refactoring.guru

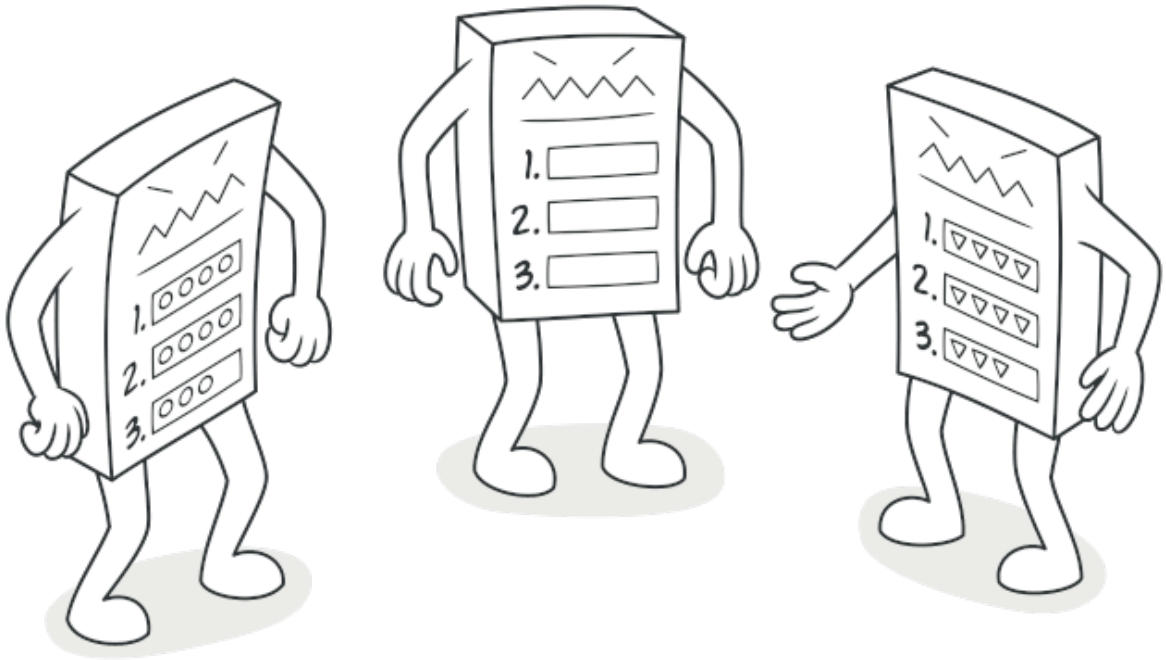
Template Method

11–15 minutes

Também conhecido como: Método padrão

Propósito

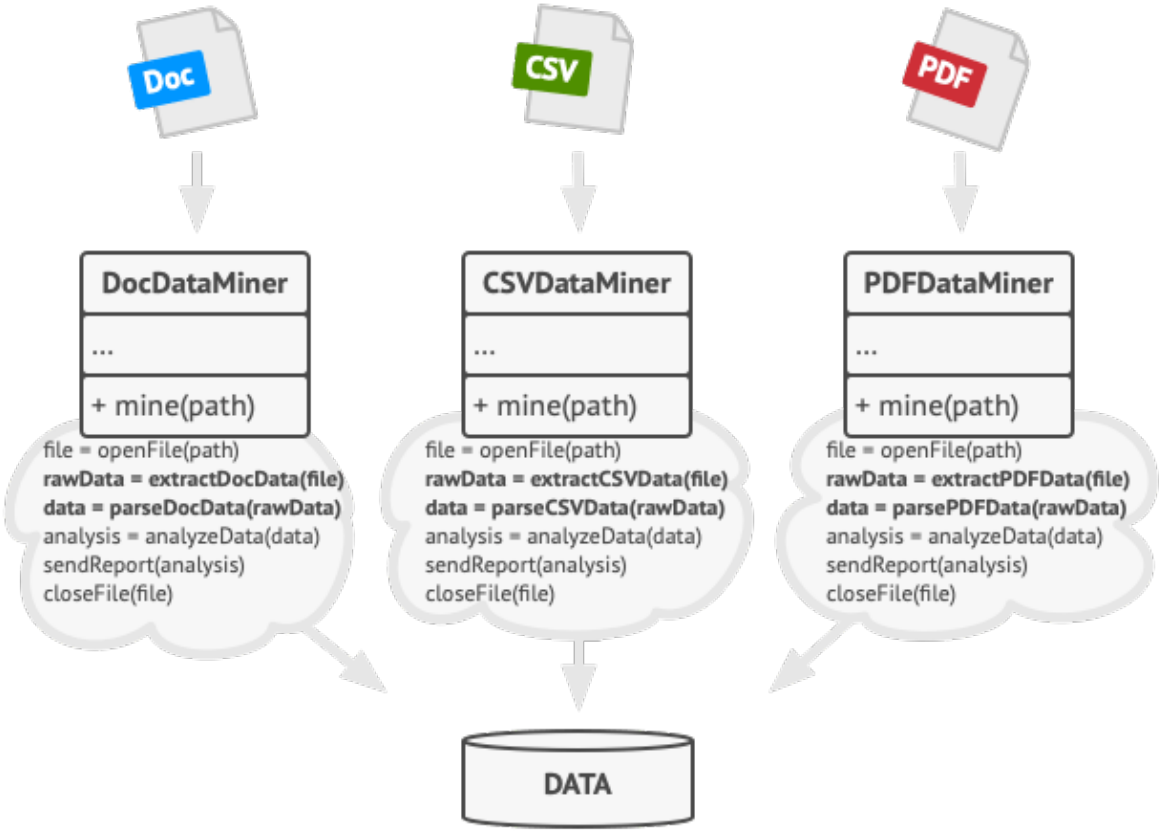
O **Template Method** é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.



Problema

Imagine que você está criando uma aplicação de mineração de dados que analisa documentos corporativos. Os usuários alimentam a aplicação com documentos em vários formatos (PDF, DOC, CSV), e ela tenta extrair dados significativos desses documentos para um formato uniforme.

A primeira versão da aplicação podia funcionar somente com arquivos DOC. Na versão seguinte, ela era capaz de suportar arquivos CSV. Um mês depois, você a “ensinou” a extrair dados de arquivos PDF.



Classes de mineração de dados continham muito código duplicado.

Em algum momento você percebeu que todas as três classes tem muito código parecido. Embora o código para lidar com vários formatos seja inteiramente diferente em todas as classes, o

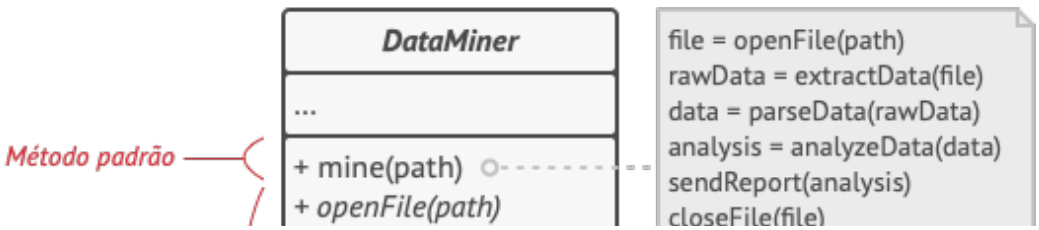
código para processamento de dados e análise é quase idêntico. Não seria bacana se livrar da duplicação de código, deixando a estrutura do algoritmo intacta?

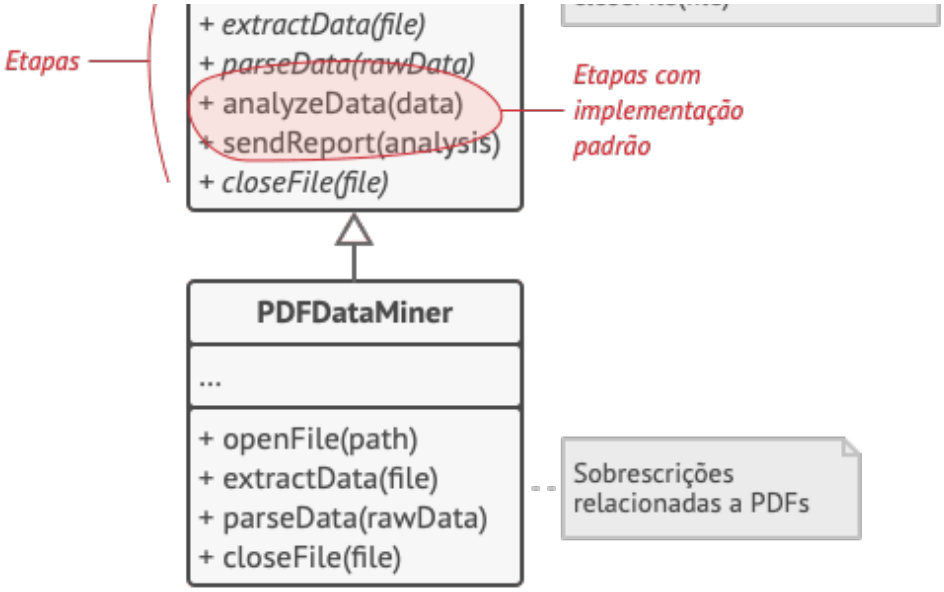
Havia outro problema relacionado com o código cliente que usou essas classes. Ele tinha muitas condicionais que pegavam um curso de ação apropriado dependendo da classe do objeto processador. Se todas as três classes processantes tiverem uma interface comum ou uma classe base, você poderia eliminar as condicionais no código cliente e usar polimorfismo quando chamar métodos em um objeto sendo processado.

Solução

O padrão do Template Method sugere que você quebre um algoritmo em uma série de etapas, transforme essas etapas em métodos, e coloque uma série de chamadas para esses métodos dentro de um único *método padrão*. As etapas podem ser tanto `abstratas`, ou ter alguma implementação padrão. Para usar o algoritmo, o cliente deve fornecer sua própria subclasse, implementar todas as etapas abstratas, e sobrescrever algumas das opcionais se necessário (mas não o próprio método padrão).

Vamos ver como isso vai funcionar com nossa aplicação de mineração de dados. Nós podemos criar uma classe base para todos os três algoritmos de processamento. Essa classe define um método padrão que consiste de uma série de chamadas para várias etapas de processamento de documentos.





O método padrão quebra o algoritmo em etapas, permitindo que subclasses sobrescrevam essas etapas mas não o método atual.

A princípio nós podemos declarar todos os passos como `abstratos`, forçando as subclasses a fornecer suas próprias implementações para esses métodos. No nosso caso, as subclasses já tem todas as implementações necessárias, então a única coisa que precisamos fazer é ajustar as assinaturas dos métodos para coincidirem com os da superclasse.

Agora vamos ver o que podemos fazer para nos livrarmos do código duplicado. Parece que o código para abrir/fechar arquivos e extrair/analisar os dados são diferentes para vários formatos de dados, então não tem porque tocar nesses métodos. Contudo, a implementação dessas etapas, tais como analisar os dados brutos e compor relatórios, é muito parecida, então eles podem ser erguidos para a classe base, onde as subclasses podem compartilhar o código.

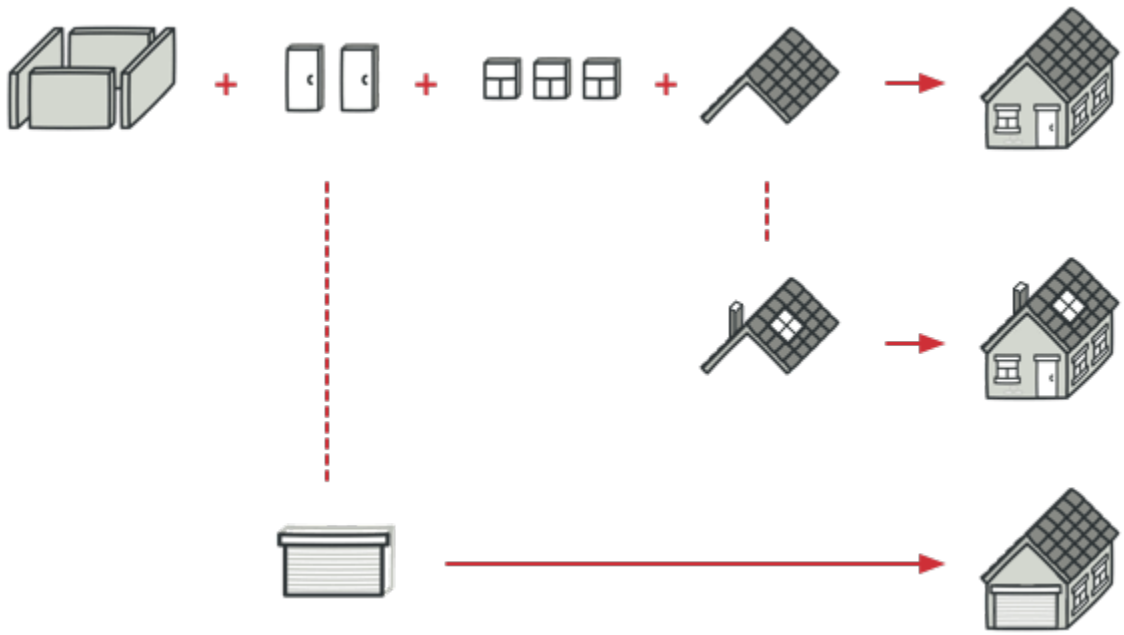
Como você pode ver, nós temos dois tipos de etapas:

- *etapas abstratas* devem ser implementadas por cada subclasse

- *etapas opcionais* já tem alguma implementação padrão, mas ainda podem ser sobrescritas se necessário.

  Existe outro tipo de etapa chamado *ganchos*(hooks). Um gancho é uma etapa opcional com um corpo vazio. Um método padrão poderia funcionar até mesmo se um hook não for sobrescrito. Geralmente os hooks são colocados antes e depois de etapas cruciais de algoritmos, fornecendo às subclasses com pontos de extensão adicionais para um algoritmo.

Analogia com o mundo real



Um típico plano de arquitetura pode ser levemente alterado para melhor servir às necessidades do cliente.

A abordagem do Template Method pode ser usada na construção em massa de moradias. O plano arquitetônico para construir uma casa padrão pode conter diversos pontos de extensões que permitiriam um dono em potencial ajustar alguns detalhes na casa resultante.

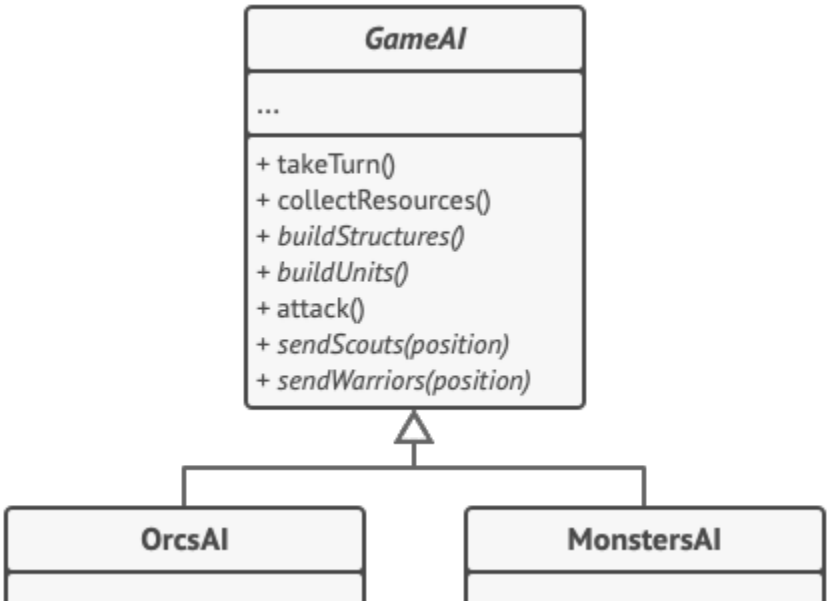
Cada etapa de construção, tais como estabelecer as fundações, enquadramento, construindo paredes, instalando encanamento e fiação elétrica para água e eletricidade, etc. pode ser levemente mudado para se ter uma casa resultante um pouco diferente das outras.

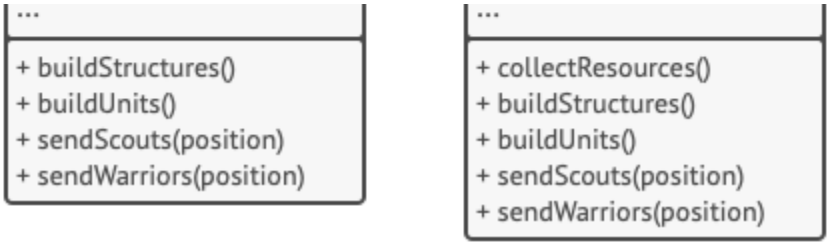
Estrutura

1. A **Classe Abstrata** declara métodos que agem como etapas de um algoritmo, bem como o próprio método padrão que chama esses métodos em uma ordem específica. Os passos podem ser declarados como `abstratos` ou ter alguma implementação padrão.
2. As **Classes Concretas** podem sobrescrever todas as etapas, mas não o próprio método padrão.

Pseudocódigo

Neste exemplo, o padrão **Template Method** fornece um "esqueleto" para várias ramificações de inteligência artificial de um jogo de estratégia simples.





Classes IA de um jogo simples.

Todas as raças do jogo tem quase o mesmo tipo de unidades e construções. Portanto você pode reutilizar a mesma estrutura de IA para várias raças, enquanto é capaz de sobrescrever alguns dos detalhes. Com essa abordagem, você pode sobrescrever a IA dos Orcs para torná-la mais agressiva, fazer os humanos mais orientados a defesa e fazer os monstros serem incapazes de construir qualquer coisa. Adicionando uma nova raça ao jogo irá necessitar a criação de uma nova subclasse IA e a sobrescrição dos métodos padrão declarados na classe IA base.

```
// A classe abstrata define um método padrão que contém um
// esqueleto de algum algoritmo composto de chamadas,
geralmente
// para operações abstratas primitivas. Subclasses concretas
// implementam essas operações, mas deixam o método padrão
em si
// intacto.
class GameAI is
    // O método padrão define o esqueleto de um algoritmo.
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()
```

```
    // Algumas das etapas serão implementadas diretamente na
    // classe base.
    method collectResources() is
        foreach (s in this.builtStructures) do
            s.collect()

    // E algumas delas podem ser definidas como abstratas.
    abstract method buildStructures()
    abstract method buildUnits()

    // Uma classe pode ter vários métodos padrão.
    method attack() is
        enemy = closestEnemy()
        if (enemy == null)
            sendScouts(map.center)
        else
            sendWarriors(enemy.position)

    abstract method sendScouts(position)
    abstract method sendWarriors(position)

// Classes concretas têm que implementar todas as operações
// abstratas da classe base, mas não podem sobrescrever o
método
// padrão em si.
class OrcsAI extends GameAI is
    method buildStructures() is
        if (there are some resources) then
            // Construir fazendas, depois quartéis, e então uma
            // fortaleza.
```

```
    method buildUnits() is
        if (there are plenty of resources) then
            if (there are no scouts)
                // Construir peão, adicionar ele ao grupo de
                // scouts (batedores).
            else
                // Construir um bruto, adicionar ele ao grupo
                // dos guerreiros.



    method sendScouts(position) is
        if (scouts.length > 0) then
            // Enviar batedores para posição.



    method sendWarriors(position) is
        if (warriors.length > 5) then
            // Enviar guerreiros para posição.

// As subclasses também podem sobrescrever algumas operações
com
// uma implementação padrão.
class MonstersAI extends GameAI is
    method collectResources() is
        // Monstros não coletam recursos.

    method buildStructures() is
        // Monstros não constroem estruturas.
```

```
    method buildUnits() is
        // Monstros não constroem unidades.
```

Aplicabilidade

Utilize o padrão Template Method quando você quer deixar os clientes estender apenas etapas particulares de um algoritmo, mas não todo o algoritmo e sua estrutura.

O Template Method permite que você transforme um algoritmo monolítico em uma série de etapas individuais que podem facilmente ser estendidas por subclasses enquanto ainda mantém intacta a estrutura definida em uma superclasse.

Utilize o padrão quando você tem várias classes que contém algoritmos quase idênticos com algumas diferenças menores. Como resultado, você pode querer modificar todas as classes quando o algoritmo muda.

Quando você transforma tal algoritmo em um Template Method, você também pode erguer as etapas com implementações similares para dentro de uma superclasse, eliminando duplicação de código. Códigos que variam entre subclasses podem permanecer dentro das subclasses.

Como implementar

1. Analise o algoritmo alvo para ver se você quer quebrá-lo em etapas. Considere quais etapas são comuns a todas as subclasses e quais permanecerão únicas.
2. Crie a classe abstrata base e declare o método padrão e o

conjunto de métodos abstratos representando as etapas do algoritmo. Contorne a estrutura do algoritmo no método padrão ao executar as etapas correspondentes. Considere tornar o método padrão como `final` para prevenir subclasses de sobrescrevê-lo.

3. Tudo bem se todas as etapas terminarem sendo abstratas. Contudo, alguns passos podem se beneficiar de ter uma implementação padrão. Subclasses não tem que implementar esses métodos.
4. Pense em adicionar ganchos entre as etapas cruciais do algoritmo.
5. Para cada variação do algoritmo, crie uma nova subclasse concreta. Ela *deve* implementar todas as etapas abstratas, mas *pode* também sobrescrever algumas das opcionais.

Prós e contras

- Você pode deixar clientes sobrescrever apenas certas partes de um algoritmo grande, tornando-os menos afetados por mudanças que acontece por outras partes do algoritmo.
- Você pode elevar o código duplicado para uma superclasse.
- Alguns clientes podem ser limitados ao fornecer o esqueleto de um algoritmo.
- Você pode violar o *princípio de substituição de Liskov* ao suprimir uma etapa padrão de implementação através da subclasse.
- Implementações do padrão Template Method tendem a ser mais difíceis de se manter quanto mais etapas eles tiverem.

Relações com outros padrões

- O Factory Method é uma especialização do Template Method. Ao mesmo tempo, o *Factory Method* pode servir como uma etapa em um *Template Method* grande.
- O Template Method é baseado em herança: ele permite que você altere partes de um algoritmo ao estender essas partes em subclasses. O Strategy é baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir ele como diferentes estratégias que correspondem a aquele comportamento. O *Template Method* funciona a nível de classe, então é estático. O *Strategy* trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.