TEMPLATE METHOD — comportamental de classes

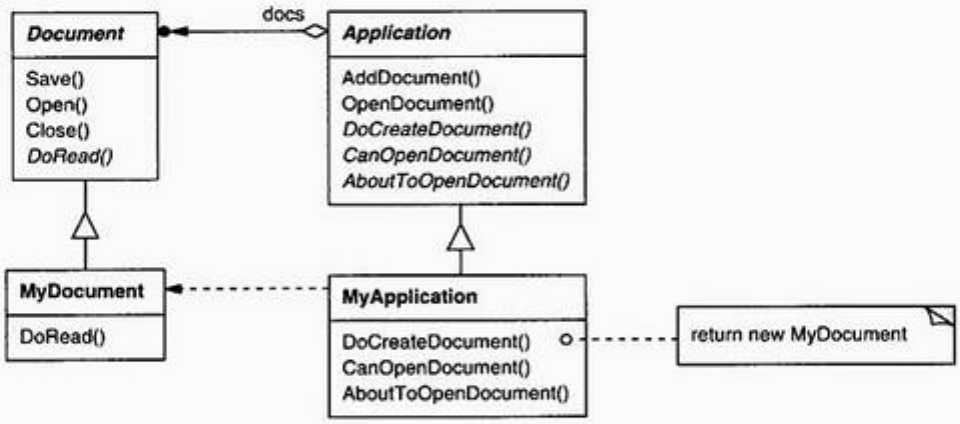
Intenção

Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

Motivação

Considere um *framework* para aplicações que fornece as classes Application e Document. A classe Application é responsável por abrir documentos existentes armazenados num formato externo, tal como um arquivo. Um objeto Document representa a informação num documento, depois que ela foi lida do arquivo.

As aplicações construídas com o *framework* podem criar subclasses de Application e Document para atender necessidades específicas. Por exemplo, uma aplicação de desenho define as subclasses DrawApplication e DrawDocument; uma aplicação de planilha define as subclasses SpreadsheetApplication e SpreadsheetDocument.



A classe abstrata Application define o algoritmo para abrir e ler um documento na sua operação OpenDocument:

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // não consegue tratar este documento
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

OpenDocument define cada passo para a abertura de um documento. Ela verifica se o documento pode ser aberto, cria o objeto Document específico para a aplicação, acrescenta-o ao seu conjunto de documentos e lê Document de um arquivo.

Chamamos OpenDocument um **template method**. Um método-template (gabarito, C++ suporta templates) define um algoritmo em termos da operação abstrata que as subclasses redefinem para fornecer um comportamento concreto. As subclasses da aplicação definem os passos do algoritmo que verifica se o documento pode ser aberto (CanOpenDocument) e cria o Document (DoCreateDocument). As classes Document definem a etapa que lê o documento (DoRead). O método template também define uma operação que permite às subclasses de Application saberem quando o documento está para ser aberto (AboutToOpenDocument), no caso de elas terem essa preocupação.

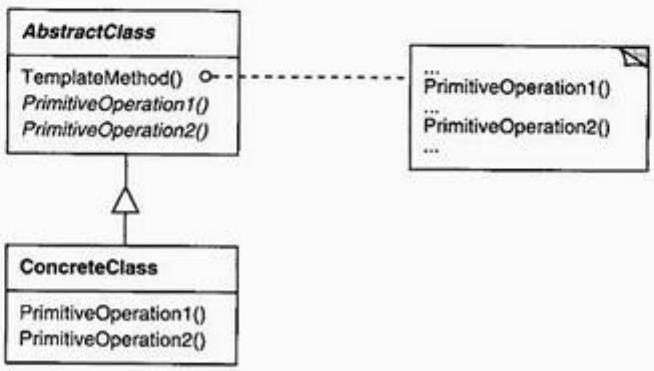
Pela definição de alguns dos passos de um algoritmo usando operações abstratas, o método template fixa a sua ordem, mas deixa as subclasses de Application e Document variarem aqueles passos necessários para atender suas necessidades.

Aplicabilidade

O padrão Template Method pode ser usado:

- para implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar.
- quando o comportamento comum entre subclasses deve ser fatorado e concentrado numa classe comum para evitar a duplicação de código. Este é um bom exemplo de "refatorar para generalizar", conforme descrito por Opdyke e Johnson [OJ93]. Primeiramente, você identifica as diferenças no código existente e então separa as diferenças em novas operações. Por fim, você substitui o código que apresentava as diferenças por um método-template que chama uma dessas novas operações.
- para controlar extensões de subclasses. Você pode definir um método-template que chama operações "gancho" (ver Conseqüências) em pontos específicos, desta forma permitindo extensões somente nesses pontos.

Estrutura



Participantes

- **AbstractClass** (Application)

  - define **operações primitivas** abstratas que as subclasses concretas definem para implementar passos de um algoritmo.
  - implementa um método-template que define o esqueleto de um algoritmo. O método-template invoca operações primitivas, bem como operações definidas em AbstractClass ou ainda outros objetos.
- **ConcreteClass** (MyApplication)
  - implementa as operações primitivas para executarem os passos específicos do algoritmo da subclasse.

Colaborações

- ConcreteClass depende de AbstractClass para implementar os passos invariantes do algoritmo.

Conseqüências

Os métodos-template são uma técnica fundamental para a reutilização de código. São particularmente importantes em bibliotecas de classe porque são os meios para a fatoração dos comportamentos comuns.

Os métodos-template conduzem a uma estrutura de inversão de controle, algumas vezes chamada de "o princípio de Hollywood", ou seja: "não nos chame, nós chamaremos você" [Swe85]. Isso se refere a como uma classe-mãe chama as operações de uma subclasse, e não o contrário.

Os métodos-template chamam os seguintes tipos de operações:

- operações concretas (ou em ConcreteClass ou em classes-clientes);
- operações concretas de AbstractClass (isto é, operações que são úteis para subclasses em geral);
- operações primitivas (isto é, operações abstratas);
- métodos fábrica (ver Factory Method, 112); e
- **operações-gancho (hook operations)** que fornecem comportamento-padrão que subclasses podem estender se necessário. Uma operação-gancho freqüentemente não executa nada por padrão.

É importante para os métodos-template especificar quais operações são ganchos (*podem* ser redefinidas) e quais são operações abstratas (*devem* ser redefinidas). Para reutilizar uma classe abstrata efetivamente, os codificadores de subclasses devem compreender quais as operações são projetadas para redefinição.

Uma subclasse pode *estender* o comportamento de uma operação de uma classe-mãe pela redefinição da operação e chamando a operação-mãe explicitamente:

```
void DerivedClass::Operation () {
    ParentClass::Operation();
    // comportamento estendido de DerivedClass
}
```

Infelizmente, é fácil esquecer de chamar a operação herdada. Podemos transformar uma tal operação num método-template para dar à (classe) mãe controle sobre a maneira como as subclasses a estendem. A idéia é chamar uma operação-gancho a partir de um método-template na classe-mãe. Então, as subclasses podem substituir essa operação-gancho:

```
void ParentClass::Operation () {
    // comportamento da ParentClass
    HookOperation();
}
```

Um `HookOperation` não faz nada em `ParentClass`:

```
void ParentClass::HookOperation () { }
```

As subclasses substituem `HookOperation` para estender o seu comportamento:

```
void DerivedClass::HookOperation () {
    // extensão da classe derivada
}
```

Implementação

Três aspectos da implementação são dignos de nota:

1. *Usando o controle de acesso de C++.* Em C++, as operações primitivas que um método template chama podem ser declaradas membros protegidos. Isso assegura que elas são chamadas somente pelo método-template.
   As operações primitivas que *devem* ser substituídas são declaradas como virtuais puras. O método-template em si não deveria ser substituído; portanto, você pode tornar o método-template uma função-membro não-virtual.
2. *Minimizando operações primitivas.* Um objetivo importante ao projetar métodos templates é minimizar o número de operações primitivas que uma subclasse deve substituir para materializar o algoritmo. Quanto mais operações necessitam ser substituídas, mais tediosas se tornam as coisas para os clientes.
3. *Convenções de nomenclatura.* Você pode identificar as operações que deveriam ser substituídas pela adição de um prefixo aos seus nomes. Por exemplo, o *framework* MacApp para aplicações Macintosh [App89] prefixa os nomes de métodos template com "Do –": "DoCreateDocument", "DoRead", e assim por diante.

Exemplo de código

O seguinte exemplo em C++ mostra como uma classe-mãe pode impor e garantir um invariante para suas subclasses. O exemplo é tirado do AppKit do NeXT [Add94]. Considere uma classe `View` que suporta a ação de desenhar na tela. `View` garante a invariante que as suas subclasses podem desenhar em uma visão somente após ela se tornar o "foco", o que requer que um certo estado de desenho seja estabelecido de forma apropriada (por exemplo, cores e fontes tipográficas).

Podemos usar um método-template `Display` para estabelecer este estado. `View` define duas operações concretas, `SetFocus` e `ResetFocus`, que estabelecem e fazem a limpeza do estado de desenho, respectivamente. A operação-gancho `DoDisplay` de `View` executa o desenho propriamente dito. `Display` chama `SetFocus` antes de `DoDisplay` para iniciar o estado de desenho; `Display` chama `ResetFocus` posteriormente para sair do estado de desenho.

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

Para manter o invariante, os clientes de `View` sempre chamam `Display` e subclasses de *View* redefinem `DoDisplay`.

`DoDisplay` não faz nada em `View`:

```
void View::DoDisplay () { }
```

As subclasses a redefinem para acrescentar o seu comportamento de desenho específico:

```
void MyView::DoDisplay () {
    // finaliza os conteúdos da visão
}
```

Usos conhecidos

Os métodos-template são tão fundamentais que eles podem ser encontrados em quase todas as classes abstratas.

Wirfs-Brock e outros [WBWW90,WBJ90] fornecem uma boa visão geral e uma boa discussão de métodos-template.

Padrões relacionados

Os Factory Methods (112) são freqüentemente chamados por métodos-template. No exemplo da seção Motivação, o método-fábrica DoCreateDocument é chamado pelo método-template OpenDocument.

Strategy (292): Métodos-template usam a herança para variar parte de um algoritmo. Estratégias usam a delegação para variar o algoritmo inteiro.

VISITOR — comportamental de objetos

Intenção

Representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Motivação

Considere um compilador que representa programas como árvores sintáticas abstratas. Ele necessitará executar operações nas árvores sintáticas abstratas para análises "semânticas estáticas", como verificar se todas as variáveis estão definidas. Também necessitará gerar código executável. Assim, poderá definir operações para verifica-