

Aula 3: Padrões GoF – Parte 2

Apresentação

Nesta aula, ilustraremos os últimos dois padrões de criação do GOF: Prototype e Singleton. Além disso, iniciaremos o estudo dos padrões estruturais pelo Bridge.

Os padrões estruturais são utilizados em projetos que possuem muitas classes e objetos inter-relacionados, formando estruturas maiores. Descreveremos nesta aula seu funcionamento, além de suas principais características.

Objetivos

- Descrever as características do Prototype;
- Identificar os tópicos do Builder;
- Classificar as propriedades do Bridge.

Primeiras palavras

Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações, enquanto os de objeto, em vez de compô-las, descrevem maneiras de se fazer a composição de objetos para obter novas funcionalidades. Um padrão consegue proporcionar flexibilidade na composição deles por meio de sua capacidade de alterar dinamicamente a composição dos objetos em tempo de execução, o que não é possível com uma composição estática (herança de classes).

Padrões estruturais possibilitam o desacoplamento entre a interface e a implementação de objetos.

Padrão	Escopo Classe	Escopo Objeto
Adapter	X	X
Bridge		X
Composite		X
Decorator		X

Flyweight	X
Proxy	X

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Prototype

Objetivos

“Este padrão deve ser utilizado quando desejarmos especificar os tipos de objetos a serem criados usando instâncias protótipos para que possamos criar objetos simplesmente copiando o protótipo, ocultando os procedimentos necessários para criar essas instâncias.”

- GAMMA *et al.*, 1994

Este padrão será necessário nas seguintes situações:

- Para tornar um sistema independente de como seus objetos são criados, compostos e representados;
- A fim de adicionar e remover objetos em tempo de execução;
- Quando quisermos especificar novos objetos ao mudar uma estrutura de objetos existentes;
- Para facilitar a configuração de uma aplicação com classes em tempo de execução;
- A fim de minimizar o número de classes de um sistema.

Problema

Quando a iniciação de um objeto for trabalhosa e precisarmos fazer algumas pequenas variações ao iniciá-lo, precisaremos de um padrão que economize tempo e recursos, evitando o retrabalho na criação de novos objetos.

Estrutura

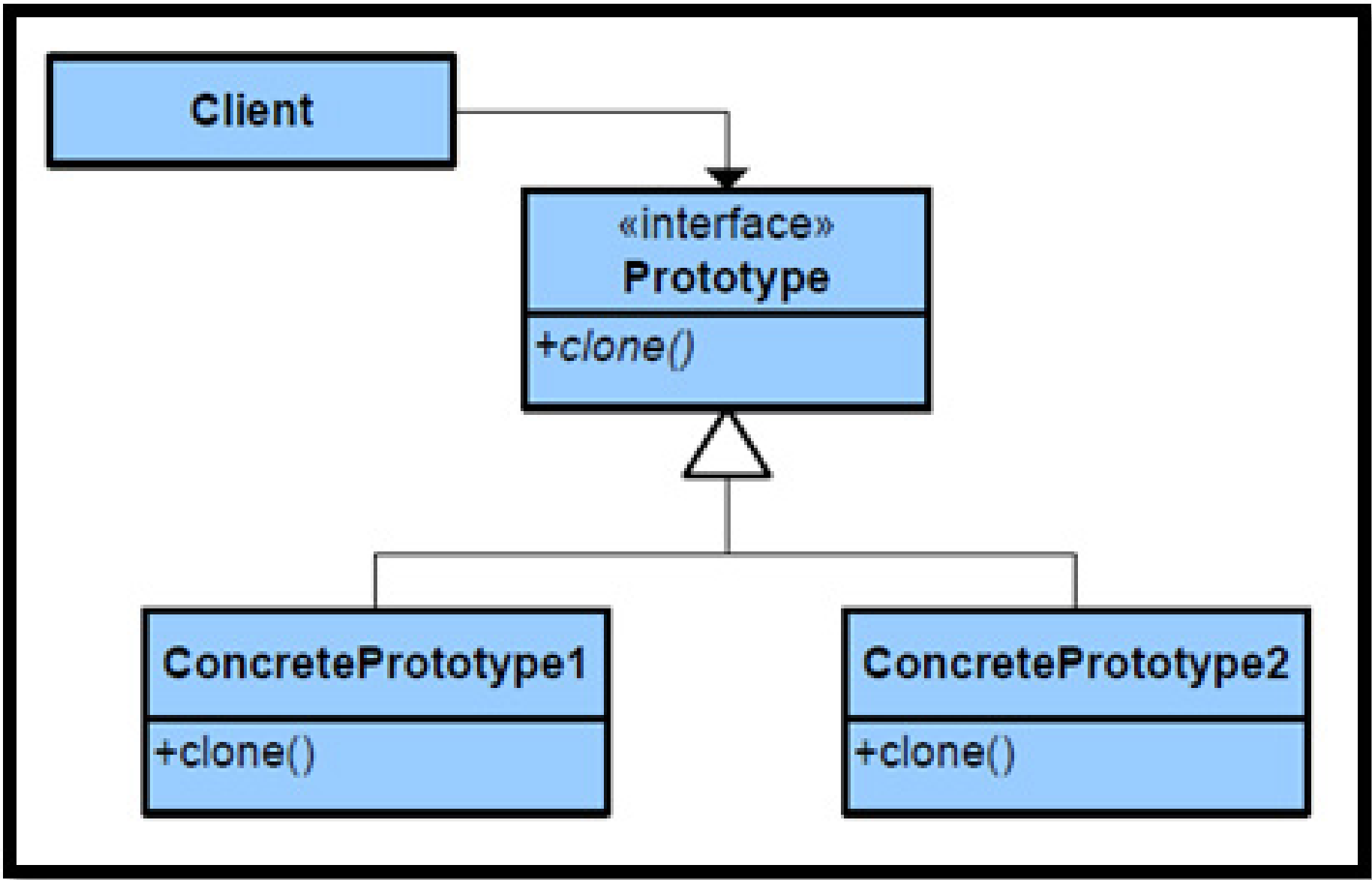
Os passos para a criação de um padrão Prototype são os seguintes:

- Defina um método **clone()** em uma hierarquia de herança existente: coloque a interface na classe base e a implementação nas classes derivadas;
- Cada classe derivada deve encapsular o uso do operador **new()** e retornar uma instância dela mesma;
- O cliente delegará a criação do clone do objeto para a classe Prototype sempre que a instância de uma hierarquia for necessária.

Diagrama UML do padrão

As classes que participam do padrão Prototype estão apresentadas a seguir:

- **Prototype:** Declara uma interface para clonar a si própria;
- **Concrete Prototype:** Implementa uma operação para clonar a si mesma;
- **Client:** Cria um objeto solicitando a um protótipo que clone a si mesmo.



 (Fonte: GAMMA et al., 1994)

Dicas

Às vezes, padrões de criação são concorrentes; em outros momentos, eles são complementares: Abstract Factory, por exemplo, pode armazenar um conjunto de protótipos a partir dos quais irá clonar e devolver objetos de produto. Os padrões Abstract Factory, Builder e Prototype podem utilizar o Singleton em suas implementações.

As classes Abstract Factory são frequentemente implementadas com Factory Methods (criados por conta da herança), mas podem sê-lo por meio do Prototype (sua criação é por meio de delegação).

Factory Method requer subclasses, mas não solicita a operação de iniciação. Prototype, ao contrário, não as requer, e sim essa operação. Ele é único entre os padrões de criação, pois tampouco requer uma classe, mas apenas um objeto.

Exemplo

Object.clone() é um ótimo exemplo do uso do padrão Prototype em Java:

```
Circulo c = newCirculo(4, 5, 6);

Circulo copia = (Circulo) c.clone();
```



Command



Iterator



Mediator

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Singleton

Objetivos

Na Matemática, conjuntos que possuem apenas um elemento são chamados de *singleton*. A ideia principal deste padrão é garantir a existência de apenas uma instância de uma classe no sistema inteiro em que todos os seus objetos serão acessados globalmente por todos os outros do sistema que precisarem se relacionar com eles: “Usamos Singleton quando o sistema que está sendo desenvolvido deve ser independente de como seus objetos são criados, compostos e representados, de forma tal que possamos adicionar e remover objetos durante a execução do sistema”. (GAMMA *et al.*, 1994)

Dica

Pode ser necessário também mudar objetos existentes ao adicionar-lhes novas características, fazendo essas configurações dinamicamente e, mesmo assim, mantendo o mínimo de classes possíveis no sistema

Problema

Como garantir que existirá apenas um objeto no sistema independentemente de quantas requisições a classe receber para criá-lo? Em quais aplicações este padrão pode ser utilizado?

As respostas estão a seguir:

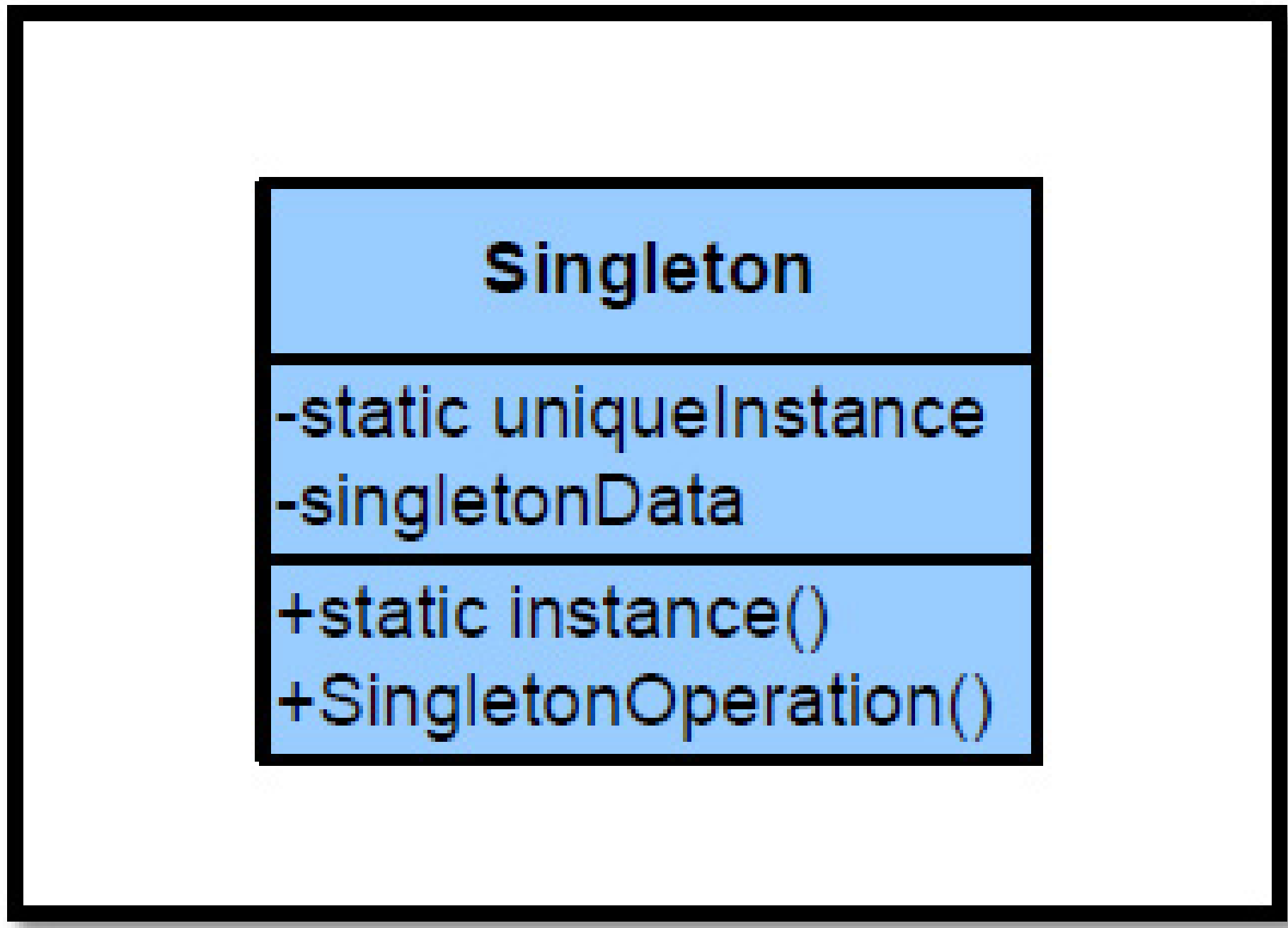
- Um único banco de dados;
- Um único acesso ao arquivo de log;
- Um único objeto que representa um vídeo;
- Uma única fachada (*façade pattern*).

Estrutura

Faça a classe Singleton ser a única responsável pelo acesso e inicialização no primeiro uso. A instância única (`the_instance`) é um atributo estático privado. Já a função que será acessada (`static instance()`) se trata de um método estático público.

- Defina um atributo `private static` na classe `single instance`;
- Defina uma função `public static` na classe;
- Faça uma iniciação tardia (criação no primeiro uso) na função pública da classe;
- Defina todos os constructors utilizando os tipos `protected` ou `private`;
- Clientes devem utilizar somente a função pública para manipular o Singleton.

Diagrama UML do padrão



Dicas

Abstract Factory, Builder e Prototype podem usar o padrão Singleton em sua implementação. A vantagem do seu uso em relação às variáveis globais é que você terá absoluta certeza do número de instâncias quando o usar, facilitando, assim, o gerenciamento de qualquer número de instâncias.

Quando o padrão Singleton é desnecessário? Há duas respostas:

- **Curta:** na maior parte do tempo;
- **Longa:** quando for mais simples passar um recurso de objeto como uma referência aos objetos que precisam em vez de permitir-lhes o acesso ao recurso globalmente.

Comentário

O problema real com Singletons é que eles lhe dão uma boa desculpa para não pensar cuidadosamente sobre a visibilidade adequada de um objeto. Encontrar o equilíbrio certo de exposição e proteção para ele é fundamental a fim de se manter a flexibilidade do sistema.

Exemplo

No caso de sistemas que precisam fazer controle de acesso utilizando um registro (log) de dados, podemos usar uma classe no padrão Singleton. Dessa forma, existirá apenas um objeto responsável pelo log em qualquer aplicação acessível unicamente por esta classe.

A seguir, apresentaremos um código na linguagem C **antes** e **depois** do uso do Singleton. Observe a vantagem do uso do padrão quando acessarmos a variável global. As linhas marcadas em negrito não são mais necessárias após o uso do padrão, deixando o programa mais eficiente:

Antes

Uma variável global é criada - quando ela for declarada –, mas não será iniciada até ser utilizada pela primeira vez no programa. Isso requer que o código da iniciação da variável seja replicado por toda o programa.

```
class GlobalClass

{
    int m_value;
    public:
    { m_value = v; }
    int get_value()
    { return m_value; }
    void set_value( int v )
    { m_value = v; }
};

// Iniciação padrão
GlobalClass* global_ptr = 0;
void funcao_1( void )
{
    // Iniciação no primeiro uso
if ( ! global_ptr )
    global_ptr = new
    GlobalClass;
    global_ptr->set_value(1);
    cout << "funcao1:
    global_ptr é "
    << global_ptr->get_value() >> '\n';
};
void funcao_2( void ) {
if ( ! global_ptr )
    global_ptr = new
    GlobalClass;
    cout << "Início: global_ptr é "
    << global_ptr->get_value() << '\n';
    funcao_1();
    funcao_2();
}
// Saída do programa:
// Início: global_ptr é 0
// Funcao1: global_ptr é 1
// Funcao2: global_ptr é 2
```

Depois

O Singleton faz a classe ser responsável por seu ponteiro global e pela iniciação no primeiro uso, utilizando um ponteiro público estático e um método público estático, que é usado pelo cliente.

```
class GlobalClass

{
    int m_value;
    static GlobalClass* s_instance;
    GlobalClass( int v=0 )
    { m_value = v; }
    public: int get_value()
    { return m_value; }
    void set_value( int v )
    { m_value = v; }
    static GlobalClass* instance(){
    if ( ! s_instance )
        s_instance = new GlobalClass;
    return s_instance;
    }
};

/* Alocando e iniciando o atributo estático global
GlobalClass. O ponteiro está sendo alocado – não
o objeto em si. */
GlobalClass;
GlobalClass::s_instance = 0;
void funcao_1( void ) {
    GlobalClass::instance()->set_value(1);
    cout << "Funcao 1:
    global_ptr é "
    <<
    GlobalClass::instance()->
    get_value() << '\n';
}
void funcao_2( void ) {
    GlobalClass::instance()->set_value(2); cout <<
    "Funcao 2: global_ptr é "
    <<
    GlobalClass::instance()-> get_value() << '\n';
}
int main( void ) {
    cout << "Início: global_ptr é "
    <<
    GlobalClass::instance()-> get_value() << '\n';
    funcao_1();
    funcao_2();
}
// Saída do programa:
// Início: global_ptr é 0
```

```
// Funcao1: global_ptr é 1
// Funcao2: global_ptr é 2
```

Padrões relacionados:



Abstract Factory



Flyweight



Template

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Objetivos

“Este padrão é utilizado quando queremos separar uma abstração de sua implementação de modo que as duas possam variar independentemente. Esse padrão pode usar encapsulamento, agregação e herança para separar responsabilidades em classes diferentes.”

- GAMMA *et al.*, 1994

Problema

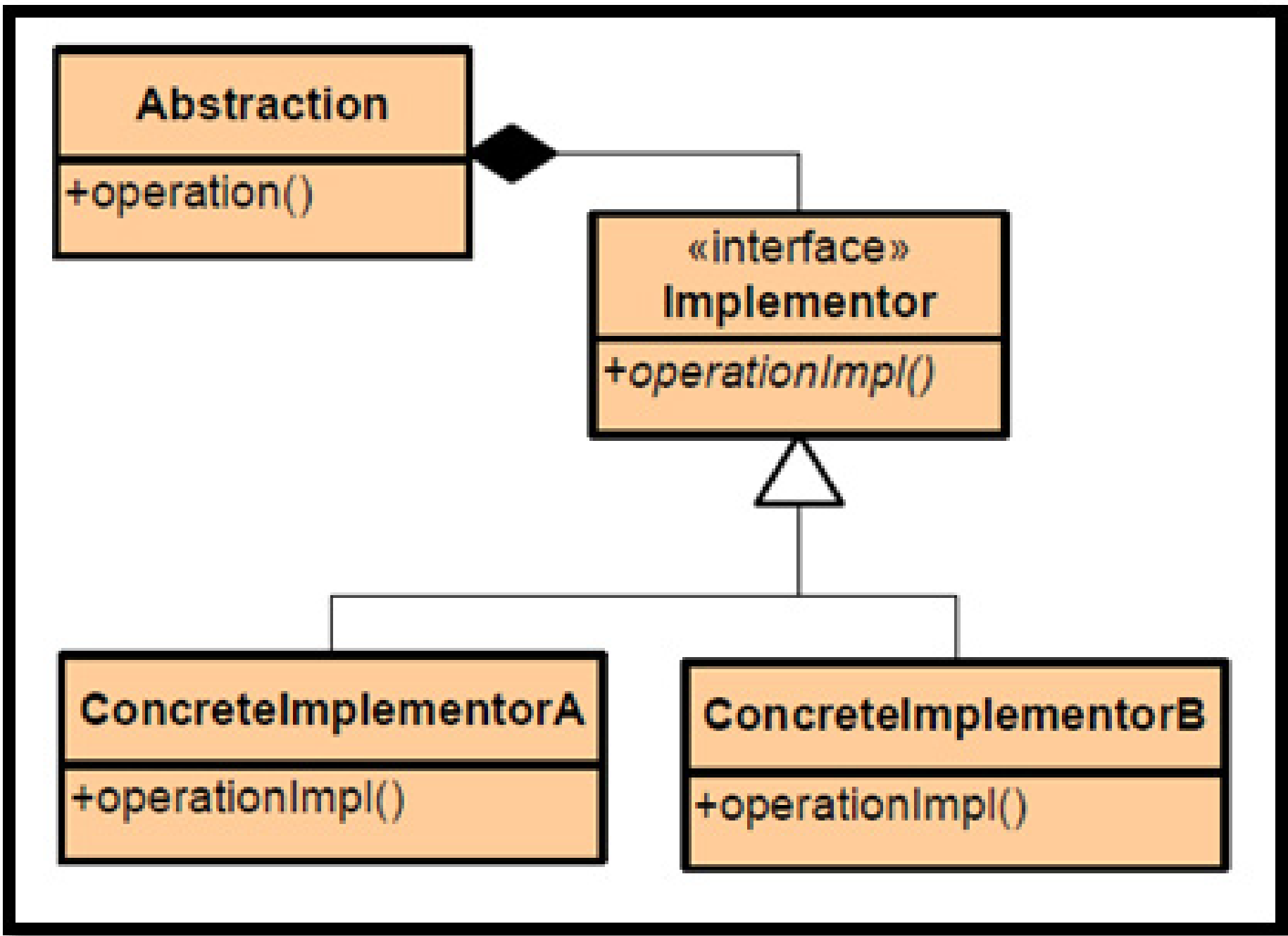
O engessamento dos relacionamentos entre as classes de um sistema ocorrerá quando utilizarmos a subclasse de uma classe base abstrata para fornecer implementações alternativas. Isso bloqueia a ligação em tempo de compilação entre a interface e a implementação. A abstração e a implementação não podem ser estendidas ou compostas independentemente.

Estrutura

- Divida um componente em abstração e implementação;
- Agrupe toda a implementação possível em uma hierarquia de herança: coloque cada implementação possível em uma classe derivada e defina uma interface em uma classe abstrata derivada que torne as instâncias destas classes intercambiáveis;
- Defina uma hierarquia de herança para os componentes da classe abstrata: use especialização para as classes derivadas e generalização na classe base abstrata;
- Defina a hierarquia de classes abstratas como um invólucro (*wrapper*) para a hierarquia de classes de implementação;
- O cliente poderá selecionar o objeto encapsulado (*wrapper*) que necessite configurar ou modificar;
- Quando o cliente fizer a requisição, o objeto *wrapper* delegará o pedido para o objeto de implementação.

Diagrama UML do padrão

- **Abstraction (classe abstrata):** Define que a interface abstrata mantém a referência para a classe de implementação;
- **Implementor (interface):** Define a interface para implementação da classe;
- **ConcretImplementor (classe normal):** Implementa a interface definida em Implementor.



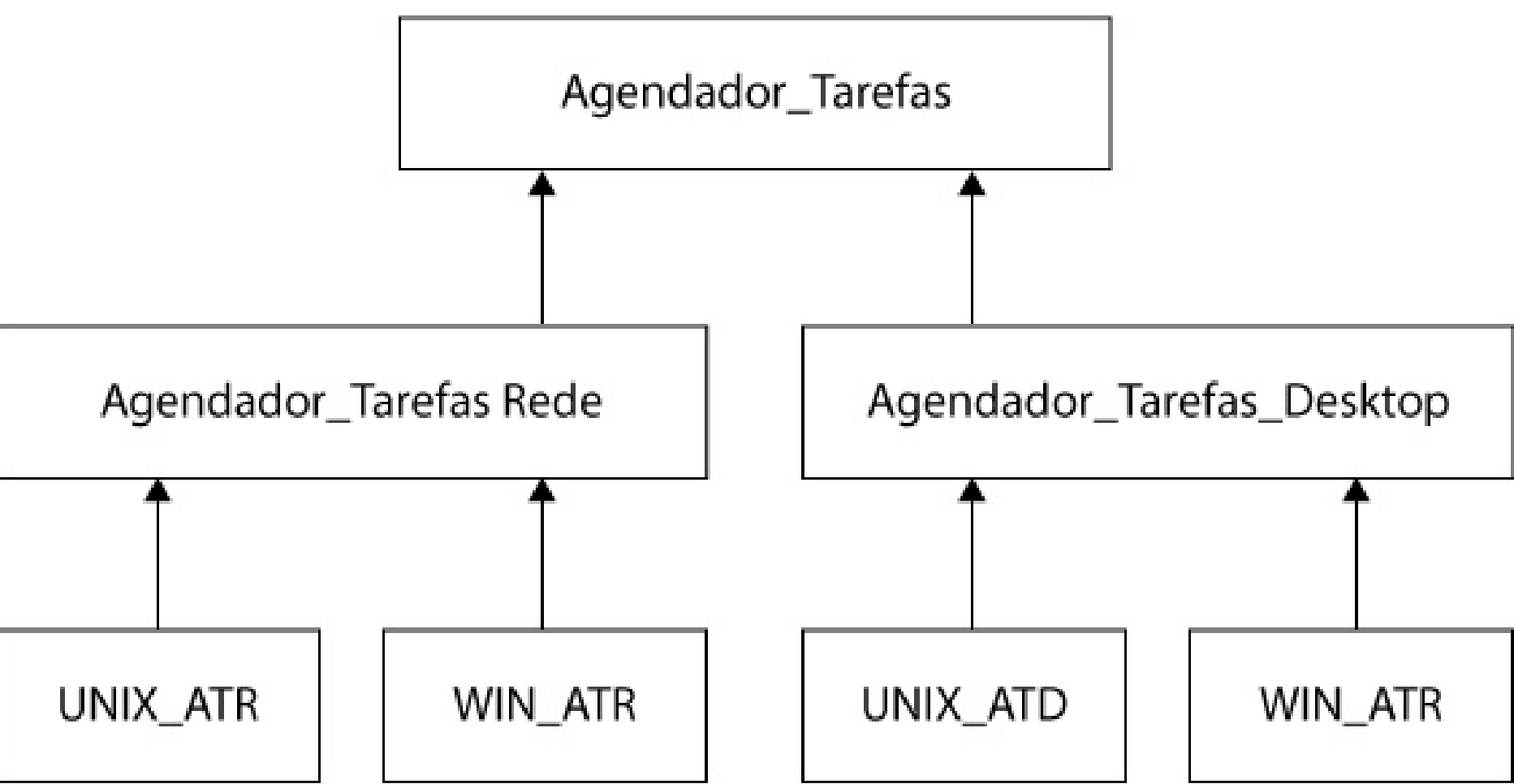
Dicas

Os padrões Adapter, Facade, Composite e Bridge atuam principalmente na adaptação de interfaces. O Adapter faz as coisas funcionarem depois de serem projetadas; o Bridge, com que funcionem antes disso.

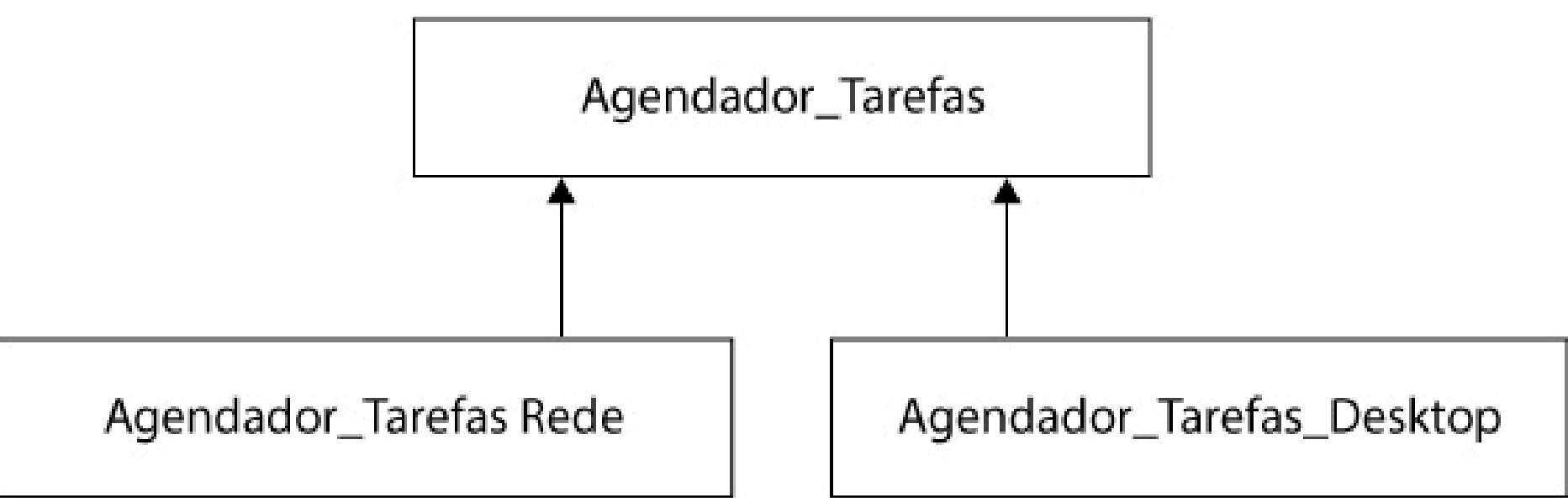
A estrutura de State e Bridge é idêntica, embora este admita hierarquias de múltiplas classes e aquele, apenas uma.

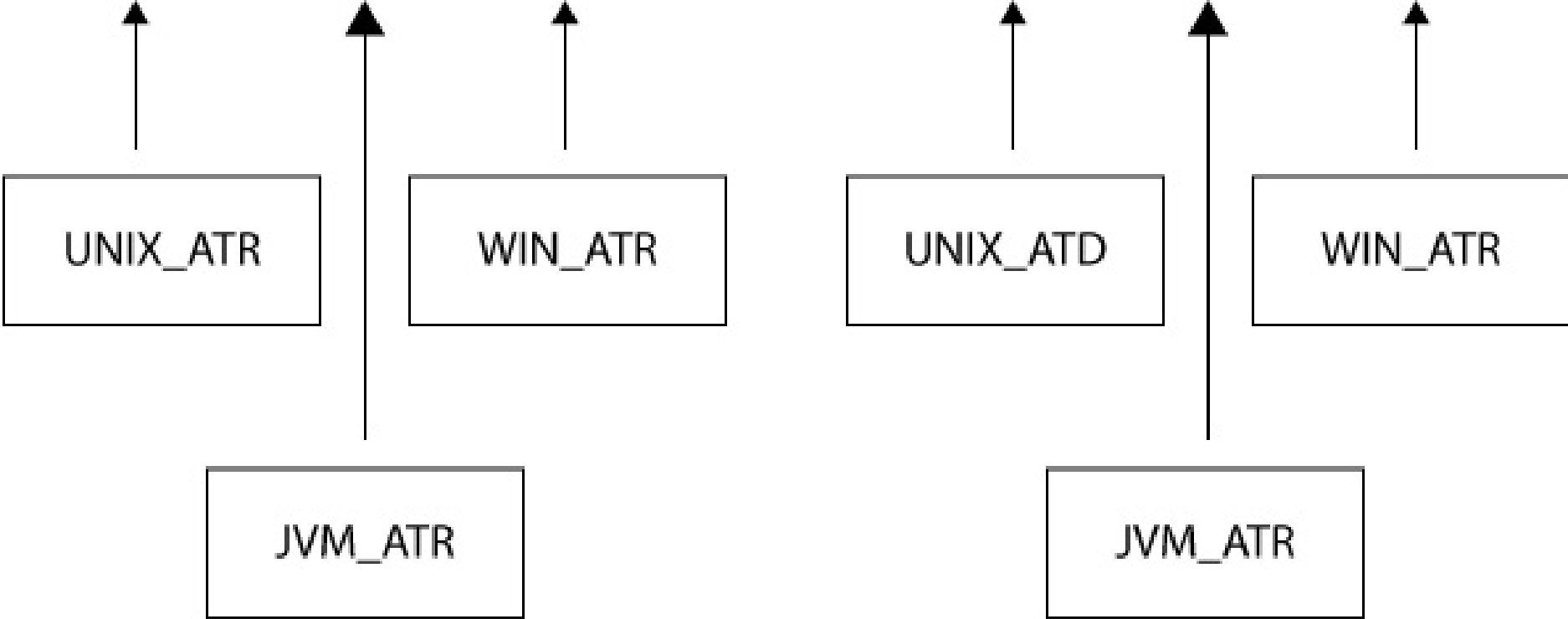
Exemplo

Considere um sistema de agendamento de tarefas que deva agendá-las em sistemas operacionais diferentes, como Linux e Windows. Temos de definir uma classe para cada permutação dessas duas dimensões conforme demonstra o diagrama de classes a seguir:




Se adicionarmos uma nova plataforma (por exemplo, Java Virtual Machine), nossa hierarquia ficará desta forma:





Surgem outras possibilidades:

- Se tivéssemos três tipos de agendadores de tarefas (threads) e quatro tipos de plataformas?
- Se tivéssemos cinco tipos de agendadores e dez tipos de plataformas?

 Clique nos botões para ver as informações.

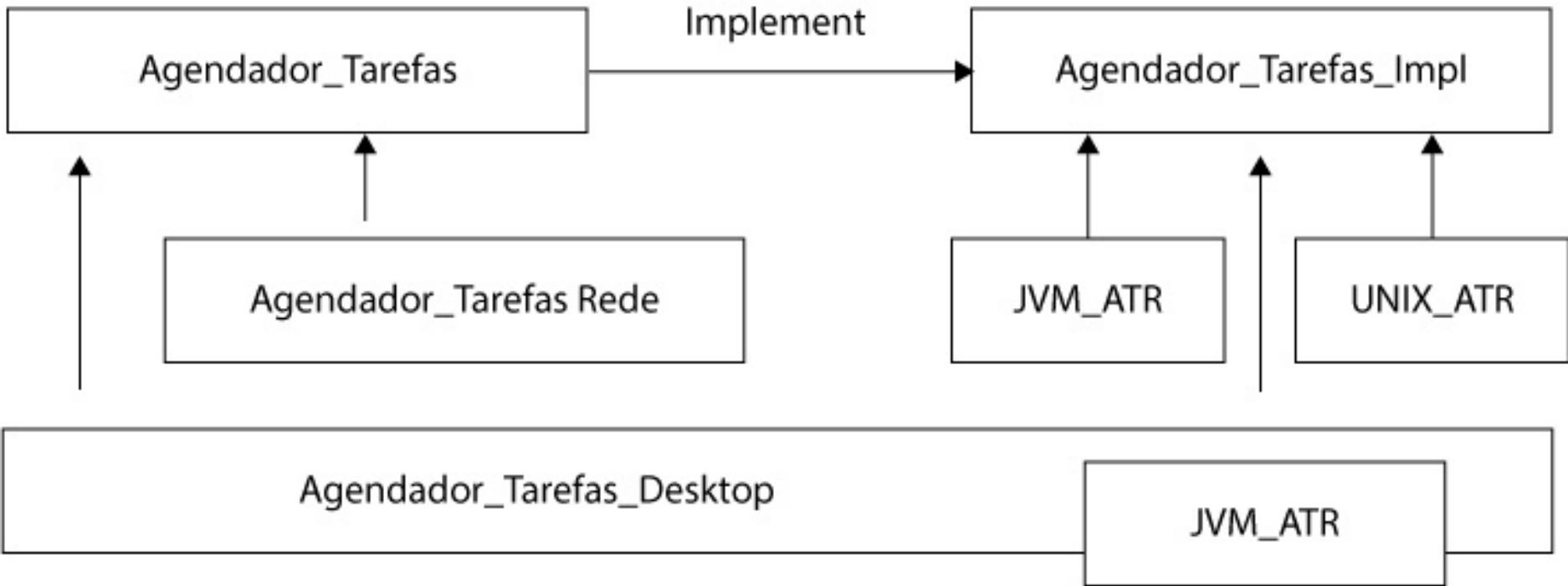
Resposta



O número de classes que teríamos de definir seria o produto (5 x10) do número de tipos de agendadores e a quantidade de plataformas.

Apresentado no diagrama de classes a seguir, o padrão Bridge propõe refatorar essa hierarquia de herança exponencialmente explosiva em duas hierarquias ortogonais:

- Uma, para abstrações independentes de plataforma;
- Outra, para implementações dependentes de plataforma.



Padrões relacionados:



Builder



Observer



State

Atividade

1 - Qual padrão de projeto **não é** adequado para uso quando o processo de criação de objetos for muito trabalhoso, havendo a necessidade da criação de várias instâncias de objetos complexos? Além disso, o projetista desse sistema deseja a diminuição do retrabalho na criação de novos objetos, querendo utilizar um padrão semelhante ao Prototype.

- a) O padrão Builder.
 - b) O padrão Singleton.
 - c) O padrão Factory Method.
 - d) Todas os padrões são inadequados.
 - e) Todos os padrões são adequados.
-

2 - Com o padrão Singleton, resolvemos qual problema de projeto?

- a) Garantia de que um objeto Singleton poderá possuir diversas instâncias no sistema.
 - b) Garantia de flexibilidade nos relacionamentos entre as classes do sistema.
 - c) Garantia de separação entre as classes de abstração e implementação.
 - d) Garantia de que existe apenas uma instância do objeto Singleton no sistema inteiro.
 - e) Garantia de facilidade na criação de objetos complexos.
-

3 - Qual das opções seguintes é um ponto positivo do padrão Bridge?

- a) A separação entre a abstração e a implementação de uma classe.
 - b) O controle da criação de instâncias de um objeto.
 - c) A possibilidade de trabalhar apenas com uma única hierarquia de classes.
 - d) A facilitação de criação de instâncias-protótipos de uma classe.
 - e) A garantia de que apenas um determinado objeto vai existir durante a execução do sistema.
-

Referências

FRIEMAN, E. **Use a cabeça!** Padrões de projeto. 2. ed. Rio de Janeiro: Elsevier, 2007.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns:** elements of reusable object-oriented software. New York: Addison-Wesley Professional, 1994.

LEÃO, L. **Padrões de projeto de software.** Rio de Janeiro: SESES, 2018.

Próxima aula

- Apresentação dos padrões Class Adapter, Composite e Decorator.

Explore mais

- [Um exemplo de uso do Prototype](#);
- Três exemplos de código C# para os padrões:
 - [Prototype](#);
 - [Singleton](#);
 - [Bridge](#).