

Intenção

Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma.

Motivação

É importante para algumas classes ter uma, e apenas uma, instância. Por exemplo, embora possam existir muitas impressoras em um sistema, deveria haver somente um *spooler* de impressoras. Da mesma forma, deveria haver somente um sistema de arquivos e um gerenciador de janelas. Um filtro digital terá somente um conversor A/D. Um sistema de contabilidade será dedicado a servir somente a uma companhia.

Como garantimos que uma classe tenha somente uma instância e que essa instância seja facilmente acessível? Uma variável global torna um objeto acessível, mas não impede você de instanciar múltiplos objetos.

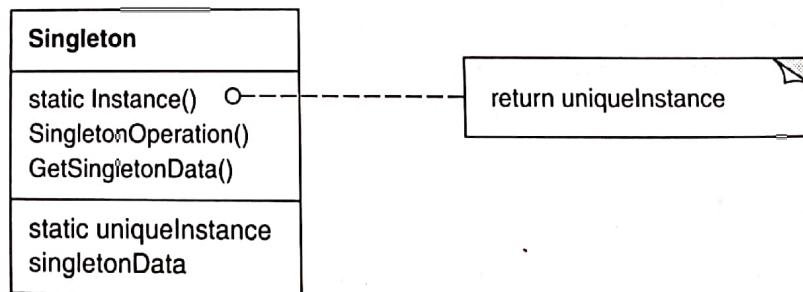
Uma solução melhor seria tornar a própria classe responsável por manter o controle da sua única instância. A classe pode garantir que nenhuma outra instância seja criada (pela interceptação das solicitações para criação de novos objetos), bem como pode fornecer um meio para acessar sua única instância. Este é o *padrão Singleton*.

Aplicabilidade

Use o *padrão Singleton* quando:

- deve haver apenas uma instância de uma classe, e essa instância deve dar acesso aos clientes através de um ponto bem conhecido;
- quando a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código.

Estrutura



Participantes

- **Singleton**
 - define uma operação Instance que permite aos clientes acessarem sua única instância. Instance é uma operação de classe (ou seja, em Smalltalk é um método de classe e em C++ é uma função-membro estática).
 - pode ser responsável pela criação da sua própria instância única.

Colaborações

- Os clientes acessam uma instância Singleton unicamente pela operação Instance do Singleton.

Consequências

O padrão Singleton apresenta vários benefícios:

1. *Acesso controlado à instância única.* Porque a classe Singleton encapsula a sua única instância, pode ter um controle total sobre como e quando os clientes a acessam.
2. *Espaço de nomes reduzido.* O padrão Singleton representa uma melhoria em relação ao uso de variáveis globais. Ele evita a poluição do espaço de nomes com variáveis globais que armazenam instâncias únicas.
3. *Permite um refinamento de operações e da representação.* A classe Singleton pode ter subclasses e é fácil configurar uma aplicação com uma instância desta classe estendida. Você pode configurar a aplicação com uma instância da classe de que necessita em tempo de execução.
4. *Permite um número variável de instâncias.* O padrão torna fácil mudar de idéia, permitindo mais de uma instância da classe Singleton. Além disso, você pode usar a mesma abordagem para controlar o número de instâncias que a aplicação utiliza. Somente a operação que permite acesso à instância de Singleton necessita ser mudada.
5. *Mais flexível do que operações de classe.* Uma outra maneira de empacotar a funcionalidade de um singleton é usando operações de classe (ou seja, funções-membro estáticas em C++ ou métodos de classe em Smalltalk). Porém, as técnicas de ambas as linguagens tornam difícil mudar um projeto para permitir mais que uma instância de uma classe. Além disso, as funções-membro estáticas em C++ nunca são virtuais, o que significa que as subclasses não podem redefinir-las polimorficamente.

Implementação

A seguir apresentamos tópicos de implementação a serem considerados ao usar o padrão Singleton:

1. *Garantindo uma única instância.* O padrão Singleton torna a instância única uma instância normal de uma classe, mas essa classe é escrita de maneira que somente uma instância possa ser criada.
Uma forma comum de fazer isso é ocultando a operação que cria instância usando uma operação de classe (isto é, ou uma função-membro estática ou

um método de classe) que garanta que apenas uma única instância seja criada. Esta operação tem acesso à variável que mantém a única instância, e garante que a variável seja inicializada com a instância única antes de retornar ao seu valor. Esta abordagem assegura que um singleton seja criado e inicializado antes da sua primeira utilização.

Em C++, você pode definir a operação de classe com uma função-membro estática `Instance` da classe `Singleton`. `Singleton` também define uma variável-membro estática `_instance` que contém um apontador para sua única instância.

A classe `Singleton` é declarada como

```
class Singleton {  
public:  
    static Singleton* Instance();  
protected:  
    Singleton();  
private:  
    static Singleton* _instance;  
};
```

A implementação correspondente é a seguinte

```
Singleton* Singleton::_instance = 0;  
  
Singleton* Singleton::Instance () {  
    if (_instance == 0) {  
        _instance = new Singleton;  
    }  
    return _instance;  
}
```

Os clientes acessam o singleton através da função membro `Instance`. A variável `_instance` é inicializada com 0, e a função-membro estática `Instance` retorna o seu valor, inicializando-a com a única instância se ele for 0. `Instance` usa *lazy initialization*; o valor que ela retorna não é criado e armazenado até ser acessado pela primeira vez.

Note que o constructor é protegido. Um cliente que tenta instanciar `Singleton` diretamente obterá como resposta um erro em tempo de compilação. Isto assegura que somente uma instância possa ser criada.

Além do mais, uma vez que `_instance` é um apontador para um objeto `Singleton`, a função-membro `Instance` pode atribuir um apontador para uma subclasse de `Singleton` para esta variável. Daremos um exemplo do que dissemos aqui na seção Exemplo de código.

Há uma outra coisa a ser observada sobre a implementação de C++. Não é suficiente definir o singleton como um objeto global ou estático, confiando numa inicialização automática. Existem três razões para isto:

- (a) não podemos garantir que somente uma instância de um objeto estático será declarada;
- (b) talvez não tenhamos informação suficiente para instanciar cada singleton em tempo de inicialização estática. Um singleton pode necessitar de valores que são computados mais tarde, durante a execução do programa;

- (c) C++ não define a ordem pela qual os constructors para objetos globais são chamados entre unidades de compilação [ES90]. Isto significa que não podem existir dependências entre singletons; se alguma existir, então é inevitável a ocorrência de erro.

Uma deficiência adicional (embora pequena) da abordagem objeto global/estático é que ela força a criação de todos singletons, quer sejam usados ou não. O uso de uma função-membro estática evita todos estes problemas. Em Smalltalk, a função que retorna a instância única é implementada como um método de classe da classe Singleton. Para garantir que somente uma instância seja criada, redefinir a operação *new*. A classe Singleton resultante pode ter os seguintes dois métodos de classe, onde *SoleInstance* é uma variável de classe que não é usada em nenhum outro lugar:

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Criando subclasses da classe Singleton.* O ponto principal não é a definição da subclasse, mas sim a instalação da sua única instância de maneira que os clientes possam ser capazes de usá-la. Em essência, a variável que referencia a instância do singleton deve ser inicializada com uma instância da subclasse. A técnica mais simples é determinar qual singleton você quer usar na operação *Instance* do Singleton. Um exemplo na secção de Exemplo mostra como implementar esta técnica com variáveis do ambiente (operacional).

Uma outra maneira de escolher a subclasse de Singleton é retirar a implementação de *Instance* da classe-mãe (por exemplo, *MazeFactory*) e colocá-la na subclasse. Isto permite a um programador C++ decidir a classe do singleton em tempo de ligação (*link-time*), mantendo-a oculta dos clientes do mesmo (por exemplo, fazendo a ligação com um arquivo-objeto que contém uma implementação diferente).

A solução da ligação fixa a escolha da classe do singleton em tempo de ligação, o que torna difícil escolher a classe do singleton em tempo de execução. O uso de instruções condicionais para determinação da subclasse é mais flexível, porém codifica de maneira rígida o conjunto das classes Singleton possíveis. Nenhuma abordagem é flexível o bastante em todos os casos.

Uma abordagem mais flexível utiliza um sistema de registro de singletons (*registry of singletons*). Em vez de ter *Instance* definindo o conjunto das classes Singleton possíveis, as classes Singleton podem registrar suas instâncias singleton por nome, num sistema de registro de conhecimento geral. O sistema de registro associa nomes e singletons. Os nomes são constituídos de cadeias de caracteres. Quando *Instance* necessita um singleton, ela consulta o sistema de registro, procurando o singleton pelo nome.

O sistema de registro procura o singleton correspondente (se existir) e o retorna ao cliente. Esta solução libera *Instance* da necessidade de ter que conhecer todas as possíveis classes ou instâncias Singleton. Tudo o que é necessário é uma interface comum para todas as classes Singleton, que inclua operações de registro:

```

class Singleton {
public:
    static void Register(const char* name, Singleton* );
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};

```

Register registra a instância de Singleton com um nome fornecido. Para manter o registro simples, necessitaremos que armazene uma lista de objetos NameSingletonPair. Cada NameSingletonPair mapeia (associa) um nome a um singleton. Dado um nome, a operação Lookup encontra o singleton correspondente. Assumiremos que uma variável do ambiente especifica o nome do singleton desejado.

```

singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}

```

Onde as classes Singleton registram a si mesmas? Uma possibilidade é fazê-lo no seu constructor. Por exemplo, uma subclasse MySingleton poderia fazer o seguinte:

```

MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}

```

Naturalmente, o constructor não será chamado a menos que alguém instancie a classe, o que repete o problema que o *padrão Singleton* está tentando resolver! Nós podemos contornar este problema em C++ através da definição de uma instância estática de MySingleton. Por exemplo, podemos definir

```
static MySingleton theSingleton;
```

no arquivo que contém a implementação de MySingleton. A classe Singleton não é mais responsável pela criação do singleton. Em vez disso, sua responsabilidade primária é tornar acessível o objeto singleton escolhido no sistema. A solução que usa o objeto estático ainda apresenta um problema potencial – todas as instâncias de todas as subclasses possíveis de Singleton devem ser criadas, pois, caso contrário, não serão registradas.

Exemplo de código

Suponha que definimos uma classe MazeFactory para construir labirintos, conforme descrito na página 100. MazeFactory define uma interface para construção de

diferentes partes de um labirinto. As subclasses podem redefinir as operações para retornar instâncias de classes-produtos especializadas, tal como `BombedWall` em lugar de simples objetos `Wall`.

O fato relevante aqui é que a aplicação `Maze` necessita somente de uma instância de uma fábrica de labirintos, e que esta instância deverá estar disponível para código que constrói qualquer parte do labirinto. É ai que o padrão Singleton entra. Ao tornar `MazeFactory` um singleton, nós tornamos o objeto-labirinto (`maze`) acessível globalmente sem recorrer a variáveis globais.

Para simplificar, suponhamos que nunca criaremos subclasses de `MazeFactory` (a alternativa será considerada mais à frente). Nós tornamos `MazeFactory` uma classe Singleton em C++, acrescentando uma operação estática `Instance` e um membro estático `_instance` para conter a única instância existente. Também devemos proteger o constructor para prevenir instanciações acidentais, as quais nos levariam a ter mais que uma instância.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

A implementação correspondente é

```
MazeFactory* MazeFactory::_instance = 0

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory();
    }
    return _instance;
}
```

Agora verificaremos o que acontece quando existem subclasses de `MazeFactory` e a aplicação tem que decidir qual delas usar. Selecionaremos o tipo de labirinto através de uma variável do ambiente e acrescentaremos código que instancia a subclasse apropriada de `MazeFactory` com base no valor da variável do ambiente. Um bom lugar para colocar este código é a operação `Instance`, porque ela já instancia `MazeFactory`:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses
    }
```

```

    } else {           // default
        _instance = new MazeFactory();
    }
}

return _instance;
}

```

Note que `Instance` deve ser modificada toda vez que você define uma nova subclasse de `MazeFactory`. Isto pode não ser um problema nesta aplicação, mas pode ser um problema para as fábricas abstratas definidas num *framework*.

Uma solução possível seria usar a técnica do uso de um sistema de registro descrita na seção Implementação. A ligação dinâmica (*dynamic linking*) poderia também ser útil aqui – ela evitaria que a aplicação tivesse que carregar para a memória todas as subclasses que não são usadas.

Usos conhecidos

Um exemplo do *padrão Singleton* em Smalltalk-80 [Par90] é o conjunto de mudanças no código efetuado por `ChangeSet current`. Um exemplo mais sutil é o relacionamento entre classes e suas **metaclasses**. Uma metaclass é a classe de uma classe, e cada metaclass tem uma instância. As metaclasses não têm nomes (exceto indiretamente, através do nome da sua única instância), mas registram e acompanham a sua única instância, e normalmente não criam outra.

O toolkit para construção de interfaces de usuário InterViews [LCI'92] usa o *padrão Singleton* para acessar as únicas instâncias de suas classes `Session` e `WidgetKit`, entre outras. `Session` define o *ciclo* de eventos disparáveis da aplicação principal, armazena o banco de dados das preferências de estilo do usuário e administra conexões para um ou mais dispositivos físicos de *display*. `WidgetKit` é uma *Abstract Factory* (95) para definir os widgets de estilo de interação. A operação `WidgetKit::instance` determina a subclasse específica de `WidgetKit` que é instanciada baseada numa variável de ambiente que `Session` define. Uma operação similar em `Session` determina se são suportados *displays* monocromáticos ou coloridos e configura a instância singleton de `Session` de acordo.

Padrões relacionados

Muitos *padrões* podem ser implementados usando *Singleton*. Ver *Abstract Factory* (95), *Builder* (104) e *Prototype* (121).

Discussão sobre padrões de criação

Existem duas maneiras comuns de parametrizar um sistema pelas classes de objetos que ele cria. Uma é criar subclasses da classe que cria os objetos; isto corresponde a usar o *padrão Factory Method* (112). A principal desvantagem desta solução é que requer a criação de uma nova subclasse somente para mudar a classe do produto. Tais mudanças podem gerar uma cascata de modificações encadeadas. Por exemplo, quando o criador do produto é ele próprio, criado por um método fábrica, então você tem que redefinir também o seu criador.

A outra forma de parametrizar um sistema baseia-se mais na composição de objetos: defina um objeto que seja responsável por conhecer a classe dos objetos produto e torne-o um parâmetro do sistema. Este é o aspecto-chave dos *padrões Abstract Factory* (95), *Builder* (104) e *Prototype* (121). Todos os três *padrões* envolvem

a criação de um novo objeto-fábrica cuja responsabilidade é criar objetos-produtos. Em Abstract Factory, o objeto-fábrica produz objetos de várias classes. Em Builder, um objeto-fábrica constrói incrementalmente um objeto complexo usando um protocolo igualmente complexo. O padrão Prototype faz o objeto-fábrica construir um objeto-produto copiando um objeto prototípico. Neste caso, o objeto-fábrica e o protótipo são o mesmo objeto, porque o protótipo é responsável por retornar o produto.

Considere o *framework* para um editor de desenhos descrito no *padrão Prototype*. Há várias maneiras de parametrizar uma GraphicTool pela classe do produto:

- Aplicando-se o *padrão Factory Method*, uma subclasse de GraphicTool será criada para cada subclasse de Graphic na paleta. A GraphicTool terá uma nova operação NewGraphic, que cada subclasse de GraphicTool redefinirá.
- Aplicando-se o *padrão Abstract Factory*, haverá uma hierarquia de classes de GraphicsFactories, uma para cada subclasse de Graphic. Neste caso, cada fábrica cria apenas o produto: CircleFactory criará círculos (*Circles*), LineFactory criará linhas (*Lines*), e assim por diante. Uma GraphicTool será parametrizada como uma fábrica para criação do tipo apropriado de Graphics.
- Aplicando-se o *padrão Prototype*, cada subclasse de Graphics implementará a operação Clone, e uma GraphicTool será parametrizada com um protótipo da Graphic que ela cria.

Definir qual é o melhor *padrão* depende de muitos fatores. No nosso *framework* para editores de desenhos, o *padrão Factory Method* é inicialmente mais fácil de usar. É fácil definir uma subclasse de GraphicTool e as instâncias de GraphicTool são criadas somente quando a paleta é definida. Aqui, a principal desvantagem é a proliferação de subclasses de GraphicTool, sendo que nenhuma delas faz muita coisa.

O padrão Abstract Factory não oferece uma grande melhoria porque também exige uma hierarquia de classes GraphicsFactory bastante grande. Abstract Factory seria preferível a Factory Method somente se já houvesse uma hierarquia de classes GraphicsFactory – ou porque o compilador a fornece automaticamente (como em Smalltalk ou Objective C) ou porque é necessária em outra parte do sistema.

No geral, o *padrão Prototype* é o melhor para o *framework* de editores do desenho porque ele somente requer a implementação de uma operação Clone em cada classe Graphics. Isso reduz o número de classes, e clone pode ser usado para outras finalidades, além de somente instanciação (por exemplo, uma operação duplicar definida no menu).

O Factory Method torna um projeto mais adaptável e apenas um pouco mais complicado. Outros *padrões de projeto* requerem novas classes, enquanto que Factory Method somente exige uma nova operação. As pessoas freqüentemente usam Factory Method como a maneira padrão de criar objetos, mas não é necessário quando a classe que é instanciada nunca muda ou quando a instanciação ocorre em uma operação que subclasses podem facilmente redefinir, tal como uma operação de inicialização.

Projetos que usam Abstract Factory, Prototype ou Builder são ainda mais flexíveis do que aqueles que utilizam Factory Method, porém, eles também são mais complexos. Freqüentemente, os projetos começam usando Factory Method e evoluem para outros *padrões* de criação à medida que o projetista descobre onde é

necessária maior flexibilidade. O conhecimento de vários *padrões de projeto* lhe dá mais opções quando trocar um critério de projeto por outro.