jabuticabeira', '#00b626', 5);
}

echo "<pre>";
echo "### Espécies ###<br>";
print_r($plantação->retornaEspecies());
echo "<br>";
echo "### Árvores ###<br>";
print_r($plantação->retornaArvores());
echo "</pre>";
```

Saída (Apenas parte da saída por ser muito extensa):

```
### Espécies ###
Array
(
    [0] => Array
        (
            [nome] => Ipê
            [cor] => #2caf1e
            [alturaMaxima] => 10
        )
    [1] => Array
        (
            [nome] => Palmeira
            [cor] => #008a29
            [alturaMaxima] => 7
        )
    [2] => Array
        (
            [nome] => Jabuticabeira
            [cor] => #00b626
            [alturaMaxima] => 5
        )
)

### Árvores ###
Array
(
    [0] => Array
        (
            [x] => 418
            [y] => 491
            [especie] => Array
                (
                    [nome] => Ipê
                    [cor] => #2caf1e
                    [alturaMaxima] => 10
                )
        )
```

```
    [1] => Array
        (
            [x] => 11
            [y] => 403
            [especie] => Array
                (
                    [nome] => Palmeira
                    [cor] => #008a29
                    [alturaMaxima] => 7
                )
        )
    [2] => Array
        (
            [x] => 237
            [y] => 6
            [especie] => Array
                (
                    [nome] => Jabuticabeira
                    [cor] => #00b626
                    [alturaMaxima] => 5
                )
        )
    [3] => Array
        (
            [x] => 494
            [y] => 157
            [especie] => Array
                (
                    [nome] => Ipê
                    [cor] => #2caf1e
                    [alturaMaxima] => 10
                )
        )
    [4] => Array
        (
            [x] => 441
            [y] => 147
            [especie] => Array
                (
                    [nome] => Palmeira
                    [cor] => #008a29
                    [alturaMaxima] => 7
                )
        )
    [5] => Array
        (
            [x] => 374
            [y] => 137
            [especie] => Array
                (
                    [nome] => Jabuticabeira
                    [cor] => #00b626
                    [alturaMaxima] => 5
                )
        )

  CONTINUA ATÉ A POSIÇÃO 29...
```

Nosso teste possui apenas 10 iterações, a cada iteração 3 árvores são criadas, totalizando assim 30 árvores. Repare que temos apenas 3 instâncias da classe espécie. As informações das espécies são alocadas apenas uma vez na memória RAM, depois são apenas referenciadas pelas árvores. Deste modo existe um compartilhamento de espécies entre as árvores.

No exemplo estamos trabalhando com pequenas *strings* e um *float*. Imagine um cenário onde os atributos de espécie sejam consideravelmente mais custosos para a RAM.

- Especie - 10kB;
- Cor - 5kB;
- AlturaMaxima - 2kB;

Considere também que ao invés de 10 (dez) iterações temos 1.000.000 (um milhão). Sem o compartilhamento promovido pelo padrão *Flyweight* teríamos 3 milhões de árvores e cada uma delas alocaria os atributos de espécie na memória RAM (17kB por árvore). Por outro lado, com o compartilhamento tal redundância de recursos não existiria.

Aplicabilidade (Quando utilizar?)

A eficiência do padrão Flyweight depende muito de como e onde ele é aplicado. É recomendado que este padrão seja aplicado somente quando todas as condições abaixo forem verdadeiras:

- Uma aplicação utiliza um grande número de objetos;
- Os custos de armazenamento são altos por causa da grande quantidade de objetos;
- Muitos objetos podem compartilhar uma mesma representação e é possível separar o estado extrínseco dos mesmos.
- Muitos grupos de objetos podem ser substituídos por relativamente poucos objetos compartilhados, uma vez que seus estados extrínsecos são removidos;
- A aplicação não depende da identidade dos objetos. Uma vez que objetos Flyweight podem ser compartilhados, testes de identidade produzirão o valor verdadeiro para objetos conceitualmente distintos.

Componentes

- **Flyweight:** Declara uma interface através da qual Flyweights podem receber e atuar sobre estados extrínsecos.

- **FlyweightConcreto:** Implementa a interface Flyweight e acrescenta armazenamento para estados intrínsecos, se houver. Um objeto FlyweightConcreto deve ser compartilhável. Qualquer estado que ele armazene deve ser intrínseco, ou seja, imutável e independente do contexto que o objeto FlyweightConcreto se encontra.

- **FlyweightConcretoNaoCompartilhavel:** Nem todas as subclasses (caso existam) de Flyweight necessitam ser compartilháveis. A interface Flyweight não força ou garante tal compartilhamento. É comum que objetos FlyweightConcretoNaoCompartilhavel tenham objetos FlyweightConcreto como filhos em algum nível da estrutura.

- **FabricaFlyweight:** Além de criar e gerenciar objetos flyweight essa classe também garante que o Flyweights sejam compartilhadas de forma apropriada. Quando um cliente solicita um Flyweight, a classe FabricaFlyweight fornece uma instância existente que seja apropriada, caso tal instância ainda não exista, FabricaFlyweight cria uma instância que atenda a solicitação.

- **Cliente:** Mantêm uma referência para flyweight(s) e computa ou armazena o estado extrínseco do flyweight(s).

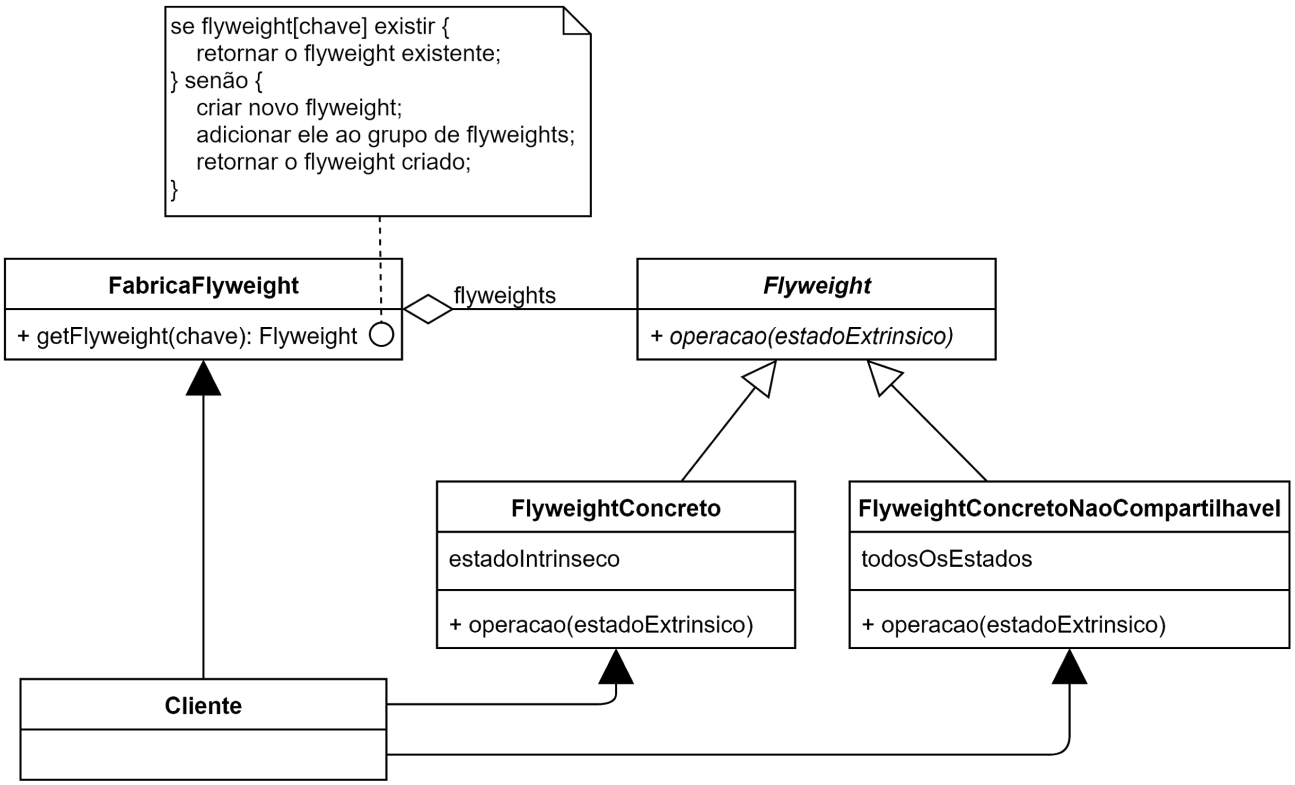


Diagrama de Classes

Consequências

Os *flyweights* podem introduzir custos de tempo de execução associados à transferência, busca e/ou processamento de estados extrínsecos, especialmente se ele era anteriormente armazenado como estado intrínseco. No entanto, esses custos são compensados pela economia de espaço, que aumenta à medida que mais flyweights são compartilhados.

A economia de armazenamento é uma função de vários fatores:

- A redução no número total de instâncias que vem do compartilhamento;
- A quantidade de estado intrínseco por objeto;
- Se o estado extrínseco é processado (produzido) ou armazenado.

Quanto mais *flyweights* forem compartilhados, maior será a economia de armazenamento. Tal economia se torna mais relevante quando os objetos usam quantidades substanciais de estados intrínsecos e extrínsecos, e os estados extrínsecos podem ser calculados ao invés de armazenados. Portanto, a economia no armazenamento se dá de duas maneiras:

- O compartilhamento reduz o custo do estados intrínsecos;
- Troca o custo de armazenamento dos estados extrínsecos pelo tempo de processamento necessário para os produzir.

Atenção: A aplicação do padrão *Flyweight* requer mais atenção ao remover objetos da memória. É importante se certificar que um objeto compartilhado não esteja sendo mais sendo referenciado por ao menos um objeto ativo antes de ser excluído.