www.anOw666.blogspot.com
Snow666

uma linguagem de configuração que é analisada por um analisador LALR(1). As ações semânticas do analisador executam operações sobre o construtor que acrescenta informações ao componente de serviço. Neste caso, o analisador é o Director.

Padrões relacionados

Abstract Factory (95) é semelhante a Builder no sentido de que também pode construir objetos complexos. A diferença principal é que o padrão Builder focaliza a construção de um objeto complexo passo a passo. A ênfase do Abstract Factory é sobre famílias de objetos-produto (simples ou complexos). O Builder retorna o produto como um passo final, mas no caso do padrão Abstract Factory o produto é retornado imediatamente.

Um Composite (160) é o que freqüentemente o builder constrói.

FACTORY METHOD — criação de classes

Intenção

Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O Factory Method permite adiar a instanciação para subclasses.

Também conhecido como

Virtual Constructor

Motivação

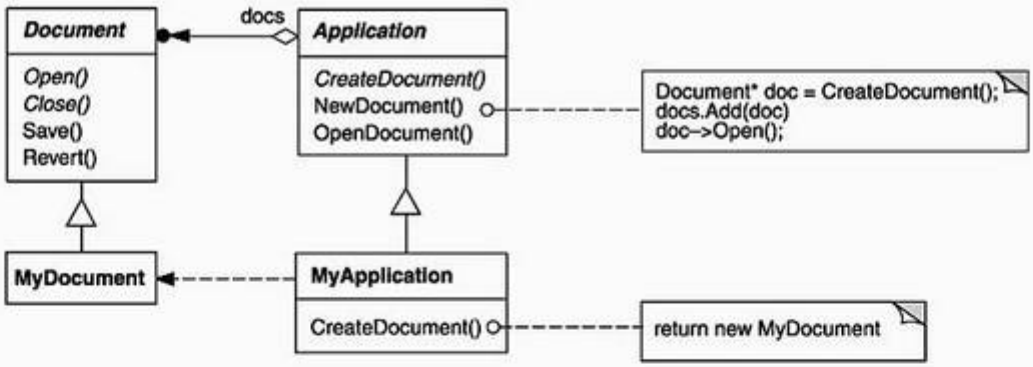
Os *frameworks* usam classes abstratas para definir e manter relacionamentos entre objetos. Um *framework* é freqüentemente responsável também pela criação desses objetos.

Considere um *framework* para aplicações que podem apresentar múltiplos documentos para o usuário. Duas abstrações-chave nesse *framework* são as classes Application (aplicação) e Document (documento). As duas classes são abstratas, e os clientes devem prover subclasses para realizar suas implementações específicas para a aplicação. Por exemplo, para criar uma aplicação de desenho, definimos as classes DrawingApplication e DrawingDocument. A classe Application é responsável pela administração de Documents e irá criá-los conforme exigido – quando o usuário seleciona Open (abrir) ou New (novo), por exemplo, num menu.

Uma vez que a subclasse Document a ser instanciada é própria da aplicação específica, a classe Application não pode prever a subclasse de Document a ser instanciada – a classe Application somente sabe *quando* um documento deve ser criado, e não *que tipo* de Document criar. Isso cria um dilema: o *framework* deve instanciar classes, mas ele somente tem conhecimento de classes abstratas, as quais não pode instanciar.

O padrão Factory Method oferece uma solução. Ele encapsula o conhecimento sobre a subclasse de Document que deve ser criada e move este conhecimento para fora do *framework*.

Snow666



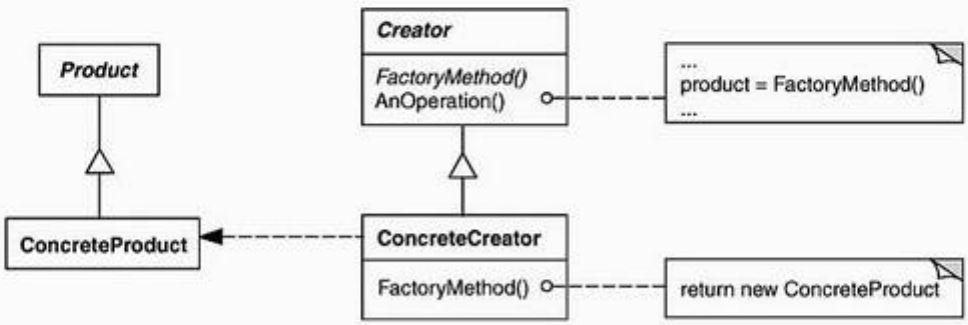
As subclasses de Application redefinem uma operação abstrata CreateDocument em Application para retornar a subclasse apropriada de Document. Uma vez que uma subclasse de Application é instanciada, pode então instanciar Documents específicos da aplicação sem conhecer suas classes. Chamamos CreateDocument um **factory method** porque ele é responsável pela "manufatura" de um objeto.

Aplicabilidade

Use o padrão Factory Method quando:

- uma classe não pode antecipar a classe de objetos que deve criam;
- uma classe quer que suas subclasses especifiquem os objetos que criam;
- classes delegam responsabilidade para uma dentre várias subclasses auxiliares, e você quer localizar o conhecimento de qual subclasse auxiliar que é a delegada.

Estrutura



Participantes

- **Product** (Document)
  - define a interface de objetos que o método fábrica cria.
- **ConcreteProduct** (MyDocument)
  - implementa a interface de Product.
- **Creator** (Application)
  - Declara o método fábrica, o qual retorna um objeto do tipo Product. Creator pode também definir uma implementação por omissão do método factory que retorna por omissão um objeto ConcreteProduct.

Snow666

  - Pode chamar o método factory para criar um objeto Product.
- **ConcreteCreator** (MyApplication)
  - Redefine o método-fábrica para retornar a uma instância de um ConcreteProduct.

Colaborações

- Creator depende das suas subclasses para definir o método fábrica de maneira que retorne uma instância do ConcreteProduct apropriado.

Conseqüências

Os Factory Methods eliminam a necessidade de anexar classes específicas das aplicações no código. O código lida somente com a interface de Product; portanto, ele pode trabalhar com quaisquer classes ConcreteProduct definidas pelo usuário.

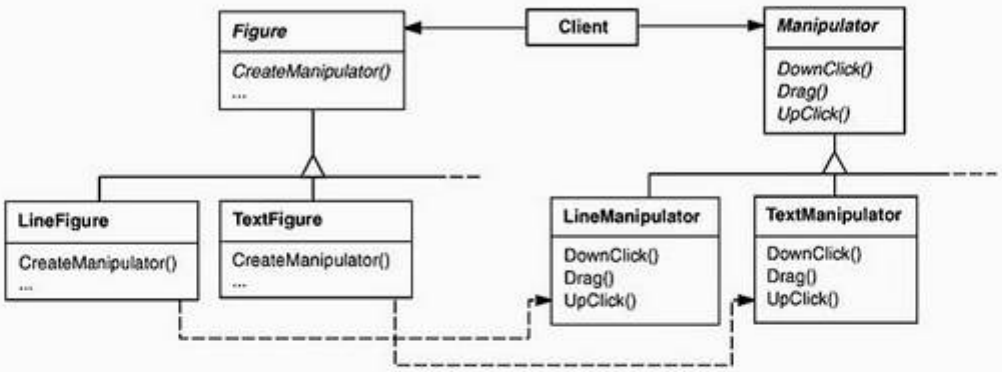
Uma desvantagem em potencial dos métodos-fábrica é que os clientes podem ter que fornecer subclasses da classe Creator somente para criar um objeto ConcreteProduct em particular. Usar subclasses é bom quando o cliente tem que fornecer subclasses a Creator de qualquer maneira, caso contrário, o cliente deve lidar com outro ponto de evolução.

Apresentamos aqui duas conseqüências adicionais do Factory Method:

1. *Fornece ganchos para subclasses*. Criar objetos dentro de uma classe com um método fábrica é sempre mais flexível do que criar um objeto diretamente. Factory Method dá às subclasses um gancho para fornecer uma versão estendida de um objeto.
   No exemplo de Documentos, a classe Document poderia definir um método-fábrica chamado CreateFileDialog que cria um objeto *file dialog* por omissão para abrir um documento existente. Uma subclasse de Document pode definir um *file dialog* específico da aplicação redefinindo este método fábrica. Neste caso, o método fábrica não é abstrato, mas fornece uma implementação por omissão razoável.
2. *Conecta hierarquias de classe paralelas*. Nos exemplos que consideramos até aqui o método-fábrica é somente chamado por Creators. Mas isto não precisa ser obrigatoriamente assim; os clientes podem achar os métodos-fábrica úteis, especialmente no caso de hierarquias de classe paralelas.
   Hierarquias de classe paralelas ocorrem quando uma classe delega alguma das suas responsabilidades para uma classe separada. Considere, por exemplo, figuras que podem ser manipuladas interativamente; ou seja, podem ser esticadas, movidas ou giradas usando o *mouse*. Implementar tais interações não é sempre fácil. Isso freqüentemente requer armazenar e atualizar informação que registra o estado da manipulação num certo momento. Este estado é necessário somente durante a manipulação; portanto, não necessita ser mantido no objeto-figura. Além do mais, diferentes figuras se comportam de modo diferente quando são manipuladas pelo usuário. Por exemplo, esticar uma linha pode ter o efeito de mover um dos extremos, enquanto que esticar um texto pode mudar o seu espaçamento de linhas.

www.snow666.blogspot.com
Snow666

Com essas restrições, é melhor usar um objeto Manipulator separado, que implementa a interação e mantém o registro de qualquer estado específico da manipulação que for necessário. Diferentes figuras utilizarão diferentes subclasses Manipulator para tratar interações específicas. A hierarquia de classes Manipulator resultante é paralela (ao menos parcialmente) à hierarquia de classes de Figure:



A classe Figure fornece um método fábrica CreateManipulator que permite aos clientes criar o correspondente Manipulator de uma Figure. As subclasses de Figure substituem esse método para retornar uma instância da subclasse Manipulator correta para elas. Como alternativa, a classe Figure pode implementar CreateManipulator para retornar por omissão uma instância de manipulator, e as subclasses de Figure podem simplesmente herdar essa instância por omissão. As classes Figure que fizerem assim não necessitarão de uma subclasse correspondente de Manipulator – por isso dizemos que as hierarquias são somente parcialmente paralelas.

Note como o método-fábrica define a conexão entre as duas hierarquias de classes. Nele se localiza o conhecimento de quais classes trabalham juntas.

Implementação

Considere os seguintes tópicos ao aplicar o padrão Factory Method:

1. *Duas variedades principais.* As duas principais variações do padrão Factory Method são: (1) o caso em que a classe Creator é uma classe abstrata e não fornece uma implementação para o método-fábrica que ela declara, e (2) o caso quando o Creator é uma classe concreta e fornece uma implementação por omissão para o método-fábrica. Também é possível ter uma classe abstrata que define uma implementação por omissão, mas isto é menos comum.

   O primeiro caso *exige* subclasses para definir uma implementação porque não existe uma omissão razoável, assim contornando o dilema de ter que instanciar classes imprevisíveis. No segundo caso, o ConcretCreator usa o método fábrica principalmente por razões de flexibilidade. Está seguindo uma regra que diz: "criar objetos numa operação separada de modo que subclasses possam redefinir a maneira como eles são criados". Essa regra garante que projetistas de subclasses, caso necessário, possam mudar a classe de objetos que a classe ancestral instancia.
2. *Métodos-fábrica parametrizados.* Uma outra variante do padrão permite ao método-fábrica criar *múltiplos* tipos de produtos. O método-fábrica recebe um parâmetro que identifica o objeto a ser criado.

www.sn0w666.blogspot.com
Sn0w666

Todos os objetos que o método-fábrica cria compartilharão a interface de Product. No exemplo de Document, Application pode suportar diferentes tipos de Documents. Você passa a Create Document um parâmetro extra para especificar o tipo de documento a ser criado.

O *framework* de edição gráfica Unidraw [VL90] usa esta abordagem para reconstruir objetos salvos em disco. Unidraw define uma classe `Creator` com método-fábrica `Create` que aceita um identificador de classe como argumento. O identificador de classe especifica a classe a ser instanciada. Quando Unidraw salva um objeto em disco, primeiro grava o identificador da classe, e então suas variáveis de instância. Quando reconstrói o objeto de disco, primeiro lê o identificador de classe.

Depois que o identificador de classe é lido, o *framework* chama `Create`, passando o identificador como o parâmetro. `Create` procura o constructor para a classe correspondente, utilizando-o para instanciar o objeto. Por último, `Create` chama a operação `Read` do objeto, a qual lê a informação restante do disco e inicia as variáveis de instância do objeto.

Um método-fábrica parametrizado tem a seguinte forma geral, onde `MyProduct` e `YourProduct` são subclasses de `Product`:

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE)  return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // repete para os produtos restantes

    return 0;
}
```

Redefinir um método fábrica parametrizado permite, fácil e seletivamente, estender ou mudar os produtos que um Creator produz. Você pode introduzir novos identificadores para novos tipos de produtos, ou pode associar identificadores existentes com diferentes produtos.

Por exemplo, uma subclasse `MyCreator` poderia trocar `MyProduct` por YourProduct e suportar uma nova subclasse `TheirProduct`:

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS)  return new MyProduct;
    if (id == MINE)   return new YourProduct;
        // nota: YOURS e MINE foram trocados propositadamente

    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id); // chamado se todos os demais falham
}
```

Note que a última coisa que essa operação faz é chamar `Create` na classe-mãe. Isso porque `MyCreator::Create` trata somente `YOURS`, `MINE` e `THEIRS` de modo diferente da classe-mãe.

Ela não está interessada em outras classes. Daí dizermos que `MyCreator` *estende* os tipos de produtos criados e adia a responsabilidade da criação de todos, exceto uns poucos produtos, para sua superclasse.

www.snow666.blogspot.com
Snow666

3. *Variantes e tópicos específicos das linguagens.* Diferentes linguagens levam a outras variantes interessantes, bem como a cuidados especiais.
Os programas em Smalltalk freqüentemente usam um método que retorna a classe do objeto a ser instanciado. Um método-fábrica Creator pode usar esse valor para criar um produto, e um ConcreteCreator pode armazenar ou mesmo computar esse valor. O resultado é uma associação ainda mais tardia para o tipo de ConcreteProduct a ser instanciado.
Uma versão Smalltalk do exemplo de Document pode definir um método `documentClass` em `Application`. O método `documentClass` retorna a classe apropriada de `Document` para instanciar documentos. A implementação de `documentClass` em `MyApplication` retorna a classe `MyDocument`. Assim, na classe `Application` nós temos

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

Na classe `MyApplication` temos

```
documentClass
    ^ MyDocument
```

que retorna a classe `MyDocument` a ser instanciada para `Application`.
Uma abordagem ainda mais flexível próxima dos métodos-fábrica parametrizados é armazenar a classe a ser criada como uma variável de classe de `Application`. Desse modo, você não tem que introduzir subclasses de `Application` para variar o produto.
Os métodos-fábrica em C++ são sempre funções virtuais e, freqüentemente, virtuais puras. Somente seja cuidadoso para não chamar métodos-fábrica no construtor de Creator – o método-fábrica em ConcreteCreator ainda não estará disponível.
Você pode evitar esse problema sendo cuidadoso, acessando produtos exclusivamente através de operações de acesso que criam o produto sob demanda. Em vez de criar o produto concreto no constructor, o constructor meramente o inicia como 0 (zero). O *accessor* retorna o produto. Mas primeiro ele verifica a existência do produto, e quando não existe o *accessor* o cria. Essa técnica é algumas vezes chamada de **inicialização tardia (*lazy inicialization*)**. O código a seguir mostra uma implementação típica:

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}
```

Snow666

4. *Utilizando* templates *para evitar o uso de subclasses.* Como já mencionamos, outro problema potencial com métodos-fábrica é que podem forçá-lo a introduzir subclasses somente para criar os objetos-produto apropriados. Uma outra maneira de contornar isto em C++ é fornecer uma subclasse *template* de Creator que é parametrizada pela classe Product:

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

Com esse *template*, o cliente fornece somente a classe-produto – não são necessárias subclasses de Creator.

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

5. *Convenções de nomenclatura.* É uma boa prática o uso de convenções de nomenclatura que tornam claro que você está usando métodos-fábrica. Por exemplo, o *framework* de aplicações MacApp para o Macintosh [APP89] sempre declara a operação abstrata que define o método fábrica como `Class* DoMakeClass ()`, onde `Class` é a classe-produto.

Exemplo de código

A função `CreateMaze` (pág. 94) constrói e retorna um labirinto. Um problema com esta função é que codifica de maneira rígida as classes de labirinto, salas, portas e paredes. Nós introduziremos o método-fábrica para permitir às subclasses escolherem estes componentes.

Primeiramente, definiremos o método-fábrica em `MazeGame` para criar os objetos-labirinto, sala, parede e porta:

```
class MazeGame {
public:
    Maze* CreateMaze();

// métodos-fábrica

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
```

www.snow666.blogspot.com
Snow666


```
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Cada método-fábrica retorna um componente de labirinto de um certo tipo. `MazeGame` fornece implementações por omissão que retornam os tipos mais simples de labirinto, salas, portas e paredes.

Agora podemos reescrever `CreateMaze` para usar esses métodos fábrica:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);
    return aMaze;
}
```

Diferentes jogos podem introduzir subclasses de `MazeGame` para especializar partes do labirinto. As subclasses de `MazeGame` podem redefinir alguns ou todos os métodos-fábrica para especificar variações em produtos. Por exemplo, um `BombedMazeGame` pode redefinir os produtos `Room` e `Wall` para retornar variedades com bombas:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

Uma variante `EnchantedMazeGame` poderia ser definida desta forma:

www.snow666.blogspot.com
Snow666


```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Usos conhecidos

Os métodos-fábrica permeiam *toolkits* e *frameworks*. O exemplo precedente de documentos é um uso típico no MacApp e ET++ [WGM88]. O exemplo do manipulador vem do Unidraw.

A classe View no *framework* Model/View/Controller/Smalltalk-80 tem um método defaultController que cria um controlador, e isso pode parecer ser o método-fábrica [Par90]. Mas subclasses de View especificam a classe no seu controlador por omissão através da definição de defaultControllerClass, que retorna a classe da qual defaultController cria instâncias. Assim, defaultControllerClass é o verdadeiro método fábrica, isto é, o método que as subclasses deveriam redefinir.

Um exemplo mais esotérico no Smalltalk-80 é o método-fábrica parserClass definido por Behavior (uma superclasse de todos os objetos que representam classes). Isto permite a uma classe usar um parser (analisador) customizado para seu código-fonte. Por exemplo, um cliente pode definir uma classe SQLParser para analisar o código-fonte de uma classe com comandos SQL embutidos. A Classe Behavior implementa parserClass retornando a classe Parser padrão do Smalltalk. A classe que inclui comandos SQL embutidos redefine este método (como um método de classe) e retorna a classe SQLParser.

O sistema ORB Orbix da IONA Technologies [ION94] usa Factory Method para gerar um tipo apropriado de proxy (ver Proxy (198)) quando um objeto solicita uma referência para um objeto remoto. O Factory Method torna fácil substituir o proxy-padrão por um outro que, por exemplo, use *caching* do lado do cliente.

Padrões relacionados

Abstract Factory (95) é freqüentemente implementado utilizado o padrão Factory Method. O exemplo na relação de Motivação no padrão Abstract Factory também ilustra o padrão Factory Method.

Factory Methods são usualmente chamados dentro de Template Methods (301). No exemplo do documento acima, NewDocument é um *template method*.

Prototypes (121) não exigem subclassificação de Creator. Contudo, freqüentemente necessitam uma operação Initialize na classe Product. A Creator usa Initialize para iniciar o objeto. O Factory Method não exige uma operação desse tipo.