

Factory

Fábrica de objetos

Padrão de projeto criacional

Problema

- Como criar uma instância de uma classe sem gerar uma dependência (**acoplamento**) entre a classe cliente e a classe alvo?
- Como lidar com mudanças da hierarquia de classes?

Padrão de projeto: Factory

- Classificação: **criacional**
- Um dos padrões mais usados.
- Trata dos detalhes de criação de objetos.
- O Factory é um padrão que **encapsula a operação de criação de objetos**, fornecendo uma interface que retorna uma instância para as classes cliente.
- Vantagem de uso: permite a **criação de objetos** **sem** especificar a classe concreta.

Solução

- O factory é a classe responsável por criar objetos de um tipo.
- Sempre que uma nova classe é implementada, o único local que precisa ser atualizado é a Factory.
- Assim, a classe cliente pede ao factory por uma instancia.

Operador **new**

- No java o operador **new** permite criar uma instância (objeto)
- Essa instância é gerada a partir de uma classe concreta
- Entretanto, nem sempre sabemos qual será a classe!
- Delegar ao factory a criação de novas instâncias:
 - flexibilidade caso a classe concreta seja modificada
 - controlar, limitar e rastrear a criação de novos objetos

Exemplo: API Java

O método **getInstance** no JDK é um exemplo comum do uso de fábricas para obter uma instância, liberando o cliente da necessidade de gerenciar a criação de objetos.

```
// Create a calendar object
Calendar cal = Calendar.getInstance();
System.out.println("Date and time is: " + cal.getTime());

// create a currency object based on the user "Country"
Locale lc = Locale.CANADA;
Currency moeda = Currency.getInstance(lc);
System.out.println("Moeda local: " + moeda.getDisplayName());
```

Código "vulnerável" a mudanças

Cenário problemático:

- (1) código faz uso de várias classes concretas.
- (2) código terá que ser re-escrito a medida que **novas classes** são adicionadas

Em outras palavras esse código não está finalizado (continua aberto a modificações)

```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```

Identificar aspectos que mudam no código

- Princípio: Identificar partes do código que "evoluem" e separar das partes que se mantêm as mesmas.
- Encapsular as partes do código que instanciam classes (sujeitas a modificações) separando elas do resto da aplicação.

```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```


Identificar aspectos que mudam no código

O código com switch pode ser refatorado:

```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```



```
SaveImageStrategy strategy =  
    factory.createStrategy(formato);
```

Exemplo: jogo NPC

- Controlar a criação de inimigos/NPC num jogo. Novos inimigos podem ser criados dinamicamente.
- Nesse cenário uma Factory será responsável por criar instâncias de classes concretas de inimigos, com base no contexto do jogo.
- Apenas a Factory precisa saber quais são os tipos de inimigos.

Atividade 1

a) Implementar o método Ataque da **Classe NPC**

- Dragon, força de ataque=15
- Esqueleto, força ataque=10

b) Implementar o Factory da **Classe Player:**

- Mario, hp=100, defesa=8
- Joker, hp=120, defesa=5

c) Gerar 2 inimigos usando o NPCFactory, para atacar 1 player. Exibir o HP do player.