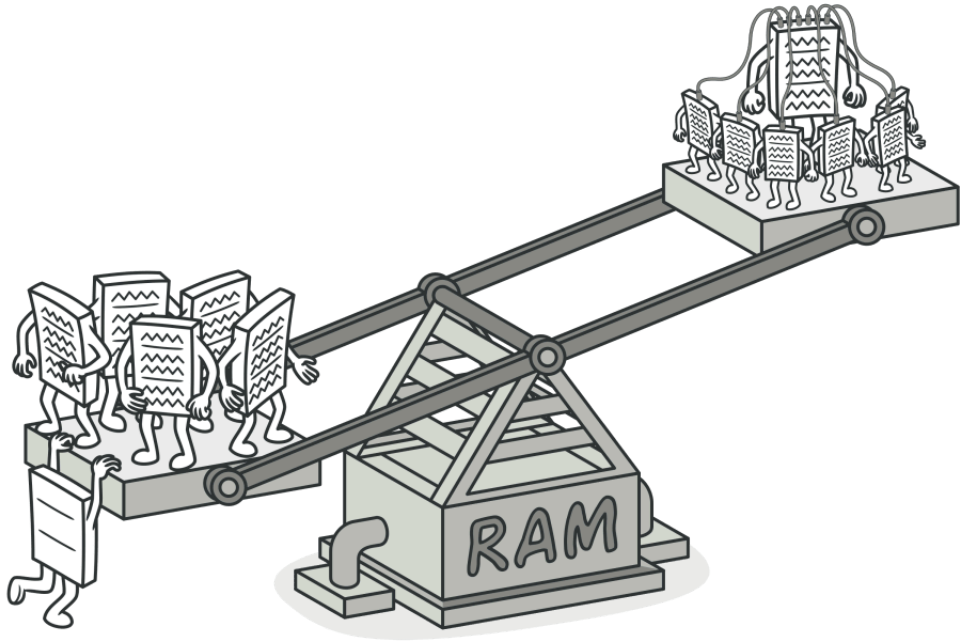
TÓPICO 15 - FLYWEIGHT

Design Patterns - Professor Ramon Venson - SATC 2026.1

Motivação

Quando uma aplicação precisa manter milhares de objetos semelhantes na memória, repetir o mesmo estado em cada instância custa caro.

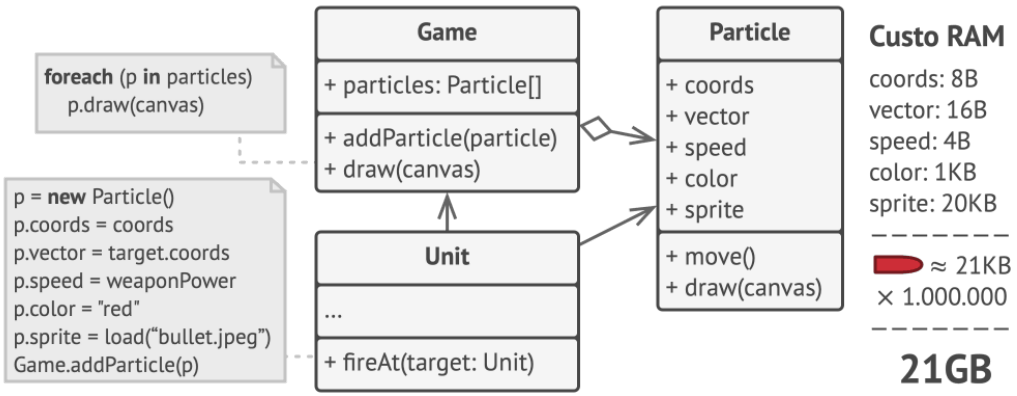
O padrão Flyweight resolve isso ao **compartilhar a parte comum** entre vários objetos.



Problema

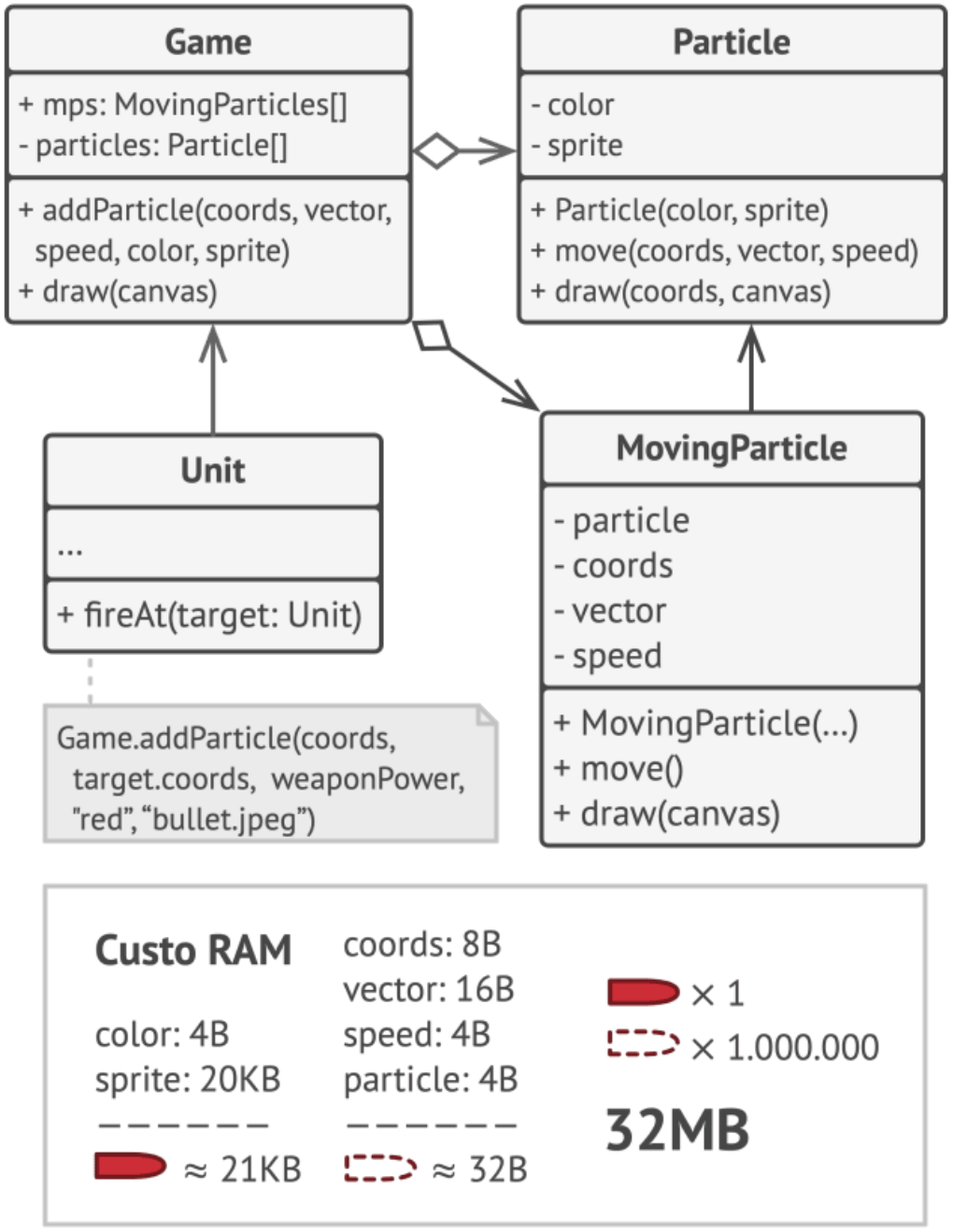
Imagine um mapa interativo do campus com milhares de marcadores de salas, laboratórios e bibliotecas.

Se cada marcador carregar cópia própria de ícone, cor e configuração visual, o consumo de memória cresce sem necessidade.



Conceito principal

- **Flyweight** é um padrão estrutural
- Ele compartilha estado comum entre muitos objetos
- O estado comum é chamado de **intrínseco**
- O estado contextual é chamado de **extrínseco**
- O ganho vem de reutilizar poucos objetos pesados

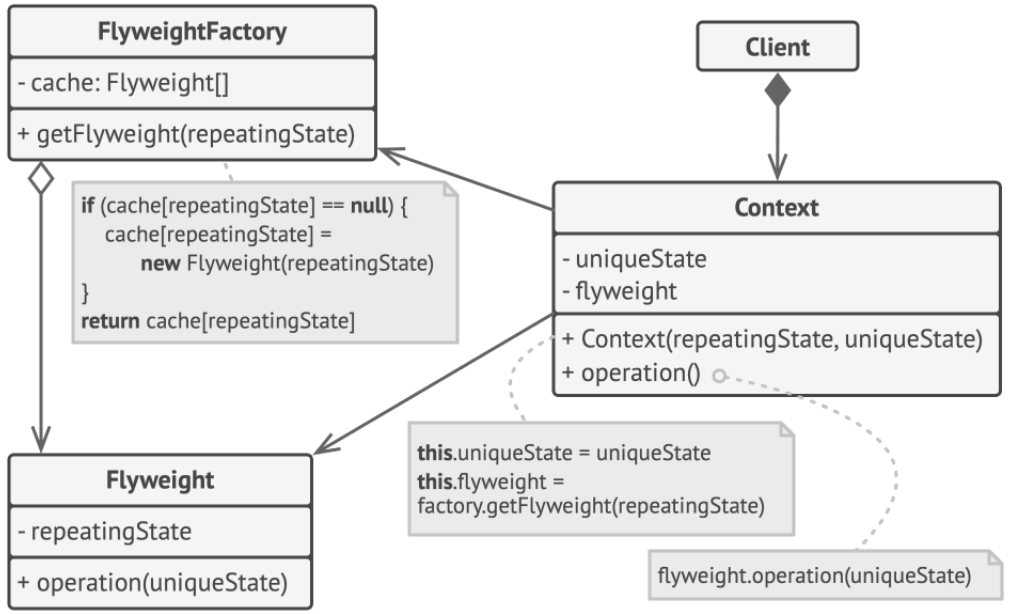


Como funciona

- O cliente pede um objeto compartilhado para uma fábrica
- A fábrica reutiliza um flyweight existente quando possível
- Cada ocorrência guarda apenas o contexto que varia
- O contexto é passado ao flyweight no momento da operação
- O flyweight precisa ser seguro para compartilhamento

Estrutura / Componentes

- **Flyweight** guarda estado intrínseco
- **Contexto** guarda estado extrínseco
- **Fábrica** reutiliza objetos por chave
- **Cliente** combina contexto com flyweight
- O conjunto final representa o objeto completo em execução



Exemplo conceitual

- Sistema exibe milhares de locais no mapa do campus
- Todas as salas compartilham o mesmo estilo visual
- Todos os laboratórios compartilham outro estilo
- Posição, nome e disponibilidade mudam por marcador
- Poucos flyweights atendem muitas ocorrencias

Código: flyweight e fábrica

```
class TipoLocalFlyweight {
    private final String categoria;
    private final String cor;
    private final String icone;
}

class TipoLocalFactory {
    private static final Map<String, TipoLocalFlyweight> tipos = new HashMap<>();

    public static TipoLocalFlyweight obter(String categoria, String cor, String icone) {
        String chave = categoria + ":" + cor + ":" + icone;
        tipos.putIfAbsent(chave, new TipoLocalFlyweight(categoria, cor, icone));
        return tipos.get(chave);
    }
}
```

Código: contexto

```
class LocalMapa {
    private final int x;
    private final int y;
    private final String nome;
    private final TipoLocalFlyweight tipo;

    public void renderizar() {
        tipo.desenhar(x, y, nome);
    }
}
```

Quando usar

- Há muitos objetos semelhantes na memória
- O estado repetido pode ser compartilhado
- O custo de RAM já é um problema real
- O estado compartilhado pode ser imutável
- O time aceita a complexidade extra em troca de otimização

Quando não usar

- O número de objetos é pequeno
- O ganho de memória seria irrelevante
- O estado não pode ser separado com clareza
- O compartilhamento exigiria muita gambiarra
- A equipe está fazendo otimização prematura

Vantagens

- Reduz bastante o uso de memória
- Evita duplicacao de dados iguais
- Reutiliza recursos pesados com eficiencia
- Torna explícito o que é comum e o que é contextual
- Funciona bem com caches e fábricas

Desvantagens

- Aumenta a complexidade do design
- Exige disciplina com imutabilidade
- Pode dificultar entendimento inicial do código
- Introduz coordenação entre contexto e flyweight
- Nem sempre vale a pena em sistemas pequenos