Facade Pattern

Tomás Felipe Melli

July 15, 2025

Índice

1 Intent **2**

2 Motivation **2**

3 Applicability **2**

4 Structure **3**

5 Participants **3**

6 Collaborations **3**

7 Consequences **3**

8 Implementation **3**

9 Sample Code **4**

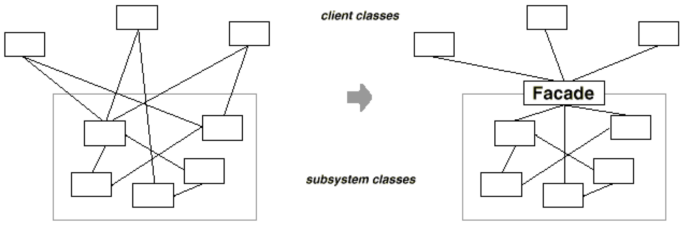
1 Intent

Proveer una interfaz unificada a un conjunto de interfaces en un subsistema. El patrón **Facade** define una interfaz de más alto nivel que hace que el subsistema sea más fácil de usar.

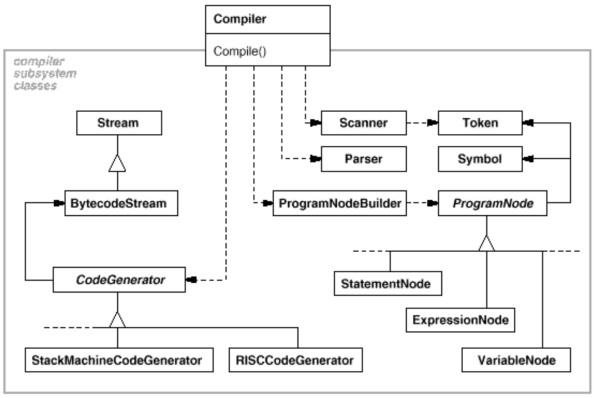
2 Motivation

Dividir un sistema en subsistemas ayuda a reducir la complejidad general. Un objetivo de diseño común es minimizar la comunicación y las dependencias entre subsistemas. Una forma de lograrlo es introduciendo un objeto **facade** que proporcione una interfaz simplificada a las funcionalidades del subsistema.

Por ejemplo, un entorno de programación que ofrece acceso a su subsistema de compilación puede tener clases como `Scanner`, `Parser`, `ProgramNode`, `BytecodeStream`, y `ProgramNodeBuilder`. Aunque algunas aplicaciones especializadas pueden necesitar acceder a estas clases directamente, la mayoría solo necesita una interfaz simple para compilar código.



Para simplificar esta interacción, se puede usar una clase `Compiler` como **facade**, que agrupa el comportamiento del compilador detrás de una interfaz simple y coherente. Esta clase no oculta totalmente el subsistema, pero sí lo abstrae lo suficiente para facilitar su uso.

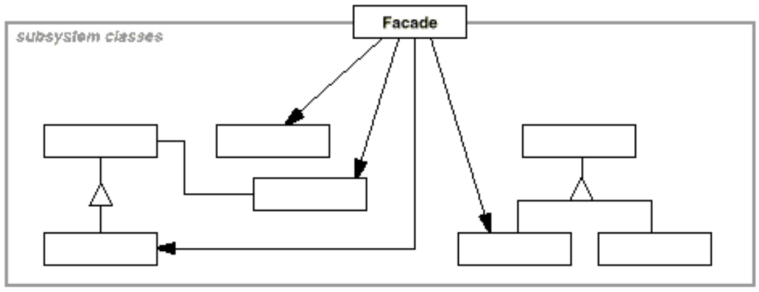


3 Applicability

Usar el patrón Facade cuando:

- Se desea una interfaz simple para un subsistema complejo.
- Hay muchas dependencias entre clientes y clases de implementación internas. Facade ayuda a reducir ese acoplamiento.
- Se quieren estructurar subsistemas en capas. Un Facade puede definir un punto de entrada por nivel, simplificando las dependencias entre ellos.

4 Structure



5 Participants

- **Facade (Compiler)**: Sabe qué clases del subsistema son responsables de una solicitud y delega las peticiones del cliente a los objetos apropiados.
- **Clases del subsistema (Scanner, Parser, ProgramNode, etc.)**:
  - Implementan la funcionalidad del subsistema.
  - Manejan las solicitudes asignadas por el Facade.
  - No conocen ni dependen del Facade.

6 Collaborations

- Los clientes interactúan con el subsistema a través del Facade.
- El Facade puede traducir sus solicitudes antes de delegarlas a las clases internas.
- Los clientes que usan el Facade no necesitan acceder directamente a los objetos internos del subsistema.

7 Consequences

- **Reduce la complejidad del cliente**: lo protege de los detalles del subsistema y reduce la cantidad de objetos con los que debe tratar.
- **Promueve el bajo acoplamiento**: facilita el cambio de implementación interna del subsistema sin afectar al cliente.
- **Mejora la portabilidad y mantenibilidad**: especialmente útil en sistemas grandes donde se busca minimizar la recompilación al cambiar clases del subsistema.
- **No limita el acceso completo**: los clientes avanzados aún pueden acceder directamente a las clases del subsistema si lo necesitan.

8 Implementation

- El Facade puede ser una clase abstracta, permitiendo diferentes implementaciones del subsistema sin que el cliente las conozca.
- Se puede configurar el Facade con diferentes objetos internos, reemplazándolos sin cambiar su interfaz externa.
- Las clases públicas del subsistema constituyen su interfaz accesible. Las privadas (cuando el lenguaje lo permite) son solo para uso interno.

9 Sample Code

Contexto del problema

Tenemos un subsistema de compilación compuesto por muchas clases que trabajan juntas: `Scanner`, `Parser`, `ProgramNode`, `CodeGenerator`, etc. Estas clases están fuertemente acopladas y ofrecen interfaces detalladas pero complejas. Muchos usuarios solo quieren compilar código fuente sin preocuparse por los detalles de parsing o generación de código.

Facade propone encapsular estas interacciones en una clase de nivel superior: `Compiler`, que actúa como interfaz unificada hacia el subsistema. El objetivo es simplificar el uso común, sin restringir el acceso total al subsistema para los usuarios avanzados.

Clases del subsistema

```
// Scanner: convierte caracteres en tokens
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();
    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

```
// Parser: construye un  rbol  de sintaxis usando un builder
class Parser {
public:
    Parser();
    virtual ~Parser();
    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

```
// ProgramNodeBuilder: aplica patr n Builder para crear nodos
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;
};
```

```
// ProgramNode: jerarqu a compuesta para representar el AST
class ProgramNode {
public:
    virtual void GetSourcePosition(int& line, int& index);
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

```
// CodeGenerator: visitante que genera c digo m quina
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
protected:
    CodeGenerator(BytecodeStream&);
    BytecodeStream& _output;
};
```

Implementación de Traverse en nodos

Cada nodo del árbol llama recursivamente a `Traverse` en sus hijos y al método `Visit` correspondiente del `CodeGenerator`.

```
void ExpressionNode::Traverse(CodeGenerator& cg) {
    cg.Visit(this);
    ListIterator i(_children);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

Clase Facade: Compiler

```
class Compiler {
public:
    Compiler();
    virtual void Compile(istream&, BytecodeStream&);
};
```

Uso del patrón

`Compiler::Compile` encapsula toda la lógica del proceso de compilación, proporcionando una interfaz sencilla:

```
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);
}
```

Resultado

Con esta fachada, los usuarios pueden compilar un programa simplemente creando un objeto `Compiler` y llamando a `Compile`, sin necesidad de entender o coordinar manualmente las interacciones entre clases internas del subsistema de compilación.

La clase `Compiler` facilita los casos comunes, pero no impide el uso avanzado de las clases internas para quienes lo necesiten.