Padrões relacionados

Adapter (140): Um padrão Decorator é diferente de um padrão Adapter no sentido de que um Decorator somente muda as responsabilidades de um objeto, não a sua interface; já um Adapter dará a um objeto uma interface completamente nova.

Composite (160): Um padrão Decorator pode ser visto como um padrão Composite degenerado com somente um componente. Contudo, um Decorator acrescenta responsabilidades adicionais – ele não se destina à agregação de objetos.

Strategy (292): Um padrão Decorator permite mudar a superfície de um objeto, um padrão Strategy permite mudar o seu interior. Portanto, essas são duas maneiras alternativas de mudar um objeto.

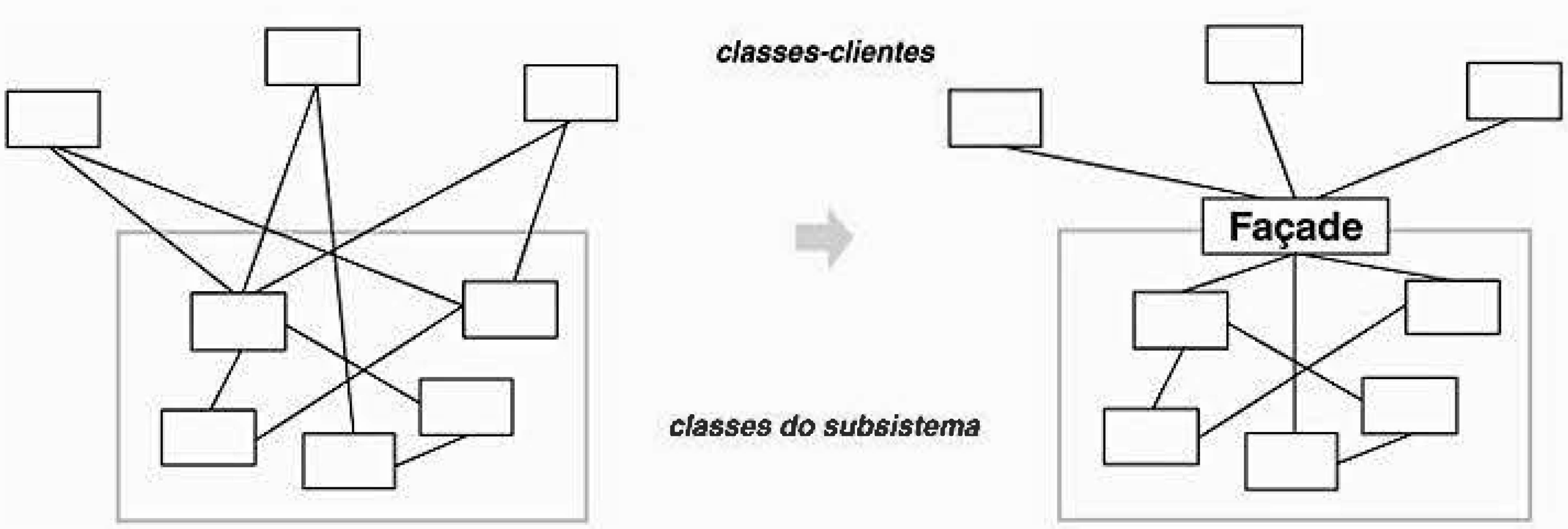
FAÇADE — estrutural de objetos

Intenção

Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.

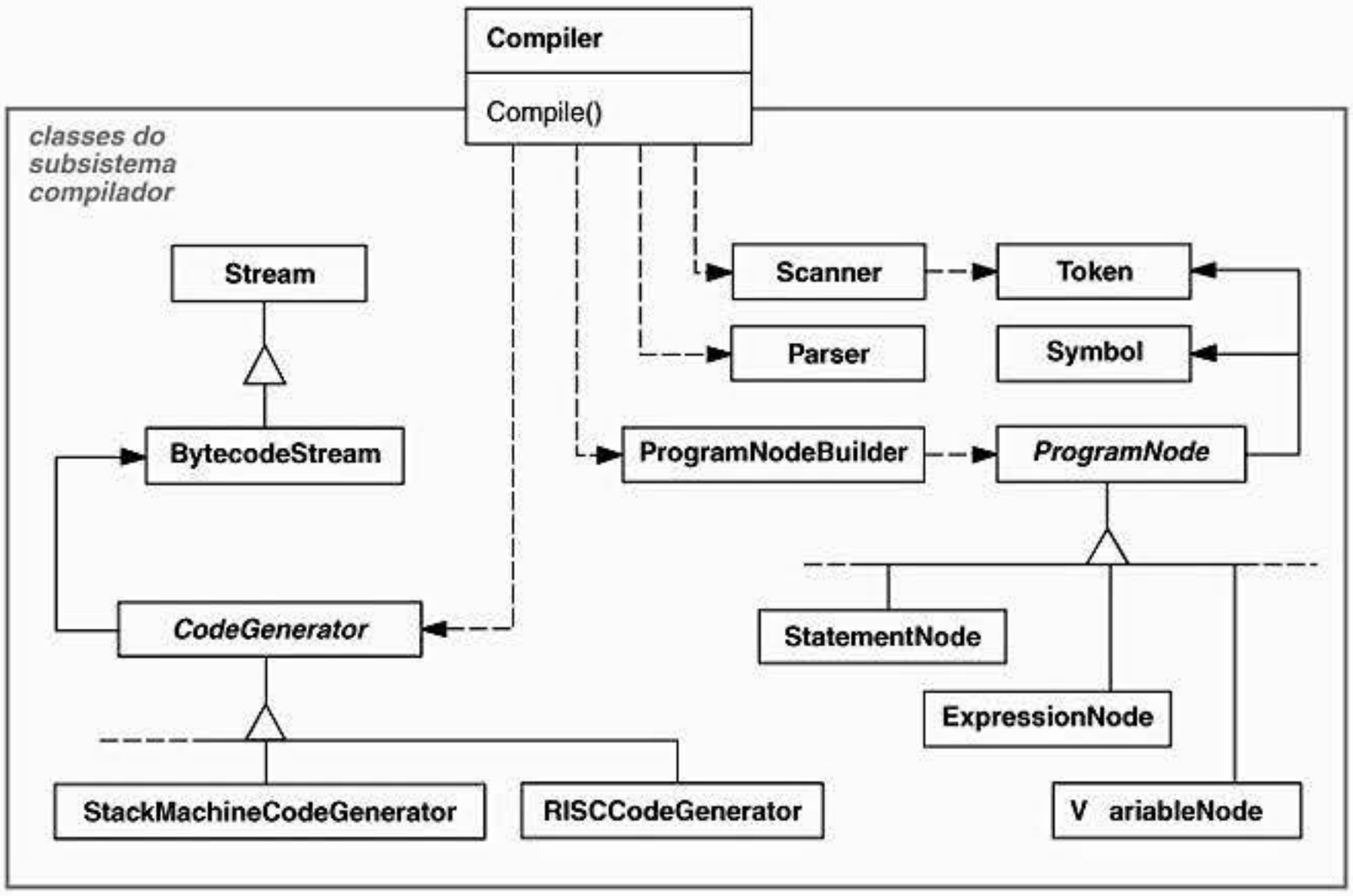
Motivação

Estruturar um sistema em subsistemas ajuda a reduzir a complexidade. Um objetivo comum de todos os projetos é minimizar a comunicação e as dependências entre subsistemas. Uma maneira de atingir esse objetivo é introduzir um objeto **façade** (fachada), o qual fornece uma interface única e simplificada para os recursos e facilidades mais gerais de um subsistema.



Considere, por exemplo, um ambiente de programação que fornece acesso às aplicações para o seu subsistema compilador. Esse subsistema contém classes, tais como Scanner, Parser, ProgramNode, BytecodeStream e ProgramNodeBuilder, que implementam o compilador. Algumas aplicações especializadas podem precisar acessar essas classes diretamente. Mas a maioria dos clientes de um compilador geralmente não se preocupa com detalhes tais como análise e geração de código; eles apenas querem compilar seu código. Para eles, as interfaces poderosas, porém de baixo nível, do subsistema compilador somente complicam sua tarefa.

Para fornecer uma interface de nível mais alto que pode isolar os clientes dessas classes, o subsistema compilador também inclui uma classe Compiler. A classe Compiler funciona como uma fachada: oferece aos clientes uma interface única e simples para o subsistema compilador. Junta as classes que implementam a funcionalidade de um compilador, sem ocultá-las completamente. O compilador Façade torna a vida mais fácil para a maioria dos programadores, sem, entretanto, ocultar a funcionalidade de nível mais baixo dos poucos que a necessitam.



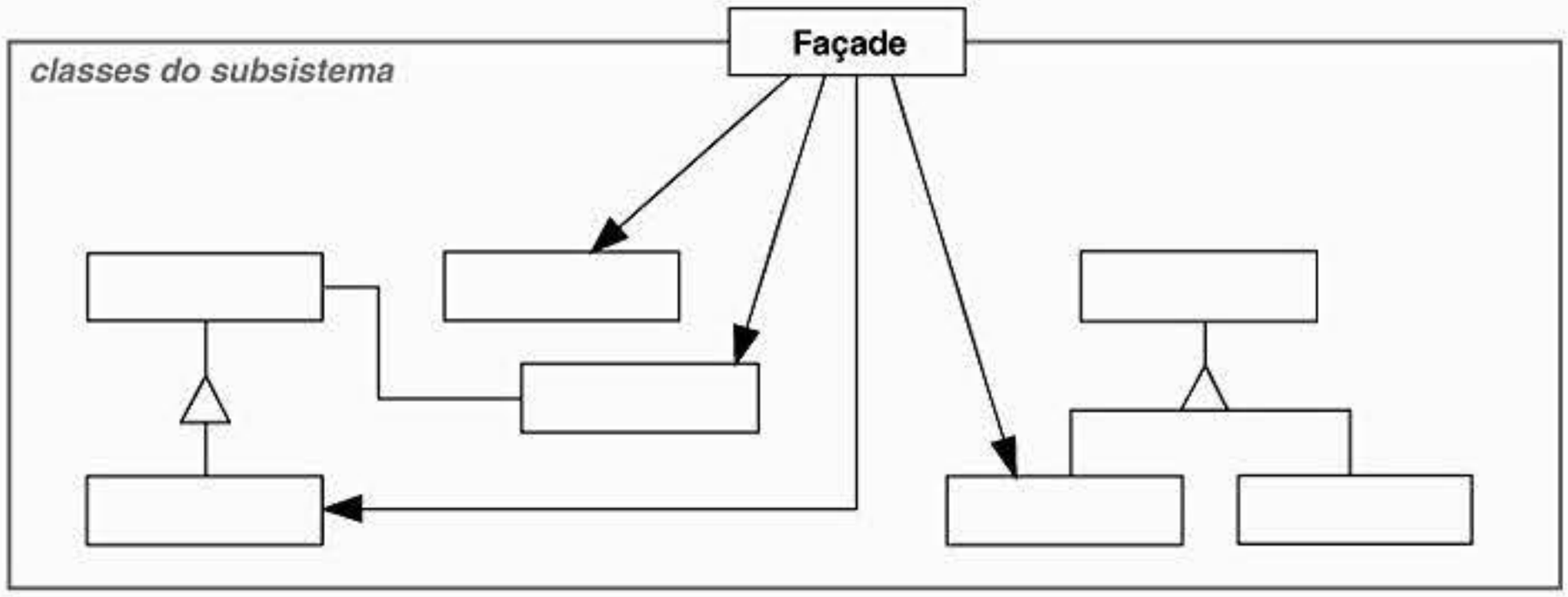
Aplicabilidade

Use o padrão Façade quando:

- você desejar fornecer uma interface simples para um subsistema complexo. Os subsistemas se tornam mais complexos à medida que evoluem. A maioria dos padrões, quando aplicados, resulta em mais e menores classes. Isso torna o subsistema mais reutilizável e mais fácil de customizar, porém, também se torna mais difícil de usar para os clientes que não precisam customizá-lo. Uma fachada pode fornecer, por comportamento-padrão, uma visão simples do sistema, que é boa o suficiente para a maioria dos clientes. Somente os clientes que demandarem maior customização necessitarão olhar além da fachada;
- existirem muitas dependências entre clientes e classes de implementação de uma abstração. Ao introduzir uma fachada para desacoplar o subsistema dos clientes e de outros subsistemas, estar-se-á promovendo a independência e portabilidade dos subsistemas.

- você desejar estruturar seus subsistemas em camadas. Use uma fachada para definir o ponto de entrada para cada nível de subsistema. Se os subsistemas são independentes, você pode simplificar as dependências entre eles fazendo com que se comuniquem uns com os outros exclusivamente através das suas fachadas.

Estrutura



Participantes

- **Façade** (Compiler)
  - conhece quais as classes do subsistema são responsáveis pelo atendimento de uma solicitação;
  - delega solicitações de clientes a objetos apropriados do subsistema.
- **Classes de subsistema** (Scanner, Parser, ProgramNode, etc.)
  - implementam a funcionalidade do subsistema;
  - encarregam-se do trabalho atribuído a elas pelo objeto Façade;
  - não têm conhecimento da façade; isto é, não mantêm referências para a mesma.

Colaborações

- Os clientes se comunicam com um subsistema através do envio de solicitações para Façade, que as repassa para o(s) objeto(s) apropriado(s) do subsistema. Embora os objetos do subsistema executem o trabalho real, a façade pode ter de efetuar trabalho próprio para traduzir a sua interface para as interfaces de subsistemas.
- Os clientes que usam a façade não acessam os objetos do subsistema diretamente.

Conseqüências

O padrão Façade oferece os seguintes benefícios:

1. Isola os clientes dos componentes do subsistema, dessa forma reduzindo o número de objetos com que os clientes têm que lidar e tornando o subsistema mais fácil de usar;

2. Promove um acoplamento fraco entre o subsistema e seus clientes. Freqüentemente, os componentes num subsistema são fortemente acoplados. O acoplamento fraco permite variar os componentes do subsistema sem afetar os seus clientes.
   As Façades ajudam a estratificar um sistema e as dependências entre objetos. Elas podem eliminar dependências complexas ou circulares. Isso pode ser uma conseqüência importante quando o cliente e o subsistema são implementados independentemente.
   Reduzir as dependências de compilação é um aspecto vital em grandes sistemas de software. Você deseja economizar tempo através da minimização da recompilação, quando as classes do subsistema sofrem transformações. A redução das dependências de compilação com o uso de façades pode limitar a recompilação necessária para uma pequena mudança num subsistema importante. Uma façade também pode simplificar a migração de sistemas para outras plataformas, porque é menos provável que a construção de um subsistema exija construir todos os outros.
3. Não impede as aplicações de utilizarem as classes do subsistema caso necessitem fazê-lo. Assim, você pode escolher entre a facilidade de uso e a generalidade.

Implementação

Considere os seguintes aspectos quando implementar uma façade:

1. *Redução do acoplamento cliente-subsistema.* O acoplamento entre os clientes e o subsistema pode ser ainda mais reduzido tornando Façade uma classe abstrata com subclasses concretas para diferentes implementações de um subsistema. Então, os clientes podem se comunicar com o subsistema através da interface da classe abstrata Façade. Este acoplamento abstrato evita que os clientes saibam qual a implementação de um subsistema que está sendo usada.
   Uma alternativa ao uso de subclasses é configurar um objeto Façade com diferentes objetos-subsistema. Para customizar Façade simplesmente substitua um ou mais dos seus objetos-subsistema.
2. *Classes de subsistemas: públicas ou privadas?* Um subsistema é análogo a uma classe no sentido de que ambos possuem interfaces e de que ambos encapsulam algo – uma classe encapsula estado e operações, enquanto um subsistema encapsula classes. E da mesma forma que é útil pensar na interface pública e na interface privada de uma classe, podemos pensar na interface pública e na interface privada de um subsistema.
   A interface pública de um subsistema consiste de classes que todos os clientes podem acessar; a interface privada destina-se somente aos encarregados de estender o subsistema. A classe Façade naturalmente é parte da interface pública, porém, não é a única parte. Também outras classes do subsistema são usualmente públicas. Por exemplo, as classes Parser e Scanner no subsistema compilador são parte da interface pública.
   Tornar privadas as classes do subsistema seria útil, porém, poucas linguagens orientadas a objetos fornecem suporte para isso. Tradicionalmente, tanto C++ como Smalltalk têm tido um espaço global de nomes para classes. Contudo, recentemente o comitê de padronização de C++ acrescentou espaços de nomes à linguagem [Str94], o que lhe permitirá expor somente as classes públicas do subsistema.

Exemplo de código

Vamos imaginar mais de perto como colocar uma fachada num subsistema compilador.

O subsistema compilador define uma classe BytecodeStream que implementa um *stream* de objetos `Bytecode`. Um objeto `Bytecode` encapsula um código de bytes, o qual pode especificar instruções de máquina. O subsistema também define uma classe `Token` para objetos que encapsulam *tokens* na linguagem de programação.

A classe `Scanner` aceita um *stream* em caracteres e produz um *stream* de *tokens*, um de cada vez.

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

A classe `Parser` usa um `ProgramNodeBuilder` para construir uma árvore de análise a partir dos *tokens* de `Scanner`.

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

`Parser` chama `ProgramNodeBuilder` para construir a árvore de análise incrementalmente. Estas classes interagem de acordo com o *padrão* Builder (104).

```
class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...
```

```
    ProgramNode* GetRootNode();
private:
    ProgramNode* _node;
};
```

A árvore de análise é composta de instâncias de subclasses de `ProgramNode` tais como `StatementNode`, `ExpressionNode`, e assim por diante. A hierarquia `ProgramNode` é um exemplo do padrão Composite (160). `ProgramNode` define uma interface para manipular o nó do programa e seus descendentes, se houver.

```
class ProgramNode {
public:
    // program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // child manipulation
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    // ...

    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

A operação `Traverse` aceita um objeto `CodeGenerator`. As subclasses de `ProgramNode` usam esse objeto para gerar código de máquina na forma de objetos `Bytecode` num `BytecodeStream`. A classe `CodeGenerator` é um visitor (ver Visitor, 305).

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    // ...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& _output;
};
```

`CodeGenerator` tem subclasses, como `StackMachineCodeGenerator` e `RISCCodeGenerator`, que geram código de máquina para diferentes arquiteturas de *hardware*.

Cada subclasse de `ProgramNode` implementa `Traverse` para chamar `Traverse` nos seus objetos `ProgramNode` descendentes. Por sua vez, cada descendente faz o mesmo para os seus descendentes, e assim por diante, recursivamente. Por exemplo, `ExpressionNode` define `Traverse` como segue:

```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

As classes que discutimos até aqui compõem o subsistema compilador. Agora introduziremos uma classe `Compiler`, uma façade que junta todas estas peças. `Compiler` fornece uma interface simples para compilar código-fonte e gerar código de máquina para uma máquina específica.

```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};
```

```
void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

Essa implementação codifica de maneira rígida o tipo de gerador de código a ser usado, de modo que os programadores não especificam a arquitetura para a qual o código está sendo gerado. Isso pode ser razoável se for sempre a mesma arquitetura para a qual será gerado código. Porém, se esse não for o caso, poderemos querer mudar o constructor de `Compiler` para aceitar como parâmetro um `CodeGenerator`. Então, os programas poderão especificar o gerador a ser usado quando eles instanciarem `Compiler`. A fachada do Compilador pode também parametrizar outros participantes, tais como `Scanner` e `ProgramNodeBuilder`, o que acrescenta flexibilidade, mas também se desvia da missão do padrão Façade, que é simplificar a interface para os casos comuns.

Usos conhecidos

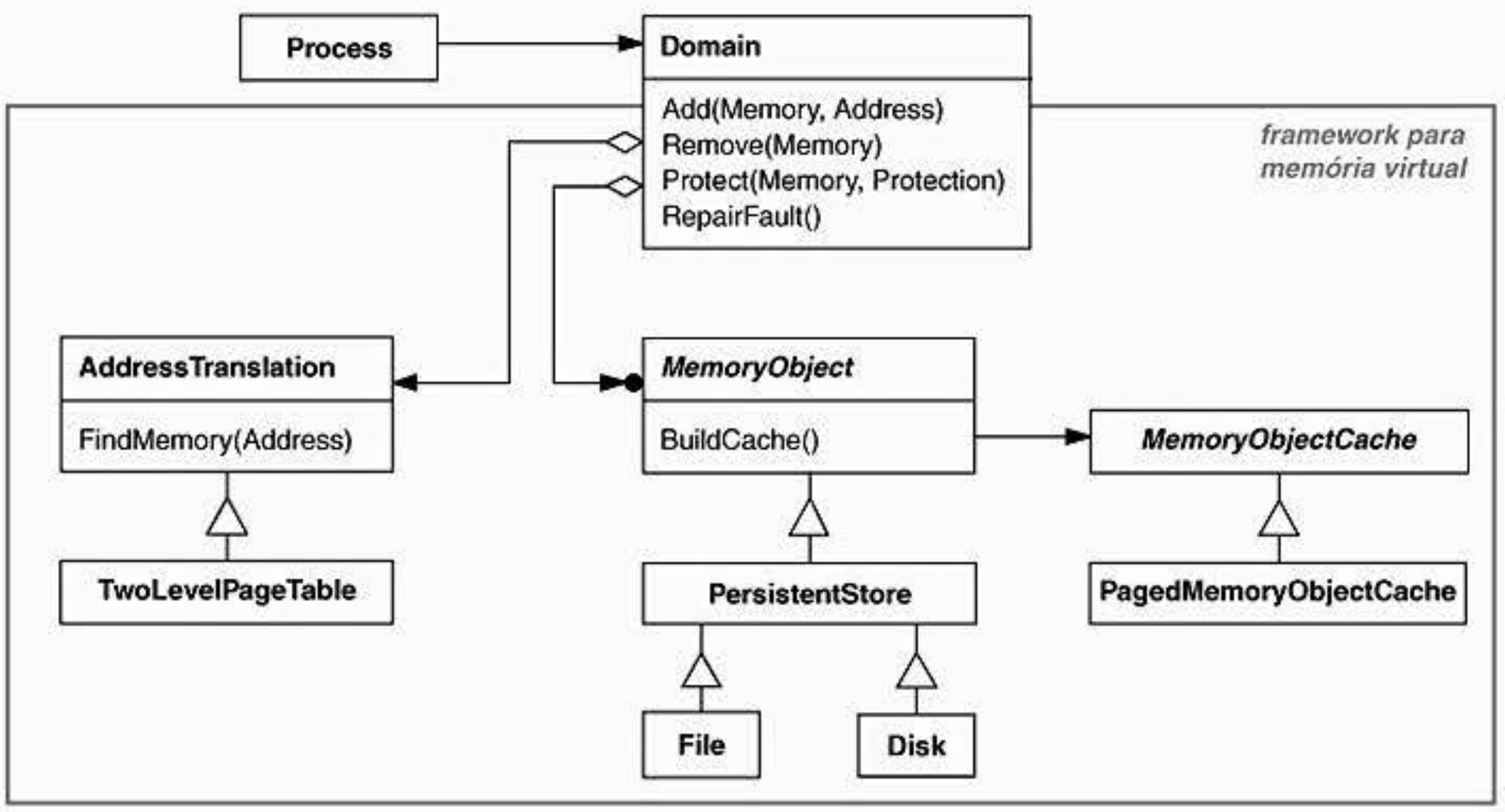
O exemplo de Compilador na seção Exemplo de código foi inspirado pelo sistema Compilador do ObjectWorks/Smalltalk [Par90].

No *framework* para aplicações da ET++[WGM88], uma aplicação pode ter incorporadas ferramentas de *browsing* para inspecionar os seus objetos em tempo de execução. Essas ferramentas para inspeção (*browsers*) são implementadas num subsistema separado que inclui uma classe Façade chamada "ProgrammingEnvironment". Essa fachada define operações, tais como InspectObject e InspectClass, para acessar os *browsers*.

Uma aplicação ET++ também pode ignorar o suporte incorporado para *browsing*. Nesse caso, ProgrammingEnvironment implementa essas solicitações como operações nulas; ou seja, não executam nada. Somente a subclasse ETProgrammingEnvironment implementa essas solicitações como operações que exibem os *browsers* correspondentes.

A aplicação não tem conhecimento se um ambiente de *browsing* está disponível ou não; existe acoplamento abstrato entre aplicação e o subsistema de *browsing*.

O sistema operacional Choices [CIRM93] usa fachadas para compor muitos *frameworks* num só. As abstrações-chave em Choices são processos, recursos de armazenamento e espaços de endereçamento. Para cada uma dessas abstrações existe um subsistema correspondente, implementado como *framework*, que suporta a probabilidade do Choices para uma variedade de plataformas diferentes. Dois desses subsistemas têm um "representante" (isto é, fachada). Esses representantes são FileSystemInterface (armazenamento) e Domain (espaços de endereçamento).



Por exemplo, o *framework* para memória virtual tem Domain como sua fachada. Um Domain representa um espaço de endereçamento. Ele fornece o mapeamento entre endereços virtuais e deslocamentos para objetos na memória, arquivos ou armazenamento de suporte. As principais operações no Domain suportam a adição de um objeto na memória em um endereço específico, a remoção de um objeto da memória e o tratamento de um *page fault*.

Como mostra o diagrama precedente, o subsistema para memória virtual usa os seguintes componentes internamente:

- MemoryObject (objeto de memória) representa um depósito de dados.
- MemoryObjetctCache acumula os dados de MemoryObject na memória física. MemoryObjectCache é, na realidade, um padrão Strategy (292) que localiza a política de *caching*.
- AddressTranslation encapsula a tradução de endereços do hardware.

A operação RepairFault é chamada sempre que uma interrupção de *page fault* ocorre. Domain acha o objeto de memória no endereço que está causando a falta da página e delega a operação RepairFault para o cache associado com aquele objeto de memória. Domains podem ser customizados mudando os seus componentes.

Padrões relacionados

Abstract Factory (95) pode ser usado com Façade para fornecer uma interface para criação de objetos do subsistema de forma independente do subsistema. Uma Abstract Factory pode ser usada como uma alternativa a Façade para ocultar classes específicas de plataformas.

Mediator (257) é semelhante a Façade no sentido em que ele abstrai a funcionalidade de classes existentes. Contudo, a finalidade de Mediator é abstrair comunicações arbitrárias entre objetos-colegas, freqüentemente centralizando funcionalidades que não pertencem a nenhum deles. Os colegas do Mediator estão cientes do mesmo e se comunicam com o Mediator em vez de se comunicarem uns com os outros diretamente. Por contraste, uma fachada meramente abstrai uma interface para objetos subsistemas para torná-los mais fáceis de serem utilizados; ela não define novas funcionalidades e as classes do subsistema não têm conhecimento dela.

Normalmente, somente um objeto Façade é necessário. Desta forma, objetos Façade são freqüentemente Singletons (130).