refactoring.guru

Flyweight

8–11 minutes

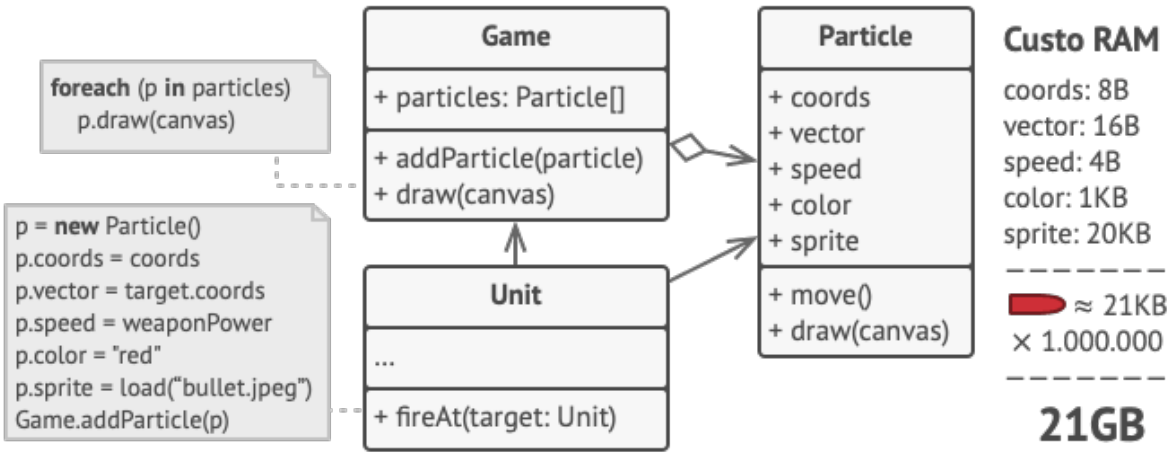
Problema

Para se divertir após longas horas de trabalho você decide criar um jogo simples: os jogadores estarão se movendo em um mapa e atirado uns aos outros. Você escolhe implementar um sistema de partículas realístico e faz dele uma funcionalidade distinta do jogo. Uma grande quantidades de balas, mísseis, e estilhaços de explosões devem voar por todo o mapa e entregar adrenalina para o jogador.

Ao completar, você sobe suas últimas mudanças, constrói o jogo e manda ele para um amigo para um test drive. Embora o jogo tenha rodado impecavelmente na sua máquina, seu amigo não foi capaz de jogar por muito tempo. No computador dele, o jogo continuava quebrando após alguns minutos de gameplay. Após algumas horas pesquisando nos registros do jogo você descobre que ele quebrou devido a uma quantidade insuficiente de RAM. Acontece que a máquina do seu amigo é muito menos poderosa que o seu computador, e é por isso que o problema apareceu facilmente na máquina dele.

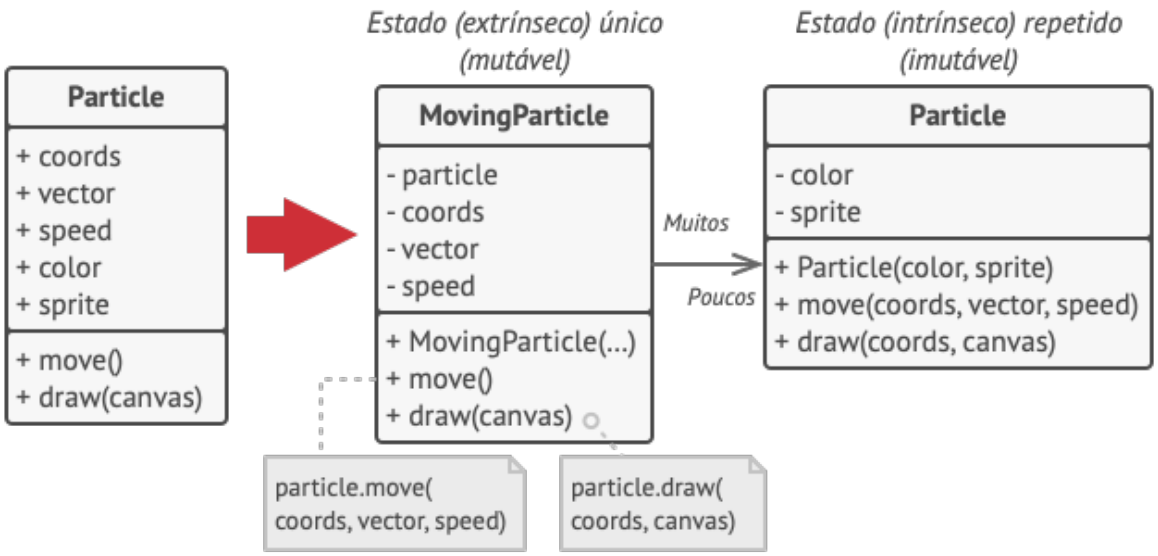
O verdadeiro problema está relacionado ao seu sistema de partículas. Cada partícula, tais como uma bala, um míssil, ou um

estilhaço era representado por um objeto separado contendo muita informação. Em algum momento, quando a destruição na tela do jogadora era tanta, as novas partículas criadas não cabiam mais no RAM restante, então o programa quebrava.



Solução

Olhando de perto a classe `Partícula`, você pode notar que a cor e o campo sprite consomem muita memória se comparado aos demais campos. E o pior é que esses dois campos armazenam dados quase idênticos para todas as partículas. Por exemplo, todas as balas têm a mesma cor e sprite.

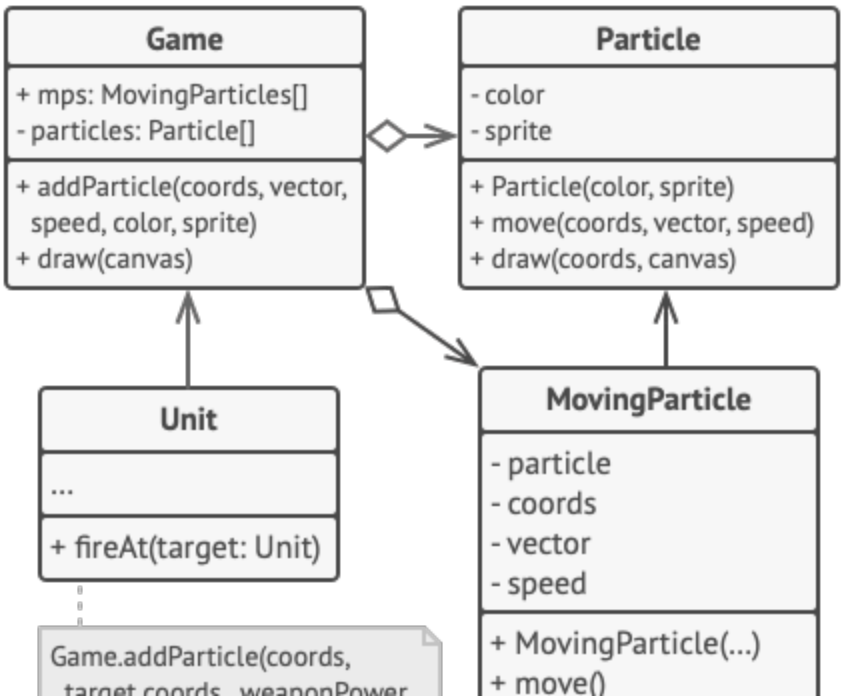


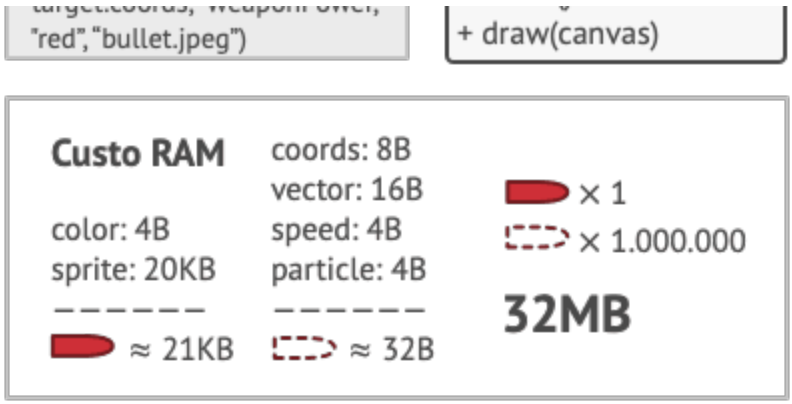
Outras partes do estado de uma partícula, tais como

coordenadas, vetor de movimento e velocidade, são únicos para cada partícula. Afinal de contas, todos os valores desses campos mudam com o tempo. Esses dados representam todo o contexto de mudança na qual a partícula existe, enquanto que a cor e o sprite permanecem constante para cada partícula.

Esse dado constante de um objeto é usualmente chamado de *estado intrínseco*. Ele vive dentro do objeto; outros objetos só podem lê-lo, não mudá-lo. O resto do estado do objeto, quase sempre alterado “pelo lado de fora” por outros objetos é chamado *estado extrínseco*.

O padrão Flyweight sugere que você pare de armazenar o estado extrínseco dentro do objeto. Ao invés disso, você deve passar esse estado para métodos específicos que dependem dele. Somente o estado intrínseco fica dentro do objeto, permitindo que você o reutilize em diferentes contextos. Como resultado, você vai precisar de menos desses objetos uma vez que eles diferem apenas em seu estado intrínseco, que tem menos variações que o extrínseco.



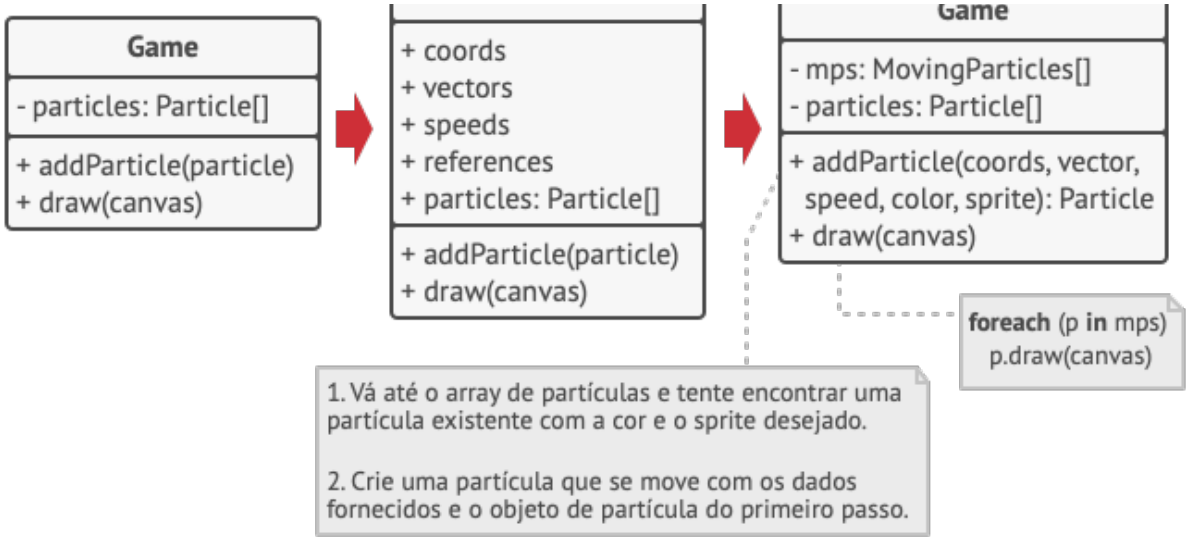


Vamos voltar ao nosso jogo. Assumindo que extraímos o estado extrínseco de nossa classe de partículas, somente três diferentes objetos serão suficientes para representar todas as partículas no jogo: uma bala, um míssil, e um pedaço de estilhaço. Como você provavelmente já adivinhou, um objeto que apenas armazena o estado intrínseco é chamado de um flyweight.

Armazenamento do estado extrínseco

Para onde vai o estado extrínseco então? Algumas classes ainda devem ser capazes de armazená-lo, certo? Na maioria dos casos, ele é movido para o objeto contêiner, que agrega os objetos antes de aplicarmos o padrão.

No nosso caso, esse seria o objeto principal `Jogo` que armazena todas as partículas no campo `partículas`. Para mover o estado extrínseco para essa classe você precisa criar diversos campos array para armazenar coordenadas, vetores, e a velocidade de cada partícula individual. Mas isso não é tudo. Você vai precisar de outra array para armazenar referências ao flyweight específico que representa a partícula. Essas arrays devem estar em sincronia para que você possa acessar todos os dados de uma partícula usando o mesmo índice.



Uma solução mais elegante é criar uma classe de contexto separada que armazenaria o estado extrínseco junto com a referência para o objeto flyweight. Essa abordagem precisaria apenas de uma única array na classe contêiner.

Calma aí! Não vamos precisar de tantos objetos contextuais como tínhamos no começo? Tecnicamente, sim, mas nesse caso, esses objetos são muito menores que antes. Os campos mais pesados foram movidos para alguns poucos objetos flyweight. Agora, milhares de objetos contextuais podem reutilizar um único objeto flyweight pesado ao invés de armazenar milhares de cópias de seus dados.

Flyweight e a imutabilidade

Já que o mesmo objeto flyweight pode ser usado em diferentes contextos, você tem que certificar-se que seu estado não pode ser modificado. Um flyweight deve inicializar seu estado apenas uma vez, através dos parâmetros do construtor. Ele não deve expor qualquer setter ou campos públicos para outros objetos.

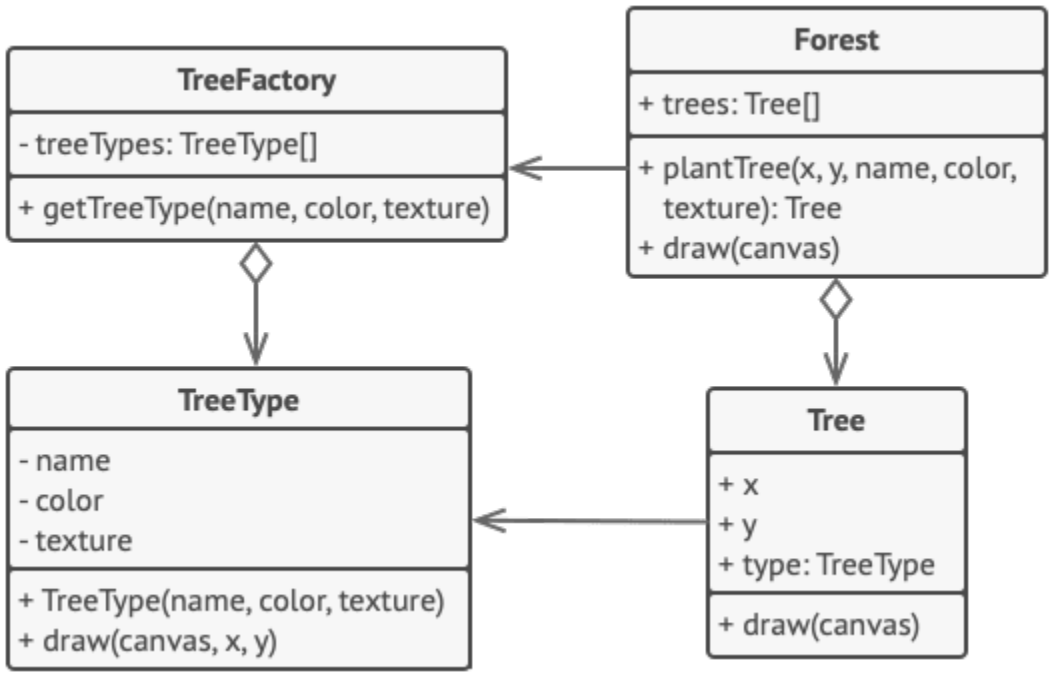
Fábrica Flyweight

Para um acesso mais conveniente para vários flyweights, você pode criar um método fábrica que gerencia um conjunto de objetos flyweight existentes. O método aceita o estado intrínseco do flyweight desejado por um cliente, procura por um objeto flyweight existente que coincide com esse estado, e retorna ele se for encontrado. Se não for, ele cria um novo flyweight e o adiciona ao conjunto.

Há várias opções onde esse método pode ser colocado. O lugar mais óbvio é um contêiner de flyweights. Alternativamente você pode criar uma nova classe fábrica. Ou você pode fazer o método fábrica ser estático e colocá-lo dentro da própria classe do flyweight.

Pseudocódigo

Neste exemplo o padrão **Flyweight** ajuda a reduzir o uso de memória quando renderizando milhões de objetos árvore em uma tela.



O padrão extrai o estado intrínseco repetido para uma classe `Árvore` principal e o move para dentro da classe flyweight `TipoÁrvore`.

Agora ao invés de armazenar os mesmos dados em múltiplos objetos, ele armazena apenas alguns objetos flyweight e os liga aos objetos `Árvore` apropriados que agem como contexto. O código cliente cria novos objetos árvore usando a fábrica flyweight, que encapsula a complexidade de busca pelo objeto correto e o reutiliza se necessário.

```
// A classe flyweight contém uma parte do estado de uma árvore
// Esses campos armazenam valores que são únicos para cada
// árvore em particular. Por exemplo, você não vai encontrar
// coordenadas da árvore aqui. Já que esses dados geralmente
são
// GRANDES, você gastaria muita memória mantendo-os em cada
// objeto árvore. Ao invés disso, nós podemos extrair a textura,
// cor e outros dados repetitivos em um objeto separado os quais
// muitas árvores individuais podem referenciar.
class TreeType is
    field name
    field color
    field texture
    constructor TreeType(name, color, texture) { ... }
    method draw(canvas, x, y) is
        // 1. Cria um bitmap de certo tipo, cor e textura.
        // 2. Desenha o bitmap em uma tela nas coordenadas X e
        // Y.


// A fábrica flyweight decide se reutiliza o flyweight existente
```

```
// ou cria um novo objeto.
class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type

// O objeto contextual contém a parte extrínseca do estado da
// árvore. Uma aplicação pode criar bilhões desses estados, já
// que são muito pequenos:
// apenas dois números inteiros para coordenadas e um campo de
// referência.
class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)

// As classes Tree (Árvore) e Forest (Floresta) são os clientes
// flyweight. Você pode uni-las se não planeja desenvolver mais
// a classe Tree.
class Forest is
    field trees: collection of Trees

    method plantTree(x, y, name, color, texture) is
        type = TreeFactory.getTreeType(name, color, texture)
```

```
    tree = new Tree(x, y, type)
    trees.add(tree)

method draw(canvas) is
    foreach (tree in trees) do
        tree.draw(canvas)
```